UNIVERSITÉ
DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l’utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik
und Rechnerorganisation

Nancy-Université
Institut National Polytechnique de Lorraine
Centre de Recherche en Automatique de Nancy
École Doctorale IAEM Lorraine

PhD-Thesis (Cotutelle de Thèse)

A Hierarchical Modelling and Evaluation Technique for Safety Critical Systems

Michael Pock

for obatining the titles:

Doktor der Naturwissenschaften (Dr. rer. nat.)

and

Docteur de l'institut National Polytechnique de Lorraine
Spécialité Automatique et Traitement du signal

Jury:

President:	Martin Bichler	TU München
Chairmen:	Jean-François Aubry	INPL Nancy
	Arndt Bode	TU München
Consultants:	Markus Siegle	Universität der Bundeswehr München
	Antoine Rauzy	École Polytechnique ParisTech
Examiners:	Frédéric Kratz	ENSI de Bourges
	Hans Joachim Bungartz	TU München

Contents

I Introduction 5

1 Preliminaries 7
- 1.1 Motivation . . . 7
- 1.2 Contribution . . . 9
- 1.3 Organisation of this thesis . . . 10

2 Foundations 11
- 2.1 Reliability and Safety Modelling . . . 11
  - 2.1.1 Definitions . . . 11
  - 2.1.2 Probability Distributions . . . 14
  - 2.1.3 Combinatorial Methods . . . 15
  - 2.1.4 State Based Methods . . . 17
  - 2.1.5 Hybrid Approaches . . . 21
- 2.2 BDD-Techniques . . . 23
  - 2.2.1 ROBDDs . . . 24
  - 2.2.2 ZBDDs . . . 26
  - 2.2.3 BEDs . . . 28
- 2.3 Related Work . . . 29
  - 2.3.1 SafeMe . . . 29
  - 2.3.2 Hip-HOPS . . . 30
  - 2.3.3 Multi-Phased Missions . . . 30
  - 2.3.4 Mode Automata . . . 31
  - 2.3.5 Fault tolerant stack processor . . . 31
  - 2.3.6 CASPA . . . 32

3 Information Flow Diagrams 33
- 3.1 The High Level Model . . . 33
- 3.2 The Low Level Model . . . 36
- 3.3 Generation of the Global Lists . . . 39
- 3.4 Model of the Hardware Resources . . . 42
- 3.5 Evaluation of the IFD-model . . . 44

II Hierarchical Modelling and Evaluation 47

4 Formalisation of the IFD-Model 49
4.1 Formalisation and Simplification of the Low Level Model . . . 50
4.2 Formalisation and Simplification of Blocks 52
4.3 Overview of the Modifications 57
4.4 Overview of the New Formalism 58
  4.4.1 The high level formalism 58
  4.4.2 The low level formalism 59
  4.4.3 Combining high and low level 61
  4.4.4 Modelling basic components 62

5 Developed Algorithms 65
5.1 Boolean Interpretation of Local Expressions 65
5.2 ZBDDs for Simple Serial Systems 67
5.3 ZBDDs for Generalised Serial IFDs 72
5.4 BDDs for Non-Serial IFDs 75
5.5 Illustrative Example 77
5.6 Quantitative Evaluation 82

6 Implementation 85
6.1 Defined Datatypes 85
  6.1.1 The IFD-model 85
  6.1.2 Expressions 86
  6.1.3 The BDD 88
6.2 Reading and Preparation of Models 90
  6.2.1 The File Format 90
  6.2.2 Creation of the Initial IFD-Model 92
6.3 BDD-Creation Process 94
6.4 Quantified Solving Process 96

7 Case Studies 99
7.1 Remote Redundancy 99
  7.1.1 The Basic System Architecture 99
  7.1.2 The Conservative Fault-Tolerant Architecture 100
  7.1.3 The Remote Redundancy Architecture 101
  7.1.4 The IFD-Models for the Two Architectures 103
  7.1.5 Evaluation of the Models 107
7.2 Chemical Reactor 109
  7.2.1 The System Architecture 109
  7.2.2 The IFD-Models 110

7.2.3 Comparison of the Two Models 113

8 Evaluation 115

8.1 Complexity Analysis 115

8.1.1 Simple Serial Systems 115

8.1.2 General Serial Systems 116

8.1.3 Non-Serial Systems 117

8.2 Measurements 117

8.2.1 Case Studies 118

8.2.2 Scalable Models 119

III Conclusions 125

9 Summary 127

9.1 Modifications 127

9.2 Algorithms 128

9.3 Implementation 128

9.4 Case Studies 128

10 Outlook 131

10.1 Performance Improvement 131

10.2 The Bertholon Model 131

10.3 Further Generalisation 131

10.4 Graphical User Interface 132

10.5 Dependencies 132

10.6 LARES 132

IV Appendix 135

A Abbreviations 137

B Glossary 139

C Examples 145

C.1 Conservative Redundancy 145

C.2 Remote Redundancy 148

C.3 Chemical Reactor 150

D IFD-File Example 153

Part I

Introduction

Chapter 1

Preliminaries

Today, many system functions have been automated, for example in power plants, production facilities or in the transportation domain. While this automation increases productivity and efficiency dramatically, it can cause problems as well. Many of these automated systems have safety critical constraints. A failure of these system can lead to accidents with severe consequences like many causalities or long lasting environmental damages.

To avoid failures, many techniques can be used in the design process of systems and components. It is however impossible to build a "perfect" system, so the system design has to take into account that certain parts of it can and will fail. By using such a design, the probability of catastrophic failures can be minimised. Before using a safety critical system, trustful data about its dependability is needed. Testing such systems is not an option. As failures should occur only very rarely, tests would need to run for a very long time for accurate results. Besides, running a safety critical system without any certain knowledge about its dependability is highly risky.

This is the reason why models are used for evaluating such systems. Many different possibilities for modelling exist. One of them is presented in this work. Now we will explain why our approach is necessary and what contributions we made.

1.1 Motivation

In order to avoid dangerous accidents safety critical systems are redundant. If one component fails, normally a spare component can take over its function so that the overall system will not fail. Furthermore, in many cases a safety function is activated. This safety function tries to bring the system into a safe state when a subsystem is not working correctly any more and could threaten the integrity of the global system. In most applications this safety function shuts down the system. In this case the system is not working any more, but at least it does not cause any accidents.

These safety functions can fail in two different ways. It can happen that the safety functions are not activated in presence of demand, leading to serious accidents. There are many famous examples for such failures, for example the nuclear disaster in Fukushima [9], in which an earthquake and the following tsunami breached all safety barriers. Due to this accident, the area around the nuclear power plant is now uninhabitable, and many workers at the power plant were exposed to high values of radiation. Another example is the crash of the flight AF 447 from Rio de Janeiro to Paris [7], caused by frozen speed indicators. 228 people were killed in this plane crash. Of course, there are also many accidents on a much smaller scale: machines which cut off a finger of an operator accidently or a car which crashes due to a failure of the ABS-system. It is very important to minimise the probability of such accidents with such serious consequences.

Besides causing serious accidents, safety critical systems can also fail by activating their safety functions in absence of demand. In this case, it will lead to a so called *Spurious Trip*. Normally this is not dangerous, as it will only lead to the unavailability of the system. But still it is annoying and can be very expensive for the owner of the system. A failure of a robot in a production line could stop the whole production of the factory which can be economically disastrous if this happens regularly. Note that in some cases spurious trips can be dangerous too, like the crash of an Airbus A320-211 in Warsaw in 1993. The safety mechanism which should prevent the unwanted activation of the engine thrust reverser in the air prevented its activation as the runway was wet and the wheels of the plane were not turning due to aquaplaning. This was seen as sign that the plane has not landed yet, so the engine thrust reverser was not activated and the plane crashed at the end of the runway, causing two fatalities[4].

Furthermore too many spurious trips can lead to decline of the safety culture of the operators. For example, a smoke detector which causes false alarms regularly will be either ignored or switched of completely, so that there is a increased risk that a real fire is not detected fast enough.

So it is important to include both failure modes into models of safety critical systems and their safety functions. Furthermore it is necessary to take into account that not only the global system itself, but also subsystems and basic components of the system can fail in several ways. For example, a sensor can give either too high or too low values, leading to different faulty behaviours. Besides that, it is necessary for models to be easy to understand and to design. Too complicated models are error prone - if they are used at all. So it is very important that the properties of safety critical systems can be included in the model in an not too complex way. Moreover, Large systems should be possible to handle. This can be done by using a hierarchical model, so that independent subsystems can be modelled on their own.

1.2 Contribution

In this thesis, a model taking into account the requirements presented in chapter 1.1 is used as base, the *IFD-model* [35]. The IFD-model is modified with two thoughts in mind: First, it should be improved in order to make it easier to understand and to use. Furthermore, this model should be transformable into a *Binary Decision Diagram* (BDD) [20].

The next step is to create algorithms for an efficient evaluation of the modified IFD-model. As this model includes all failure scenarios of the described system there is a state space explosion regarding the number of used components. A naive implementation would lead to an exponential complexity, so such an implementation would be of no practical use. Instead, different BDD-techniques are combined: *zero suppressed BDDs* (ZBDDs) [42] and *Boolean Expression Diagrams* (BEDs) [11] to avoid the combinatorial explosion.

To be able to apply BDD-techniques, the model itself is transformed into a Boolean form. This Boolean form is used as base for creating a BDD. This creation process itself is hierarchic and uses properties of the IFD-model for a very efficient evaluation.

To test the developed algorithms, they were implemented in a rudimentary software. This software is mainly meant as proof of concept - but it was also used for modelling several case studies. For example, we made a comparison of a novel concept for redundancy originally developed by the group of Klaus Echtle of the University of Duisburg-Essen with classical redundant architectures. Furthermore a chemical reactor was modelled, which was already used as example in the dissertation [35] of Karim Hamidi. Finally, the developed algorithms were evaluated regarding their performance. We made two different analysis: at first, the theoretical complexity was determined. Afterwards, measurements were taken based on the implementation.

The next chapters explain these steps in detail and demonstrate the power of the IFD-approach.

1.3 Organisation of this thesis

The thesis is structured as follows: Chapter 2 explains the foundations of dependability modelling and Binary Decision Diagrams. Chapter 3 presents the IFD-approach developed by Karim Hamidi, on which this work is based. Chapter 4 shows how the original IFD-model was modified, so that the algorithms presented in Chapter 5 can be applied. Chapter 6 explains how these algorithms were implemented. Chapter 7 presents two case studies and is followed by an evaluation of the work in Chapter 8. Chapter 9 will give a short summary followed by an outlook in Chapter 10.

Chapter 2

Foundations

This chapter explains important facts about two domains: reliability modelling and Binary Decision Diagrams. Both are necessary to understand the work presented in this dissertation. First some definitions in the domain of reliability modelling are given. Afterwards, several widely used possibilities to model systems for estimating their reliability are discussed. Afterwards a few related works are presented which have similarities with this thesis.
The next section explains several BDD-techniques. These techniques are used to avoid combinatorial explosions which can occur easily evaluating reliability models. Several versions of this widely used technique are presented and compared.

2.1 Reliability and Safety Modelling

This section explains the important base for reliability and safety modelling. First some definitions are given, followed by the introduction of two important widely used distributions (Exponential and Weibull distribution) for modelling single components. Afterwards, three general ways to model the reliability of more complex systems are compared: state based methods, combinatorial methods and hybrid models. These different approaches differ in intuitivity, computational demands and expressive power.

2.1.1 Definitions

There are several important terms and measures used in the domain of reliability modelling. They are used for both the whole system and single components of the system. These measures have to be defined in a clear way to avoid confusion.

First we have to clarify what system means in the context of this thesis. Here a *System* is an assembly of electronic, electrical, or mechanical components with interdependent functions, usually forming a self-contained unit. [8]
Reliability [62] itself is defined as the probability that a system will work satisfying for a specified time t. Let T be the random variable representing the lifetime of the system, $u(t)$ its probability density function and $U(t)$ its cumulative density function. Then the reliability, denoted as $R(t)$, can be defined as follows:

$$R(t) = Pr(T > t) = 1 - U(t) = \int_t^\infty u(x)dx$$

$U(t)$ is also called the *unreliability*, as it is equal to the probability that the system will fail during the specified time interval $[0, t]$.
The next important measure is the *Mean Time to Failure* [39], abbreviated as MTTF. It is defined as the expected value of the lifetime of the system and can be calculated by:

$$MTTF = \int_0^\infty R(t)dt$$

Many systems are repairable. In such systems, also a *Mean Time To Repair* (MTTR)[50] can be defined. It is defined as the expected value for the length of the repairing process, including the failure detection.
The next important measure is the *availability*, denoted as $A(t)$. It is the probability that the system is working at the time t. For non-repairable systems $A(t)$ is obviously equal to $R(t)$. For repairable systems, the following inequality holds: $A(t) \geq R(t)$.
For repairable systems the so called *steady state availability*, denoted as A, is especially interesting. It is defined as:

$$A = \lim_{t\to\infty} A(t).$$

If we assume that repairs are always perfect, i.e. the system is as good as new after each repair, the steady state availability can be calculated with the following equation: [56]

$$A = \frac{MTTF}{MTTF+MTTR}.$$

We can distinguish four different classes of failures [49]. *Crash failures* lead to an immediate and uncontrolled shutdown of a system. A system with an *omission failure* gives no results or orders at all. Systems with *Timing failures* give the correct results or reactions, but later as specified. If a *Byzantine*

failure occurs, the system gives wrong results or orders.
These different types of failures have different consequences. While some failures will just lead to unavailability other failures can cause dangerous accidents. The severity of these consequences depends on the tasks and the architecture of the system and whether failures are detected so that safety steps like a controlled shut down can be taken. Note that some failures can not be detected at all. Also the detection process itself can fail.
This leads to the measure of *safety* $S(t)$ [55][6]. A system is safe if it is not causing harm, injuries or damage. We can also define it more formally as the probability that a system is either working correctly or is shut down. For non-repairable systems the following inequality holds: $S(t) \geq R(t)$.
This definition has an interesting consequence: the safety can be increased by shutting down the system, leading to less reliability. For real fail-safe systems this means that a very cautious approach where the system is shut down as soon as there is a tiny doubt about its integrity will lead to a very safe, but very unreliable system. It is even possible to create a system with $S(t) = 1$ by never starting it at all. Obviously, such a system does not make any sense. But it shows that analysing the safety without taking into account the reliability is not enough. The following table depicts a safety-availability matrix for a fail-safe system showing four different operating states. In this context *safe* and *unsafe* describe the state of the system's environment based on the specification, not the state of the system itself. *Unsafe* demands for a safety shutdown. *Available* and *unavailable* describe if the system is currently working or if it has been shut down.

	Unavailable	Available
Safe	Spurious trip	Working correctly
Unsafe	Safety shutdown	Failure on demand

This matrix has four different cells. The overall system is reliable if it is in the state *Working correctly*. It is safe if it is in one of the state *Working correctly*, *Spurious trip* or *Safety shutdown*.
In two of the four states the system does not work like specified, i.e the safety functions of the system have failed: *Spurious trip* and *Failure on demand*. These two failure states have large differences. Spurious trips are annoying and should occur only rarely as they lead to unnecessary unavailability and costs, but at least they are not dangerous. In contrary, failures on demand should hopefully never happen at all as they can cause severe accidents. So, in order to analyse the safety functions of a system, it is necessary to distinguish these two failure states. For evaluating the safety functions of fail-safe systems, two different probabilities are defined, the *probability of*

failure on demand (PFD(t)) and the *probability of a spurious trip* (PFS(t)) [35].

- PFD(t) is the probability that a failure on demand occurs during the time interval [0,t].
- PFS(t) is the probability that a spurious trip occurs during the time interval [0,t].

With these two measures, it is possible to evaluate fail-safe systems and their safety functions.

2.1.2 Probability Distributions

In order to evaluate reliability models in a quantitative way, it is necessary to model the possible failures quantitatively. For this task probability distributions are needed, so this subsection presents the two most important ones. One of the most common probability distributions is the *exponential distribution.* It has only one parameter λ, representing the failure rate of the component or system. Its probability density function is

$$f(t) = \lambda e^{-\lambda t}.$$

This leads to the following reliability and unreliability:

$$R(t) = e^{-\lambda t}$$
$$U(t) = 1 - e^{-\lambda t}$$

The failure rate λ is constant, i.e. it is independent of the component's or system's age. This a reasonable assumption for many components, for example electronic circuits which have almost no wearoff. Also external events like cosmic rays can be modelled with the exponential distribution. It is not suitable for modelling components or systems with non-constant failure rates for their whole lifetime, though. Examples for such components are motors or valves where mechanical wearoff has a strong influence on the reliability. But even for these components and systems the exponential distribution can be used to approximate their reliability. For many components first the failure rate is decreasing fast as the early failures are mostly caused by faulty production ("infant mortality"). Then the failure rate stays constant for some time until the wearoff really kicks in. The curve of such a failure rate is also called the *bath tube curve*[39]. The exponential distribution can be used for the time in which the failure rate is constant. This is a good approximation as in most systems with high reliability demands components are introduced only after strict testing and replaced before their failure starts to rise due to wearoff. So for their lifetime in the system the components have a constant failure rate.

One of the main advantages of the exponential distribution is that it simplifies the calculation processes. It also allows the use of Markov Chains.

When the ageing effects have to be included in the model, the *Weibull distribution*[67] can be used. Its probability density function is

$$f(t) = \frac{\beta t^{\beta-1}}{\eta^{\beta}} e^{-(t/\eta)^{\beta}}$$

where $\beta > 0$ is the shape parameter and $\eta > 0$ is the scale parameter. This leads to the following reliability:

$$R(t) = e^{-(t/\eta)^{\beta}}$$
$$U(t) = 1 - e^{-(t/\eta)^{\beta}}$$

Note that sometimes the parameter $\lambda = \eta^{-1}$ is used instead of η.

Depending on β, the failure rate of the Weibull distribution is changing with time. For $\beta < 1$, the failure rate decreases. For $\beta = 1$ it stays constant, i.e. it is equivalent to an exponential distribution with the failure rate η^{-1}. If $\beta > 1$, the failure rate increases. So the Weibull distribution can be used to model each phase of the bathtub curve. The calculus with the Weibull distribution is more complicated than for the exponential distribution, though.

2.1.3 Combinatorial Methods

In this section we present several methods to model systems consisting of several components. Basically, there are two different classes of modelling formalisms: combinatorial and state based formalisms. This subsection presents two combinatorial formalisms. Combinatorial or Boolean formalisms are formalisms which create a model equivalent to a Boolean expression. They can be evaluated with simple algorithms well known in the domain of stochastic. The most common combinatorial (also called Boolean) formalisms are *Fault Trees* (FTs, [58]) and *Reliability Block Diagrams* (RBDs, [59]). Both are based on graphs for representing the modelled system.

Fault trees are trees in which the leafs represent the components of the system. The inner nodes of the tree contain the Boolean operators *and* and *or* and are called *and-gates* and *or-gates*. An example is shown in Figure 2.1.

The system modelled by this fault tree is a Triple Modular Redundancy-system(TMR-system) [31], consisting of three identical sensors S1, S2 and S3

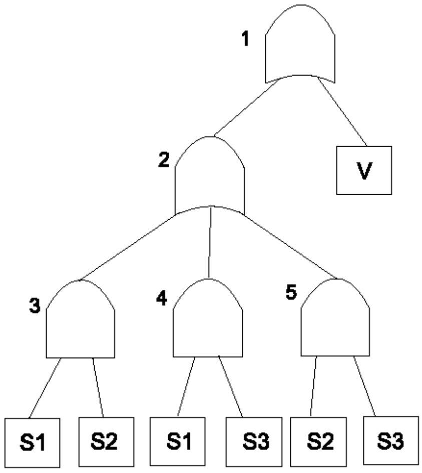

Figure 2.1: A fault tree for a TMR-system

and a voter V. The voter takes the three results of the sensors and chooses the median. The system will fail if either the voter or at least two of the three sensors fail.

The root of the fault tree (node one) and node two are or-gates, while the nodes three, four and five are and-gates. Basically, it represents a Boolean expression consisting of variables for the components. These variables are true if their corresponding components fail. If the expression for the fault tree is satisfied, the system fails.

Reliability Block Diagrams are an alternative for fault trees. RBDs are graphs with two special nodes s and t. The vertices of the RBD represent the system's components. The system works correctly if there is a working path from s to t. An RBD for the TMR-system is depicted in Figure 2.2.

Like for Fault Trees, a Boolean expression describing the system can be extracted from the RBD. If this expression is satisfied, the system is working correctly.

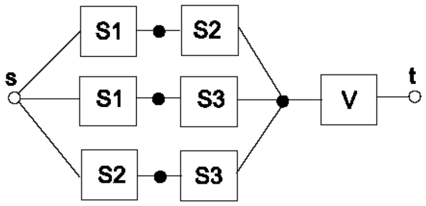

Figure 2.2: A reliability block diagram for a TMR-system

Both Fault Trees and Reliability Block Diagrams can be evaluated in several ways. Qualitative analysis are possible with minimal cut sets. These can determine which components have to fail, so that the overall system will fail. A quantitative evaluation for calculating reliability measures like availability is possible, too. If failure probabilities for all components are known and all the Boolean variables are stochastically independent, the failure probability of the overall system can be calculated. Also it is possible to do qualitative analysis to determine which component failures will lead to a global system failure by using minimal cutsets. For these purposes, Binary Decision Diagrams are created. Section 2.2 will explain this process in more details.

Combinatorial reliability modelling has several advantages. First, the models are easy to learn and to master. A deep knowledge about stochastic processes is not a mandatory for creating fault trees or RBDs. It is sufficient to know the system and its behaviour.

Furthermore, both RBDs and FTs can be created hierarchically. It is possible to create a model of the global system architecture in which several subsystems are included as single components. Afterwards, these subsystems can be modelled themselves as RBDs or FTs. In the end, the subsystems in the general model can be replaced by the appropriate RBDs or FTs, leading to a large and detailed model.

Finally, for evaluating combinatorial formalisms efficient algorithms exist, so less computing power is needed compared to state based formalisms. The main reason for this is that these formalisms can be transformed into the very efficient BDDs.

The combinatorial formalisms have some limits, though. At first, each component has only two states, failed or working. Neither it is possible to include different failure modes nor these formalisms can describe a continuous state (E.g. "Component is working at 62% capacity").

For a quantitative evaluation it is necessary that there are no stochastic dependencies in the model. For many systems this is not true, though. Common cause failures, failure propagation, warm/cold standby or shared repairs lead to stochastic dependencies between the variables in a FT or RBD. These effects can not be modelled with combinatorial formalisms, but they occur often in real-life applications. So these effects have either to be neglected, or more powerful methods have to be applied.

2.1.4 State Based Methods

A more powerful alternative to combinatorial formalisms are state based formalisms . The two most common methods are *Markov chains* (MCs) and *stochastic Petri nets.*

Markov chains[57] are sequences of random variables $X_1, X_2, ...X_n$ with the

Markov property. This means that the next state in the chain depends only on the current state:

$$Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x | X_n = x_n)$$

Markov chains are depicted as directed graphs in which the nodes represent the different states and the edges represent the transitions between the states. The transitions have different rates based on an exponential probability distribution. Figure 2.3 depicts a Markov chain representing a TMR-system:

The Markov chain has eight different states. The system is in the states S0, S1, S2 or S3 if the voter is working and zero, one, two or three sensors failed. The system is in the states S0V, S1V, S2V or S3V if the voter and zero, one, two or three sensors failed. The transition rates are based on the failure rates λ_S and λ_V and the repair rates μ_S and μ_V of the sensors and the voter.

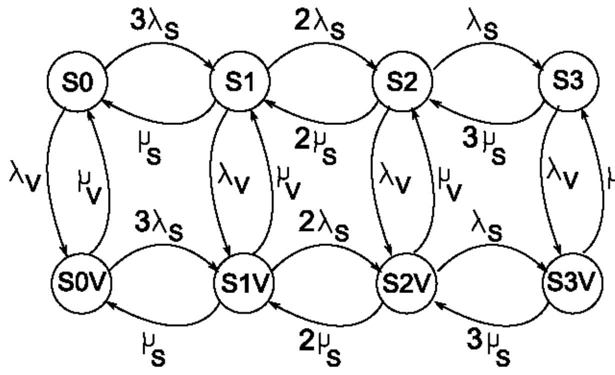

Figure 2.3: A Markov chain for a TMR-system

Markov chains can be evaluated regarding the probabilities that the system is in a certain state at time t. This is the *transient evaluation.* Furthermore, it is possible to calculate the probabilities for $t \to \infty$, the so called steady state evaluation. Reliability and Safety can be calculated by summing up the probabilities of the reliable respective safe states. For example, the TMR-system is working if it is in the states S0 or S1.

One possibility to get the probabilities for the transient and the steady state analysis is a Monte Carlo simulation [34]. The results are not exact, though. Their accuracy depends on the number of simulation runs. Furthermore the accuracy is lower for states which occur only very rarely. So for very reliable systems in which failures are very improbable a lot of simulation runs are needed to receive an accurate result.

For Markov chains, there is an alternative to the simulation. It is possible to solve them analytically, both for the transient and the steady state evaluation. At first, the transient evaluation is explained. Be $S1, S2, ..., Sn$ the states of a Markov chain, λ_{ij} the rate of the transition from the state Si to the state Sj and $p_i(t)$ the probability that the Markov chain is in the state Si at time t. For each state Sk with incoming transitions from the states $Si_1, ..., Si_l$ and outgoing transitions to the states $So_1, ..., So_m$ the following equation holds:

$$\frac{d}{dt}p_k(t) = \sum_{a=1}^{l} \lambda_{i_a k} \cdot p_{i_a}(t) - \sum_{b=1}^{m} \lambda_{k o_b} \cdot p_{o_b}(t)$$

For a Markov chain with n states, n such equations create a *Differential Equation System* (DES). This DES can be solved numerically in order to determine the probabilities of each state in the Markov chain.

The solution for the steady state case is even simpler. Be $p_i = \lim_{t \to \infty} p_i(t)$. For each state Sk with incoming transitions from the states $Si_1, ..., Si_l$ and outgoing transitions to the states $So_1, ..., So_m$ the following equation holds:

$$\sum_{a=1}^{l} \lambda_{i_a k} \cdot p_{i_a} = \sum_{b=1}^{m} \lambda_{k o_b} \cdot p_{o_b}$$

Furthermore, it holds obviously: $\sum_{i=1}^{n} p_i = 1$. Together with the n equations for the states a linear equation system can be defined, consisting out of $n + 1$ equations for n variables. The linear equation systems can be evaluated for determining the probabilities for all the states.

Markov Chains have several advantages over combinatorial formalisms. They can model different states for components and the global system. They can also include effects like common cause failures or failure propagation by adding fitting transitions in the chain. Effects like delayed repair or different standby strategies can be considered in MCs, too. In general, they are much more powerful than FTs or RBDs.

The powerfulness has its disadvantages, though. MCs can not be created hierarchically. In the FT- or RBD-model of the TMR-system, it is easy to replace a component by a more complex system or to add a single component. MCs can not be changed that easy as the state space is altered dramatically by such a step. Besides that, the size of the MC grows exponential with the number of components in the original system. For the TMR-system, a MC only representing the three sensors would need fours states. Adding the voter leads to eight states. For this reason, it is impossible to create MC-models for large systems by hand. Markov chains also lack intiutivity. It is almost impossible to recognize the original system architecture of a system by just looking at its Markov model. So modelling with MCs is quite error prone. Another widely-used state based formalism are *stochastic Petri nets* (SPNs, [45]). A SPN is a 4-tuple (S, T, W, M_0) with:

- S is a finite set of places
- T is a finite set of transitions
- $S \cap T = \{\}$
- $W : (S \times T) \cup (T \times S)$ is a set of arcs. Arcs connect either places to transitions or transitions to places.
- M_0 is the initial marking of the stochastic Petri net.

Figure 2.4 shows a SPN modelling the TMR-system.

The places of the stochastic Petri net can contain markings. Markings can move around the net by *firing* the transitions. A transition can be fired if all the places at the source of each incoming arc (*input places*) contain at least one marking. After firing the transition, each input place loses one marking while the places at the targets of the outgoing arcs of the transition (*output places*) gain a marking. A transition is *active* if it can be fired. If several different transitions are active, it is not determined which one will be fired in the next step. The probability of each transition depends on the rates defined for the transitions, hence the name stochastic Petri net.

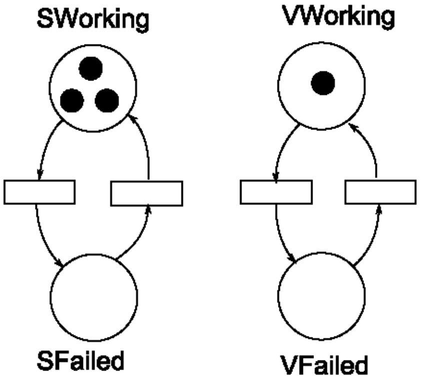

Figure 2.4: A stochastic Petri net for a TMR-system

Unlike Markov chains, the probability distributions for the rates of the transitions of SPNs are not restricted. This allows to model more complex system or component behaviour.

For a quantified evaluation of SPNs, the so called *reward rates* are needed.

The example in Figure 2.4 has four places and four transitions. The places *SWorking* and *SFailed* show how many sensors are working or have failed by the number of markings. The places *VWorking* and *VFailed* show if the voter has failed or not. The reward rate describing the global system is $SFailed \geq 2 \wedge VFailed = 1$. The system will fail if either the voter or two sensors fail. The probability of a system failure can be calculated by calculating the probability that the reward rate is fulfilled.

In general, the probabilities for the reward rates can be estimated by a Monte-Carlo-simulation. For the special case that the rates of the transitions are exponential, it is possible to transform the SPN to a equivalent MC which can be solved analytically [41]. Both ways for the evaluation have similar properties like the evaluation for MCs. A simulation for rare events have to run a long time. Converting a large SPN to a MC is not always faster, though. As MCs grow exponential with the number of components, the calculation costs are exponential too.

Stochastic Petri nets are a very powerful tool for modelling the reliability of systems. Non-Boolean behaviour can be modelled as well as dependencies between different components in the system[59]. They are easier to handle than Markov Chains, as they support modularized modelling to a certain degree. There are limits to this, though: New components without dependencies can be added easily, while adding dependencies into an existing model can lead to a total different structure of the SPN.

While being more intuitive than MC, SPNs can still not compete with the high level approaches of RBDs and FTs regarding the understandability. Creating and understanding models based on Petri nets is more difficult than using combinatorial formalisms. Especially for large models this can be a problem.

2.1.5 Hybrid Approaches

While state based formalisms are quite powerful, they are rarely used for modelling the Reliability or Safety of systems in industrial applications. These formalisms lack the simplicity and intuitivity of their combinatorial counterparts. So several formalisms were developed to support a high level modelling process like with FTs and RBDs, but which also allow to model dependencies and non-Boolean behaviour in an easy way.

One possibility to create a powerful but intuitive model is to enrich existing high level mechanisms like FTs and RBDs with new constructs for certain non-Boolean behaviours or dependencies. For solving these models, state based models are generated and solved automatically. The designer of the formers does not need to know anything about the latter. Examples for this technique are Dynamic Fault Trees (DFTs, [25]), used in the tools DIFtree [24] and Galileo [23] or the tool OpenSESAME [66].

Dynamic Fault Trees are fault trees with additional gates for different kind of standby modes (cold, warm, hot). Furthermore, they contain Priority-And and Priority-Or-gates. The OpenSESAME-model is based on an RBD, but inter-component dependencies can be added in extra failure dependency diagrams. Besides that, repair groups can be defined to model shared repair resources, and like in DFTs different kind of standby modes can be included. To solve these models, Markov chains (for DFTs) or Petri nets (for OpenSESAME) are created. The structure of these low-level models is based on the kind of dependency or behaviour which the high-level formalism has specified. It is neither possible nor necessary for the user to change the low-level models. The advantage of this method is that the user does not need to know anything about the low-level mechanisms. The disadvantage is that only dependencies and non-Boolean behaviour included in the high-level formalism can be modelled. So these tools are limited in their usability. They were developed for special application domains. For example, OpenSESAME has been developed for modelling highly-available systems which can be easily evaluated with this tool. It does not support several failure modes or disjoint events which are necessary for modelling safety critical systems, though.

Instead of creating just a high-level model and generating a low-level model automatically, it is also possible to create a high-level formalism in which the components or events can be described closer by a low-level model. An example for this technique are the *Boolean Driven Markov Processes* (BDMPs, [17][18]). With BDMPs a Fault Tree with extensions similar to Dynamic Fault Trees is used to define the global system architecture. Single components can be added as classic Boolean variables, but it is also possible to add leafs in the tree which are described by State Space formalism. Furthermore, different parts of the Fault Tree can be linked with so called triggers, which allows to model the activation of different modes of the system. These triggers can be adapted using MCs. For using such tools, the modeller has still to use low-level formalisms. But at least only quite small systems have to be modelled by them, which is not so difficult. These models can be combined easily with the high-level model. An advantage of this method is that the modeller is not restricted by the tool which dependencies or non-Boolean behaviour he wants to include in his model.

Note that state based methods are not always necessary to describe dependencies or non-Boolean behaviour. Boolean expressions can describe quite a lot of these effects [58]. For example a component with two different failure states can be simply represented by two Boolean variables instead of by just

one. Of course, this leads to stochastic dependencies. If the component is represented by two Boolean variables for its two failure states, at most one of these variables may be true. For a quantified evaluation these dependencies have to be taken into account [69]. It is not sufficient to create a model in a standard FT- or RBD-tool which normally assume stochastic independence of all components.
In this thesis, a Boolean method will be used to describe components with several failure modes. It is just important to make sure that probability calculations are adapted accordingly.

2.2 BDD-Techniques

Binary Decision Diagrams (BDDs) are an efficient way to represent Boolean expressions. This section gives a short overview about different BDD-techniques and how these can be used in the domain of reliability modelling.
The base for BDDs is the *Shannon Decomposition*[58]. Be Φ_{X_i} resp. $\Phi_{\overline{X_i}}$ the expression which results from substituting the variable X_i in the Boolean expression Φ with true resp. false. For a Boolean function $\Phi : \mathbb{B}^n \rightarrow \mathbb{B}$ it holds:

$$\Phi = (X_i \wedge \Phi_{X_i}) \vee (\overline{X_i} \wedge \Phi_{\overline{X_i}}) \tag{2.1}$$

where X_i is a variable of Φ.
This decomposition can be applied recursively on the remaining expressions Φ_{X_i} and $\Phi_{\overline{X_i}}$.
The decomposition is demonstrated for the expression $\Phi = (a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$.
At first, a is substituted:

$\Phi = (a \wedge \Phi_a) \vee (\overline{a} \wedge \Phi_{\overline{a}})$ with
$\Phi_a = \overline{b} \wedge c$
$\Phi_{\overline{a}} = c$
We continue by replacing b in Φ_a and $\Phi_{\overline{a}}$. As the latter does not contain the variable b, the resulting expressions are equal to $\Phi_{\overline{a}}$. For Φ_a, the decomposition is not so trivial:

$\Phi_{ab} = false$
$\Phi_{a\overline{b}} = c$
$\Phi_{\overline{a}b} = \Phi_{\overline{a}\overline{b}} = c$

Finally in these four expressions c can be replaced by true and false. This process can be illustrated with a tree, the so called *Binary Decision Tree* (BDT). The BDT for the example is depicted in Figure 2.5.

This tree has got leaves representing the Boolean constants true (also called *One-Node*) and false (also called *Zero-Node*). The inner nodes of this tree contain the variables and represent different subexpressions. Each variable has two children, one *One-Child* representing the expression after substituting the variable of this node with true, and one *Zero-Child* representing the expression after substituting the variable of this node with false. The One-Child is connected with a plain edge while the edge to the Zero-Child is dashed.

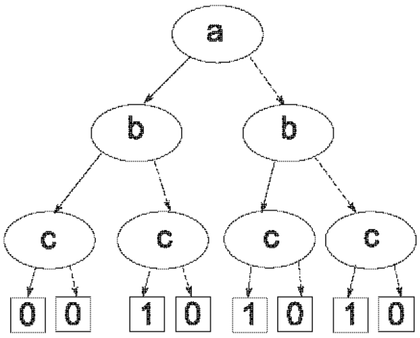

Figure 2.5: A BDT for the expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$

Obviously this is a very inefficient way to represent the Boolean expression. For an expression with n different variables, $O(2^n)$ nodes are needed. But the size of the tree can be reduced drastically with simple techniques. In this thesis, three of these techniques are explained in more detail.

2.2.1 ROBDDs

The first BDD-technique discussed here is the *Reduced Ordered Binary Decision Diagram*(ROBDD)[20][19]. It uses the Shannon Decomposition[61] to create a *Directed Acyclic Graph* (DAG) similar to the BDT. Two differences exist to the BDT, though:

1. If a newly created node v_2 is equivalent to an already existing node v_1 (i.e. their variable and their children are identical), v_2 is removed and its incoming edges are linked to v_1 instead.

2. If both outgoing edges of a node v_1 have the same target node v_2, v_1 is removed and its incoming edges are linked to v_2 instead. v_1 is also called *Don't-care-node.*

Figure 2.6 depicts these so called *reduction rules.*

These rules lead to a much slimmer diagram than the BDT. In the worst

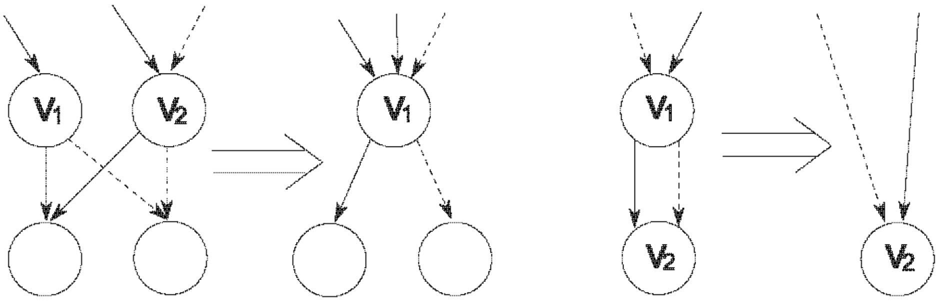

Figure 2.6: The reduction rules for an ROBDD

case, an exponential number of nodes is still needed. But for most practical cases, ROBDDs need fewer nodes. Figure 2.7 shows how the BDT for the presented example can be reduced to an ROBDD.

Note that the ordering of the variables influences the size and the structure of the ROBDD. For example, an ordering b, c, a for the expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$ would lead to an ROBDD with six nodes instead of five. A good ordering of the variables is essential for the efficiency of ROBDDs. With a good ordering, the decomposition will lead to many equivalent subexpressions enabling a lot of reductions. On the other hand, a poor ordering will lead to almost no reduction of size. Unluckily, there is no easy way to estimate the optimal ordering as this problem is co-NP-complete [20]. Only quite costly approaches exist (for example [12]). So in most applications heuristics are used for finding a good ordering.

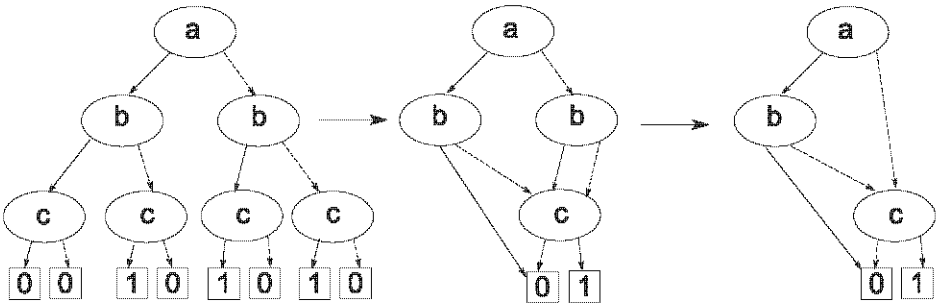

Figure 2.7: An ROBDD for the expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$

For an efficient implementation the BDT is not constructed in advance and minimised afterwards like Figure 2.7 could imply, as this would still need exponential time and memory. For an efficient implementation Bryant's apply-algorithm is used. This recursive algorithm can transform a Boolean expression $f_1 < op > f_2$ with a binary Boolean operator $< op >$ and two Boolean expressions f_1 and f_2 into an ROBDD. By splitting the original expression into two subexpressions, an ROBDD for the whole expression can be created. The basic idea is to use the Shannon decomposition for the given Boolean expression the following way:

$$f_1 < op > f_2 = \overline{x}_i \cdot (f_1|_{x_i=0} < op > f_2|_{x_i=0}) + x_i \cdot (f_1|_{x_i=1} < op > f_2|_{x_i=1})$$

The decomposition is continued recursively with the expressions $(f_1|_{x_i=0} < op > f_2|_{x_i=0})$, $(f_1|_{x_i=1} < op > f_2|_{x_i=1})$ and the next variable x_{i+1} in the variable ordering. A ROBDD-node is created for x_i, too. Its two children are the two ROBDDs of the subexpressions. Furthermore, a hash table is created which contains an entry for each created ROBDD-node u of the form (f_1, f_2, u). This table is looked up after a call of the apply-function first. If there is a hit, i.e. the two functions f_1 and f_2 were already combined, the already created node u is used instead of creating a new node. So the first reduction rule of ROBDDs is applied implicitly. This leads to a smaller diagram and the expansion of the children of n_1 respective n_2 has to be done only once instead of twice. This can reduce the complexity of the apply-algorithm dramatically if there are a lot of such node equivalences.

The recursion stops if the expression to decompose is reduced to a single Boolean variable or a Boolean constant. For these trivial cases simple sub-ROBDDs are created directly. These small sub-diagrams are merged, so that finally a diagram representing the whole original expression is created. Also the don't-care-nodes are eliminated on the fly while the algorithm is working, so that the graph created by the apply algorithm is a proper ROBDD.

This algorithm has the time- and space-complexity of $O(|G_1| \cdot |G_2|)$ where $|G_1|$ and $|G_2|$ are the number of nodes of the ROBDDs for the subexpressions f_1 and f_2. In the worst-case this can still lead to an exponential complexity for the whole ROBDD-creation process regarding the number of variables in the original Boolean expression. But for most cases ROBDDs are much more compact than BDTs.

In the literature *BDD* is often used as a synonym for *ROBDD*. In this work this is not the case. BDD is used as generic term for different types of diagrams based on BDD-techniques.

2.2.2 ZBDDs

An alternative to ROBDDs are the *zero-suppressed Binary Decision Diagrams* (ZBDDs)[42]. They are created similar like ROBDDs using the apply-algorithm, but they use different reduction rules:

1. If the one-child of a node v_1 is the Zero-Node, it is removed. Its incom-

ing edges are linked with the zero-child of v_1 instead.

2. If a newly created node v_2 is equivalent to an already existing node v_1 (i.e. their variable and their children are identical), v_2 is removed and its incoming edges are linked to v_1 instead.

Figure 2.8 depicts the reduction rules for ZBDDs.

If these reduction rules are applied to the presented example, the ZBDD shown in Figure 2.9 is created.

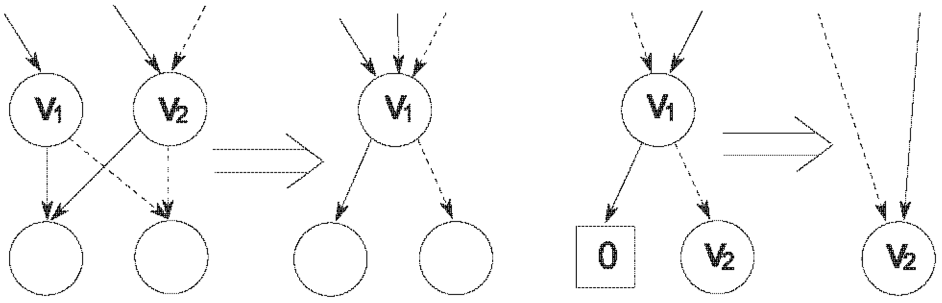

Figure 2.8: The reduction rules for a ZBDD

Whether an ROBDD or a ZBDD is more efficient for representing an expression, depends on its structure. If the corresponding BDT contains a lot of nodes with two equivalent children, an ROBDD is superior. If it contains many nodes with the zero-node as one-child, a ZBDD is better. In general, both have the same worst-case-complexity. In the worst case, exponential time and memory is needed for creating both the ROBDD and the ZBDD. But for most practical examples the complexity of both BDD-types is much better.

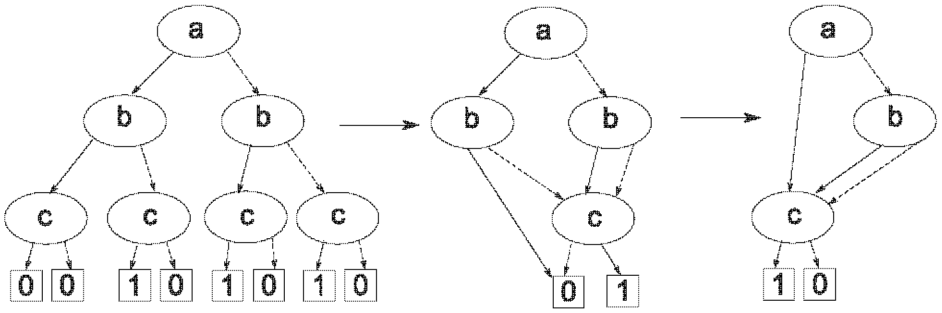

Figure 2.9: A ZBDD for the expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$

Once created, the different types of BDDs can be used for several analysis, including reliability evaluation. Many systems can be described by Boolean expressions in which the variables represent their basic components. If there is no stochastic dependency between these different variables and probabilities for each variable are given, a BDD can be used to calculate the probability of the overall expression very fast.[52] Furthermore, it is possible to create a conjunction or disjunction out of two given expressions in BDD-form very fast by using the apply-operator.

2.2.3 BEDs

The apply-algorithm presented in the previous sections can be realised in a visual way by using *Boolean Expression Diagrams* (BEDs, [11]). BEDs are extended ROBDDs which also contain binary *operator-nodes*. Inner nodes can either contain a Boolean variable x or a Boolean operator op. Operator nodes have always two children a_1 and a_2 just like variable nodes and represent the Boolean function $a_1\ op\ a_2$. It is possible to remove the operator nodes in the BED and transform it to a regular ROBDD by using two reduction rules:

By applying the rules from Figure 2.10 iteratively the operator nodes can be pushed down until they reach the terminal nodes. After that they can be removed totally by using the Boolean logic.

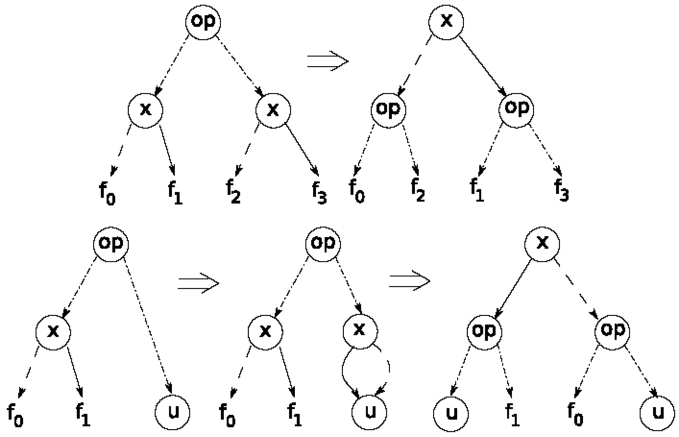

Figure 2.10: The reduction rules for BEDs

This process of pushing down the operator nodes and replacing it with variable nodes is exactly the same as in the apply algorithm, the resulting ROBDDs are equal. But one major difference exists which is used in this work: To use the apply-algorithm the basic Boolean expression has to be known in advance and it has to be in an explicit form. This algorithm starts to create the BDD from the bottom. For BEDs this explicit form is not necessary. It is sufficient to know only parts of the expression explicitly while the rest of the expression can be still encoded into another form. So it is possible to start the construction of the BDD from top. In certain circumstances [68] - like in our application - in which it is complicated or inefficient to extract the whole Boolean expression in one step this can be a major advantage.

2.3 Related Work

This section presents other projects related to this thesis in regards to the usage of BDD-techniques and safety modelling. Both the similarities and the differences to this thesis are explained. Overall, three different projects are presented: SafeMe, a modelling formalism for safety critical systems; HipHOPS, an annotation technique for systems with several failure modes; and an approach to evaluate multi-phased missions with BDDs.

2.3.1 SafeMe

In [65] the authors present a formalism to describe safety critical systems based on standard fault trees or RBDs. This approach is an extension of the modelling environment OpenSESAME already explained in Chapter 2.1.5.
As safety critical systems normally have more than one undesirable event, each of it can be described with a FT or an RBD. Furthermore, relations between these different failure modes can be defined in interrelation diagrams, e.g. that a system in the state "safety shutdown" can not create an accident. Besides that, several other dependencies can be defined using the diagrams of OpenSESAME, like fault propagation, common cause failures, shared repair or necessary time spans for fault detection.
After defining the high level model, an equivalent state based model can be created automatically. The SBM can evaluate the modelled system taking into account all the different effects in the SafeMe-model. This makes SafeMe very powerful while it remains easy to understand.
Like in IFDs which are explained afterwards, SafeMe supports several failure modes and a hierarchical modelling approach. But it is not possible to describe the whole system with one high level model, though. Instead, each global failure mode needs its a own high level model. Furthermore SafeME does not support multiple failure modes for single components directly.

2.3.2 Hip-HOPS

Hierarchically Performed Hazard Origin and Propagation Studies (Hip-HOPS) presented for example in [43] or [44] can be applied to several different high level models like RBDs, FTs or data flow diagrams. These can be annoted so that the failures of components in these high level components can be discriminated into different types like omission or timing failures. Afterwards, the model can be evaluated in two steps. First, the local failure behaviour of each component is evaluated. Afterwards, a global model is created based on the connections of the components. For this global model different fault trees can be derived automatically. These FTs can be connected to a larger graph, which is used as base for a BDD. With the BDD an overall quantitative evaluation can be done in order to estimate the safety of the modelled system.

The main target of this approach is to facilitate the step from the first very abstract and simplified models like fault trees to a more detailed model including several failure modes. Hip-HOPS only support one global failure mode, though.

2.3.3 Multi-Phased Missions

At Loughborough University, a method for evaluating multi-phased missions has been developed[48][26]. The reliability structure of systems with multiple, different phases in their mission time can change, so it is necessary to model all the different phases on their own.

In the method presented in [48], each phase is described with a fault tree. These fault trees can be transformed into BDDs. The main target is to gather a global BDD for the whole mission time.

To reach this aim, the BDDs for the different phases are combined. The zero-node in the BDD for the n-th phase (i.e. the system is still working) is replaced by the root of the BDD for the $n+1$-th phase. By this process, large and complex systems with multiple phases can be described easily. Furthermore, this methods allows a fast estimation of the probability of mission success for systems in operation, which can be used to determine if the mission should be aborted or not. The responding global BDD is created in advance, and as soon as a component fails, the failure probabilities for this component are changed in the whole BDD to one. Afterwards, the new success probability can be calculated directly.

In this work, we also create a BDD by combining several small BDDs. But unlike in [48] we can not simply change failure probabilities of the components during the calculus. Furthermore, we have more than one failure mode for one component.

2.3.4 Mode Automata

Mode automata are a quite powerful formalism for describing safety critical systems. A mode automaton is an input/output automaton with several possible discrete states, called *modes*. Furthermore, there are continuous *flow variables* for input and for output flows. Events change modes taking into account the current mode and the values of the flow variables. In each mode, a transfer function determines the values of the output flows from the values of the input flows[53].

One advantage of mode automata is that they can be assembled hierarchically, just like fault trees or reliability block diagrams. There are three possible ways to compose two mode automata: parallel, serial and by synchronisation.

Mode Automata are more powerful than Boolean methods. The problem of this is that the analysis of such automata is harder as there are much more possible states. To avoid this combinatorial explosion, it is possible to transform the automata into a Boolean form. By doing this, some information of the original automaton is lost. Still it is a good way to get a Boolean description of a complex system which includes as many of its properties as possible.

Mode automata are the base for the language AltaRica[33][13]. AltaRica allows a textual description of mode automata. One advantage of this possibility that AltaRica-models can be created automatically from other representations. Besides that, often it is easier to describe all important properties of a node in a textual way than using graphic means. An example for a complex model is [16].

AltaRica is interesting as a powerful high-level formalism can be transformed into a simpler Boolean form for evaluation. A similar process is also necessary for the IFD-formalism.

2.3.5 Fault tolerant stack processor

The information flow approach was also used for evaluating a model of a fault tolerant stack processor. [15][29][28]

In this work, a stack processor was modelled in VHDL-RTL. This VHDL-description was used as base for an information flow model of this processor. In this special case, the number of potential states was extended in comparison to the standard IFD-model:

- Working correctly
- Detected and tolerated failure
- Detected and non-tolerated failure
- Non-detected and non-tolerated failure
- Spurious shutdown
- Stop of system

For quantified results, a state automaton was extracted out of the IFD. These results were applied to several different machine programs for this stack processor. So it was possible to estimate a reliability of each program, not only a general reliability of the whole processor.

2.3.6 CASPA

Stochastic process algebras (SPA)[38] are capable of describing systems in order to evaluate their reliability and performance. A textual model is created which specifies the system to model by describing the behaviour of its processes. For example it is possible to include different kind of compositions of processes, interruptions, deadlocks and several other features.

On example of a SPA is CASPA.[40][14] CASPA is quite powerful as SPAs can be mapped to equivalent Continuous time Markov Chains (CMTS). In order to solve them, efficient algorithms are necessary, though.

In CASPA, a so called *Stochastic Labelled Transition System* (SLTS) is created, based on the textual description. The SLTS can be used to create multi terminal ZBDDs (MT-ZBDDs). MT-ZBDDs are like normal ZBDDs, their only difference is that the leaves of this BDD are not limited to contain only the Boolean constants *true* and *false*, but also Boolean variables.

By using such BDD-techniques, even complex systems can be evaluated with CASPA in a quantitative way efficiently. CASPA also uses and modifies ZBDDs just like we will do for our application. This shows that ZBDDs can be a viable and practicable alternative to ROBDDs.

Chapter 3

Information Flow Diagrams

The task of this thesis is to enhance the evaluation process of Information Flow Diagrams (IFDs) introduced by Karim Hamidi [35]. The IFD-model is well adapted to solve the problem of estimating the PFD and PFS for safety critical systems mentioned in Chapter 1.1. The system is described by its logical structure based on the information flow. It is possible to extract different scenarios leading to different system states out of this model.
Its target is to extract two lists: L_a and L_d of scenarios leading respectively to spurious trips and to dangerous failures. These lists are used to calculate the PFD and PFS. This chapter presents the original IFD-approach in detail and explains how it can handle several failure modes for both the global system and its components.
The IFD-model is a hierarchic model. It consists of a high-level model describing the system structure and a low level model for single functional entities. Furthermore, atomic components can be defined by using Markov chains. First the high-level model, a block diagram representing the information flow, is presented. Afterwards, the low-level model based on finite automata is shown, followed by the model for hardware resources. This Chapter also explains how these two different models are combined for an overall solution. Finally a short evaluation about the advantages and disadvantages will follow.

3.1 The High Level Model

The high-level model depicts the information flow through the different functional entities of the modelled system. It consists of blocks representing the functional entities and directed edges between these blocks representing the information flow. This graph is acyclic. An example, modelling a chemical reactor, is shown in Figure 3.1.

The IFD-approach distinguishes between three different types of information:

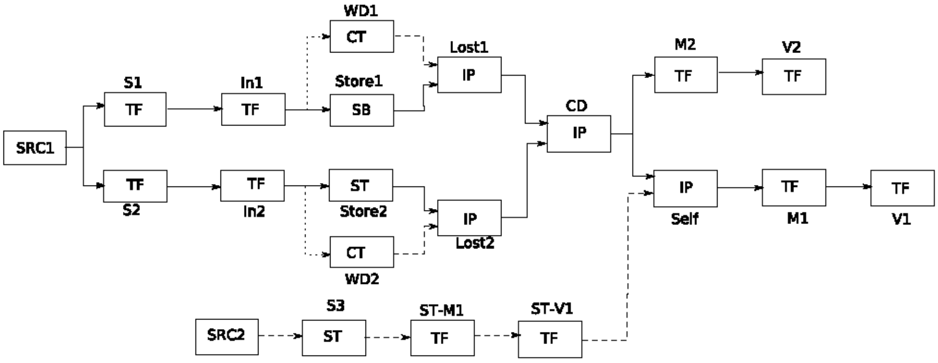

Figure 3.1: An IFD of a chemical reactor

- D-information are used for diagnostic purposes. Information about detected faults is classified in this category. It is drawn as wide dashed arrows.

- F-information represents information about the current system state and the necessity for shutting down the system. It is drawn as plain arrows.

- Control instructions for special entities in the system. Its graphical representation are pointed dashed arrows.

Figure 3.2 shows how these three types are represented in the graph. The information type and its content can be altered in the different blocks. The information can also be split between two blocks. This allows to forward the output information of one block to several other blocks in the IFD.

The blocks can represent electronic (e.g. controllers, memory), electrical (e.g. electric motors) or mechanical (e.g valves) subsystems. They are also distinguished: There are SRC1-, SRC2-, TF-, ST-, SB-, CT- and IP-blocks.

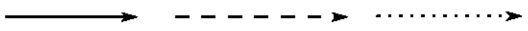

Figure 3.2: The arrows for F-information (left), D-information (middle) and control instructions (right).

SRC1-blocks: SRC1-blocks represent the sensors of the system. They have no input and one output. They deliver F-information which is considered to be faultless. This information tells if a shutdown is necessary (presence of demand) or not (absence of demand) based on the environment of the system.

SRC2-blocks: SRC2-blocks deliver faultless information about detectable existing faults in the system. They have no input and one output of diagnostic D-information. They do not represent the actual testing procedures, though. These are represented by ST-blocks which can produce faults.

TF-blocks: TF-blocks represent entities transforming signals. TF-blocks have always one input and most times one output. The input information can be either F- or D-information, the type of the output information is always the same as the type of the input information. TF-blocks transform the incoming signal (e.g. by amplification, encoding, ...), including even changes of the nature of the signal (electronic, electric, mechanical,...). A TF-block in the IFD is defined as the final block and has no output. The final block represents the actuators of the modelled system.

ST-blocks: ST-blocks represent self test entities in the system which can detect faulty components. They have one incoming and one outgoing edge. The predecessor block is always of type SRC2, i.e its input information is always of the type D. Its output is also a signal with diagnostic D-information.

SB-blocks: SB-blocks represent entities storing information for a short time. They memorise the incoming signal which can be either F- or D-information and forwards it afterwards without changing it or its type. These blocks are used for synchronizing the system.

CT-blocks: CT-blocks control the information flow. They can detect the loss of signals and are used to model functions like watchdogs. Their input are control instructions, their output are diagnostic results, i.r D-information. The control instructions are split from either F- or D-information between two blocks. Figure 3.3 shows how CT-blocks are represented in the diagram.

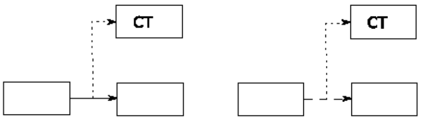

Figure 3.3: Possible use of CT-blocks.

IP-blocks: IP-blocks are used for decision entities. They have two inputs and one output signal. The type of the input and output signals have to be F- or D-information. It is also possible to have one input of F- and one input of D-information for an IP-block. The outgoing information can also be of both types.

The high level model represents the system structure based on the system's information flow. In each block the system can be in one faulty state: S, D or I. The state S is used for spurious trips, i.e. the system has detected and reacted on non existing failures. D represents dangerous failures in which failures are not detected and the system is not shut down although it would be necessary. The so called inhabitant state I is used for the loss of signals. To define in which state each block is, a low level model is defined. For all blocks but SRC1- and SRC2-blocks, an automaton exists which is used to extract up to three different lists (L_1, L_2, L_3) with several failure events, based on the actual resources of the system. These lists define under which circumstances the state of the blocks can change. The next section explains how these lists are created.

3.2 The Low Level Model

Before explaining the low level model for the IFD-formalism, we have to define finite automata. A *Deterministic Finite State Automaton*[37] is a quintuple $(\Sigma, S, s_0, \delta, F)$ where:

- Σ is the input alphabet (a finite, non-empty set of symbols)
- S is a finite, non empty set of states
- $s_0 \in S$ is the initial state

- δ is the state transition function: $\delta : S \times \Sigma \rightarrow S$
- F is the set of final or accepting states

Finite automata are often illustrated as directed graphs in which nodes represent the states and edges annoted with symbols from the alphabet Σ represent the state transitions.

Finite state automata are used to test if a given input consisting out of symbols from Σ are part of a *regular language*[63].

For the IFD-formalism, deterministic finite state automata are used as low level model. In this case, the automata are also always acyclic. One of these automata is depicted in Figure 3.4. Each automaton has one initial state and three accepting states E1, E2 and E3. E1, E2 and E3 accept a language describing all events leading to the states D (E1), S (E2) and I (E3).

The automata use the following alphabet:

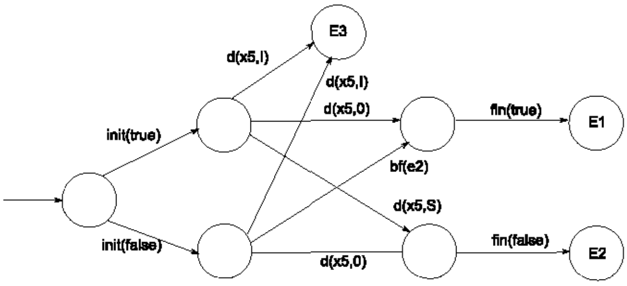

Figure 3.4: A finite automaton for a TF-block

- init(b), $b \in \{true, false\}$ (only for TF, ST, SB and CT-blocks)
- init($i_1 i_2$) $i_1, i_2 \in \{1, 2\}$ (only for IP-blocks)
- fin(b), $b \in \{true, false\}$
- $d(x_i, y)$, $i \in \mathbb{N}, y \in \{0, S, D, I\}$
- $bf(e_i)$ for a bit-flip resource e, $i \in \mathbb{N}$
- tf(k), $k \in \mathbb{N}$
- other events, noted with small letters

Initial events init(b)/init($i_1 i_2$) represent the state of the input signal. The value of b determines the state, depending on the type of block. For blocks of the type TF or SB, init(true) describes the state D, init(false) the state S. For ST-blocks, only init(true) is allowed, describing the state D. CT-blocks also allow only init(true), but for these blocks the state I is meant. As IP-blocks have two entries, they use tuples, in which i_1 describes the first incoming signal and i_2 the second. 1 represents the the state D and 2 the state S.

Final events (fin(b)) lead to the accepting states E1 and E2. For edges leading to E1, b has got always the value true, for edges leading to E2 its value is false.

The so called transient events ($d(x_i, y)$) represent hardware failures. x_i is a hardware resource, y its state. The state can be either one of the faulty ones (S,D,I) or it can be 0, i.e. the hardware resource is working properly. One special event is x_0. It is used to model the absence ($d(x_0, S)$) or the presence ($d(x_0, D)$) of demand.

Transient events are capable of modelling multiple failure modes for atomic components which is needed in complex safety critical systems. With these events, it is possible to introduce non-Boolean behaviour of components.

Environmental faults (bf(e)) like bit-flips are also included in the automatons. They are used mainly for representing bit-flips. There can be several possible occurrences of bit-flips, so several bit-flip resources e_i can be defined.

Test faults ($tf(k)$) only occur in automata for ST-blocks. The predecessors of ST-blocks are always SRC2-blocks which define a list of n components which are periodically tested. The parameter $k \leq n$ tells which of these component tests has failed. Events of the type *others* model signal transmissions in functional sub-entities. They allow to add additional ways to the nodes E1, E2 and E3. In the end, they will not influence the generated list, though. So basically the use of these events in the automaton is equivalent to the empty word ϵ.

Using these automatons, the local lists L1, L2 and L3 can be extracted by generating the languages for the nodes E1, E2 and E3. The following example illustrates this process.

Figure 3.4 depicts an automaton for a TF-block. There are two different ways possible from the initial node to the Node E1. Each way defines a *word*. These words are sequences of events which will cause the block to switch to the state D. If the different words are combined in one set, an *accepted language* is created. This language contains all possible sequences leading to the state D. Out of the example, the following three lists can be extracted using the concept explained above:

$L1 = \{Init(True)d(x5, 0)Fin(True); Init(False)d(x5, S)Fin(True)\}$

$L2 = \{Init(True)bf(e2)Fin(False); Init(False)d(x5, 0)Fin(False)\}$
$L3 = \{Init(True)d(x5, I); Init(False)d(x5, I)\}$

For SRC1- and SRC2-blocks, the lists are defined directly without using automatons. SRC1-blocks always have the following lists:

$L1 = \{d(x0, D)\}$
$L2 = \{d(x0, S)\}$
$L3 = \{\}$

For SRC2-blocks, L2 and L3 are empty. L1 contains transient events representing the resources tested in this blocks. Unlike the other blocks, the lists of SRC1- and SRC2-blocks do not give any sequences leading to a state change. Instead they define the original state of the environment.

3.3 Generation of the Global Lists

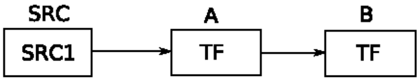

Figure 3.5: A small serial IFD

The final step is to create the two lists L_a and L_d, using the high level model and the local lists. L_d represents all scenarios leading to a dangerous failure, while L_a consists of all scenarios leading to spurious trips of the global system. These two lists can be generated automatically from the model by the so called *aggregation*. This generation process starts with the lists L1 and L2 of the final block. L1 is used for L_d, L2 is the base for L_a. The methodology to generate the lists is explained with two short examples illustrating the generation for serial IFDs and for IP-blocks. The first example, depicted in Figure 3.5 consists only out of three blocks SRC, A and B. The lists of the final block are equivalent to the lists defined by the automaton in Figure 3.4. The lists of the block A are like follows:

$L1 = \{Init(True)d(x5, 0)d(x2, S)Fin(True);$
$Init(False)d(x5, S)d(x2, 0)Fin(True)\}$

$L2 = \{Init(True)d(x2, D)aFin(False); Init(False)d(x2, S)bFin(False)\}$

$L3 = \{\}$

The events a and b are events of the type other.
L_d can be generated by replacing the initial events in L1(B). In TF-, SB- and ST-blocks, Init(true) is substituted with the sequences of L1(A), Init(false) with the sequences of L2(A) the following way:

$L_a = \{Init(True)d(x5,0)d(x2,S)Fin(True)d(x5,0)Fin(True);$
$Init(False)d(x5,S)d(x2,0)Fin(True)d(x5,0)Fin(True);$
$Init(True)d(x2,D)aFin(False)d(x5,S)Fin(True);$
$Init(False)d(x2,S)bFin(False)d(x5,S)Fin(True)\}$

$L_d = \{Init(True)d(x5,0)d(x2,S)Fin(True)bf(e2)Fin(False);$
$Init(False)d(x5,S)d(x2,0)Fin(True)bf(e2)Fin(False);$
$Init(True)d(x2,D)aFin(False)d(x5,0)Fin(False);$
$Init(False)d(x2,S)bFin(False)d(x5,0)Fin(False)\}$

Note that for CT-blocks Init(true) is replaced with the sequences of L3 of its predecessor.
The next step is to replace the new initial events with the sequences of the lists of SRC. As this block is a SRC1-block, Init(true) is replaced with $d(x0,D)$ and Init(false) with $d(x0,S)$, leading to the following result:

$L_a = \{d(x0,D)d(x5,0)d(x2,S)Fin(True)d(x5,0)Fin(True);$
$d(x0,S)d(x5,S)d(x2,0)Fin(True)d(x5,0)Fin(True);$
$d(x0,D)d(x2,D)aFin(False)d(x5,S)Fin(True);$
$d(x0,S)d(x2,S)bFin(False)d(x5,S)Fin(True)\}$

$L_d = \{d(x0,D)d(x5,0)d(x2,S)Fin(True)bf(e2)Fin(False);$
$d(x0,S)d(x5,S)d(x2,0)Fin(True)bf(e2)Fin(False);$
$d(x0,D)d(x2,D)aFin(False)d(x5,0)Fin(False);$
$d(x0,S)d(x2,S)bFin(False)d(x5,0)Fin(False)\}$

The final step is to reduce the two lists using the reduce operator RED in order to remove invalid sequences and simplify valid ones. The following simplifications are possible:

- *other*- and final-events can be removed.
- Sequences which contain one transient resource multiple times in two different states (e.g. $d(x5,S)d(x5,0)$), it can be removed as this is an impossible sequence.

- If a sequence contains a transient resource multiple times in the same state (e.g. $d(x5,0)d(x5,0)$), all but one of the transient events can be removed.

Applying these simplifications, the lists L_a and L_d can be reduced to the following form:

$$\begin{array}{l} L_a = \{d(x0,D)d(x5,0)d(x2,S); \\ d(x0,D)d(x2,D)d(x5,S); \\ d(x0,S)d(x2,S)d(x5,S)\} \end{array}$$

$$\begin{array}{l} L_d = \{d(x0,D)d(x5,0)d(x2,S)bf(e2); \\ d(x0,S)d(x5,S)d(x2,0)bf(e2); \\ d(x0,D)d(x2,D)d(x5,0); \\ d(x0,S)d(x2,S)d(x5,0)\} \end{array}$$

For IFDs with IP-blocks, the generation of the lists is similar. The difference is that for IP-blocks there are four possible values for the initial event instead of two. The following example illustrates the generation of the lists for IP-blocks. It consists of three blocks: A, B and C, depicted in Figure 3.6. We assume that the lists L1 and L2 for the blocks A and B are already aggregated. The aim is to estimate the aggregated lists for the IP-block.

The lists L1 and L2 of the blocks A, B and C are defined like that:

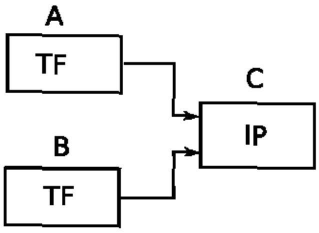

Figure 3.6: A sub IFD containing an IP-block

$$\begin{array}{l} L1(A) = \{d(xo,S)d(x5,0); d(xo,D)d(x5,S)d(x1,D)\} \\ L2(A) = \{d(xo,D)d(x5,0)d(x1,S); d(xo,S)d(x5,D)\} \end{array}$$

$$\begin{array}{l} L1(B) = \{d(xo,S)d(x5,0)d(x3,D); d(xo,D)d(x5,S)\} \\ L2(B) = \{d(xo,D)d(x5,0); d(xo,S)d(x5,D)d(x3,S)\} \end{array}$$

$$\begin{array}{l} L1(C) = \{Init(11)d(x4,0)aFin(True); Init(12)d(x4,0)aFin(True); \\ Init(21)d(x4,0)aFin(True); Init(22)d(x4,S)Fin(True)\} \end{array}$$

$L2(C) = \{Init(11)d(x4, D)Fin(False); Init(12)d(x4, D)Fin(False);$
$Init(21)d(x4, D)Fin(False); Init(22)bd(x4, 0)Fin(False)\}$

To substitute the initial events $Init(ij)$ is replaced by the list L_{ij}. It holds:

$L_{ij} = RED(Li(A) \otimes Lj(B))$

where $\otimes$ is the operator for the set product. This leads to the following four lists:

$L_{11} = \{d(xo, S)d(x5, 0)d(x3, D); d(xo, D)d(x5, S)d(x1, D)\}$
$L_{12} = \{\}$
$L_{21} = \{\}$
$L_{22} = \{d(xo, D)d(x5, 0)d(x1, S); d(xo, S)d(x5, D)d(x3, S)\}$

Now these lists can be used to replace the initial events in $L1(C)$ and $L2(C)$, and the RED-operator can be applied on these lists. This leads to the following result:

$RED(L1(C)) = \{d(xo, S)d(x5, 0)d(x3, D)d(x4, 0);$
$d(xo, D)d(x5, S)d(x1, D)d(x4, 0);$
$d(xo, D)d(x5, 0)d(x1, S)d(x4, S);$
$d(xo, S)d(x5, D)d(x3, S)d(x4, S)\}$

$RED(L2(C)) = \{d(xo, S)d(x5, 0)d(x3, D)d(x4, D);$
$d(xo, D)d(x5, S)d(x1, D)d(x4, D);$
$d(xo, D)d(x5, 0)d(x1, S)d(x4, 0);$
$d(xo, S)d(x5, D)d(x3, S)d(x4, 0)\}$

After using these aggregations, the lists L_a and L_d can be generated. These can be used for quantified analysis if the failure probabilities of the single components are known. The next section will explain how these can be gained.

3.4 Model of the Hardware Resources

In classical models the components of a system are simply modelled with Boolean variables. This is possible as in these models the components normally have just two states, failed or working. In the IFD-approach, there

are four different states for hardware resources: S, D, I and 0. To represent these states, Markov Chains are used.
Markov chains are very versatile and can be used for including a lot of different aspects into the model. Depending on effects like repairability, maintenance or failure detection, the resulting MC can look different. The easiest example is a MC for a non-repairable component depicted in Figure 3.7:

This MC consists of four states 0, S, D and I and three transitions with rates λ_S, λ_D and λ_L.

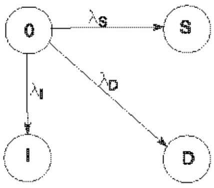

Figure 3.7: A Markov chain for a non-repairable component.

Of course more complicated component behaviour can be expressed by using MCs. It is even possible to model the MCs themselves in a hierarchical way, in which several states are grouped in four macro states. An example is a component which can fail either detected or undetected. If it fails undetected, a regular test routine can find the failure. Once a failure is known, the component is switched into a safe state. Afterwards, it is repaired. A Markov model for such a component is shown in Figure 3.8.

In the figure, the states DU and DD are in the macro state D, SU, SD and R are in the macro state S, IU and ID are part of the macro state I and finally the macro state 0 consists only of the state 0. The probability

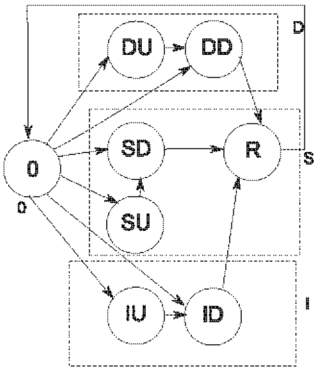

Figure 3.8: A Markov chain for a component with failure detection and repair.

that this component is e.g. in state S can be easily calculated by adding the probabilities that its MC is in the states SU, SD or R. So MCs can be used in a wide variety for modelling the components.
It is easy to change the model for the hardware resources. For a quantified solution of the global model, only the probabilities for the three failure modes are necessary. How these probabilities are estimated is not important. So it would be theoretically possible to use a total different approach to model the components without any changes to the other two layers of the IFD-model.

3.5 Evaluation of the IFD-model

The presented model has two main advantages. At first, it uses a hierarchical approach. A system can be divided into multiple functional entities, consisting of different components. Afterwards these entities and the components can be modelled on their own. This allows the user to model even large systems in a structured and easy way.
Besides that, it allows for multiple failure modes for resources and the global system. In conventional high-level modelling approaches like fault trees or reliability block diagrams, only two failure modes are supported. State based methods like Petri nets and Markov chains support such a feature directly, but they have other severe disadvantages. Chapter 2.1 gives more information about other modelling methods.
The IFD-approach has also some weaknesses, though. Some of the formalisms are a little bit confusing. For example, the three faulty states are called S,D and I, the accepting states in the automata are called E1, E2 and E3, the values of initial events are either Boolean constants or a pair of numbers and the global lists are called L_a and L_d. More structured naming conventions would make it much easier to understand this model.
The IFD-approach has a lot of different blocks and events. At one hand, it makes the model very powerful, at the other hand, it makes it less clear. The model should be checked if it is possible too remove some of the blocks or events without weakening its expression power.
The most serious problem is the combinatorial explosion while creating the lists, though. The examples showed that already one extra block could lead to a doubling of the list's size. For large systems, such a complexity is not acceptable. Luckily, there is much room for improvement. The sequences in the lists often contain the same subsequences. With BDD-techniques such equivalences can be used to create a much smaller formal representation of the model.
The main target of this thesis is to deal with the disadvantages of the IFD-

approach. Chapter 4 explains how the model was changed to improve it. Chapter 5 shows how the combinatorial explosion was avoided.

Part II

Hierarchical Modelling and Evaluation

Chapter 4

Formalisation of the IFD-Model

To be able to increase the efficiency of the information flow model's evaluation process, we have to formalise and simplify it. In its original form, it is extremely complex with a lot of different types of blocks, information flows and atomic events. In order to use BDD-techniques, this model has to be transformed into a Boolean model. This transformation process should keep the most significant features of the IFD-model presented in the previous chapter, though. The most important ones are:

- Multiple failure modes for components
- Two failure modes for the global system
- Hierarchical structure

So we need to be cautious which parts of the model can be simplified, and which parts have to remain in order to keep the characteristics of the IFD-model.

The modifications can be applied at three different parts of the original IFD-model: The information flow itself, the blocks and the low level model. But first, two simple renamings are made to increase intiutivity: Inhabitant failures (failure mode I) are called *Loss of information failures* (failure mode L) as this name is much clearer. Furthermore, CT-blocks which represent watch dog functions are renamed to *Watch Dog-blocks* (WD-blocks).

Originally, there are three types of information flow: F-information, D-information and control instructions. For each kind of block only one or two types of information are allowed as in- and output. Besides this, the kind of information is of no importance, no expressive power is gained by distinguishing these different types. These types may be interesting for understanding a modelled system, but for this task it is sufficient to add remarks

to the high level model. In the low level model, it is even impossible to take the kind of information into account for creating the local lists. So, from a functional point of view, the three kind of information flows can be seen as equivalent. From now on, only a general information flow is used in the IFD-model, we will not discriminate between the different kinds of information anymore.
The possibilities to simplify and formalise the model regarding the blocks and the low level model are explained in the following sections.

4.1 Formalisation and Simplification of the Low Level Model

In chapter 3, the low level model is based on finite automata. For the purpose of this work, it is not really important how the low level model is exactly designed. It is only important to know the local lists generated from these automata, not how these are obtained. So this section will concentrate on the lists itself.
Originally, there are three lists for each block ($L1$, $L2$ and $L3$) and two lists for the whole IFD (L_a and L_d). Each list describes several event sequences which can lead to one of the three failure modes (S, D or L). In an alternate point of view, they can be seen as a pseudo-Boolean expression in disjunctive normal form (DNF)[5]. For example the list

$$\{init(true)d(a,0)d(b,S)final(true); init(false)d(a,S)(final(true)\}$$

can also be interpreted as:

$$(init(true) \wedge (a = 0) \wedge (b = S) \wedge final(true)) \vee (init(false) \wedge (a = S) \vee final(true)).$$

From an computer scientist's point of view, expressions are more practical. Algorithms can be described much more formally and standard Boolean operations can be used. Besides, the whole lists should be transformed into a BDD. As BDDs describe expressions themselves, it is more practical and elegant to use expressions instead of lists. So from now on these lists will be transformed into equivalent expressions.
The lists $L1$, $L2$ and $L3$ of the block B are transformed to the *local expressions* $S(B)$, $D(B)$ and $L(B)$. The lists L_a and L_d are transformed to the *global expressions* S and D. This naming shows which state is described with

these expressions immediately.

In order to transform the lists into expressions, the events have to be adapted. Here only changes made to non-IP-blocks are explained, as IP-blocks will be modified much more significantly than other kind of blocks. These modifications are explained in the next subsection.

The initial events describe the state of the input signal. In order to increase intuitivity and readability of the model, the syntax is changed. For ST-, TF- and SB-blocks $init(true)$ and $init(false)$ are replaced with $input(S)$ and $input(D)$, for WD-blocks $init(true)$ is replaced with $input(L)$. So it is much clearer in which state the system has to be for the following event sequence to be valid.

The events $final(true)$ and $final(false)$ were used to indicate that the automaton will be in the final state $E1$ respective $E2$, i.e. that the according block will be in the state S respective D. In fact, these are not needed as the resulting state is already defined by the type of expression (S, D or L). So the $final$-events are simply removed.

Events of the type "other" should represent exchanged signals between HW-resources. But while aggregating the global lists, the RED-operator (chapter 3.3) will remove them. In the end, they will not influence the outcome of the list. So they can simply be removed totally.

There are three types of failures in the original model: The so called transient errors, bit flips and fault tests. While transient errors have got three different failure states, bit flips have only one. Fault tests refer to other failure events with one failure mode. To reduce the complexity of the model, it is possible to transform bit flip and fault test failures into transient ones by defining a component c, for which only one of the three failure states can occur, i.e. the probability of the other two failure modes is zero. Besides, the syntax is changed: A component c, representing either hardware or software, in state $s \in \{S, D, L, 0\}$ is noted as $(c = s)$. Transient failures will just be called *failures*, as they represent any possible hardware or software failure, and as the name "transient failure" is a little bit misleading anyway.

Now a short example is given to show how the lists are transformed into expressions. The following lists for a block B are given:

$$L1(B) = \{init(true)d(a,0)d(b,S)final(true); init(false)d(a,S)(final(true)\}$$
$$L2(B) = \{init(true)d(a,D)final(false); init(false)d(b,S)bf(e)(final(false)\}$$
$$L3(B) = \{\}$$

The equivalent expressions looks like this:

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

$D(B) = (input(S) \wedge (a = D)) \vee (input(D) \wedge (b = S) \wedge (e = S))$
$L(B) = false$

The empty set is simply transformed to $false$. For the bit flip failure, a component is created for which only one failure mode, in this case S, is possible. Test faults can be modelled similarly by creating a functional component representing the test which can fail in one mode. The transient failures and the initiating events are written in the new syntax. So the expressions are absolutely equivalent to the respective lists. It just uses another formalism which is more practical for the purpose of this work.
Furthermore, one restriction is removed: In the original model, each sequence in the lists for non-SRC-blocks had to start with an initial event. This is not necessary anymore, as there are many systems in which a failure of one component will always lead to the same system state, no matter what input information the component gets. So the models can be created with shorter expressions.
The assumption that the expressions have to be in DNF is not changed, it will make it easier to handle them this way.

4.2 Formalisation and Simplification of Blocks

In chapter 3, seven kind of blocks were defined: $SRC1$, $SRC2$, ST, TF, SB, WD and IP. The next step is to examine these for possible simplifications. Blocks of the type $SRC1$ and $SRC2$ both create information. The only difference is that blocks of the type $SRC1$ create F-information while blocks of the type $SRC2$ create D-information. As we do not discriminate between different kind of information anymore, only one type of source block is needed, which is called SRC. Furthermore, the information created of the blocks of the type $SRC1$ and $SRC2$ just defines the state of the environment of the system, i.e if the system should be stopped or not. This allows for a direct calculation of the PFD and PFS. For evaluating the safety of a control system, it is better to concentrate just on the reliability of the safety functions and the probabilities of the two global failure modes, though. The failure of these functions in absence of demand is still a failure of the control system which needs to be repaired, although no accident has happened. It is still possible to calculate the PFD and PFS, if both the probabilities of the two different failure modes S and D and the frequency of dangerous situations is known. So the expressions of SRC-blocks can be arbitrarily complex representing actual sensors in the modelled system, their only limit is that they can not contain any *input*-events.

Blocks of the type ST, TF, SB and WD have all one input and one output. There is an important difference between blocks of the type WD and the other three types, though. WD-blocks can detect only loss of information (failure mode L), the other three types can detect only the failure modes S and D.
The three different types ST, TF and SB are functionally equivalent, though. They all have one input signal and one output signal, and all of them can only handle signals in the state S and D. So these three types can be merged to a standard block, abbreviated as ST-block.
IP-blocks are used to merge two different paths, so they are quite unique. Obviously, such blocks need to remain in the model if it should not lose to much usability. IP-blocks have some serious drawbacks, though:

- They can only merge two paths. If three or more paths need to be merged, for example in a 2-out-of-3-system, several IP-blocks have to be combined. The number of IP-blocks needed grows exponentially with the number of paths to merge.
- The creation of the automatons for the IP-blocks is quite cumbersome. Four different initiating events are needed instead of two like in the other blocks. It is necessary to have an order of the two predecessor blocks to initialise the values for these events correctly, as $Init(12)$ is not equal to $Init(21)$. But it is quite impossible to see which predecessor block initiates the first value and which block initiates the second one.

This is the reason that IP-blocks were strongly overhauled. Three major changes were made:

- They can have arbitrarily many predecessor blocks.
- Their low level model is changed to expressions consisting of the predecessor blocks and their possible states.
- Component failures can not be included in these blocks anymore.

Furthermore, they are renamed to *decision nodes* (DEC-nodes). In the graphical representation, they are shown as circles. The reason for this renaming and the different graphical representation is that we want to highlight the fact that decision nodes have very different characteristics compared to SRC-, ST- and WD-blocks. The low-level model of DEC-nodes is defined with three expressions per node, one for each failure mode.
Now an example will visualise show such expressions. The expressions of the

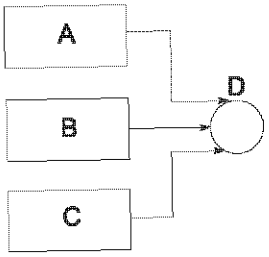

Figure 4.1: A decision node D with three predecessor blocks A, B and C

DEC-node D in Figure 4.1 can look like this:

$S : ((A = S) \wedge (B = S)) \vee ((A = D) \wedge (C = S))$
$D : (A = D) \wedge (B = D) \wedge (C = D)$
$L : ((B = D) \wedge (C = L)) \vee ((A = D) \wedge (B = S) \wedge (C = L))$

In the original IP-blocks, it was possible to integrate component failures into the block's automaton. Now this is impossible as the decision nodes do not represent an actual component or subsystem, so they can not fail and take the "wrong" decision. If there is an actual subsystem or component taking a decision based on several inputs, like a voter, it has to be added as successor block after the decision node. For example, a 1-out-of-2-voter monitoring a larger system can be modelled with the IFD in Figure 4.2: Two sensors deliver their measurements to the voter, demanding either a system shutdown or to continue the operation. If at least one of the sensors demands a shutdown, the voter will stop the monitored system. Both the sensors and the voter can fail in two different ways. The sensor can demand an unnecessary shutdown (failure mode S) or fail to demand a necessary one (failure mode D). The voter, represented by the component *vot*, can either start a spurious shutdown (failure mode S) or not stop the system if it has to (failure mode D). This set-up of the voting system leads to the depicted IFD and the following expressions for the decision node Dec:

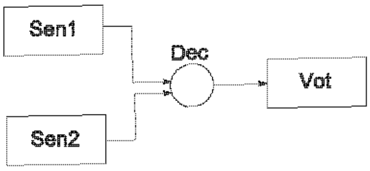

Figure 4.2: The IFD for an 1-out-of-2-voter

$S : (Sen1 = S) \vee (Sen2 = S)$
$D : (Sen1 = D) \wedge (Sen2 = D)$
$L : false$ — This failure mode is impossible

This means that the voter should stop the system unnecessarily if one of the sensors is demanding such an unwanted shutdown. The voter should not stop the system in a dangerous situation if no sensor demands it. But as the voter *vot* itself can fail, it is included in the block Vot with the following expressions:

$S : (input(S) \wedge (vot = 0)) \vee (vot = S)$
$D : (input(D) \wedge (vot = 0)) \vee (vot = D)$

So, the voting system will fail dangerously either if both sensors fail dangerously and the voter works correctly or if the actual voter does not start a shutdown in presence of demand. The system will fail spuriously if either one of the sensors fail spuriously and the voter works correctly or if the voter shuts down the monitored system unnecessarily.

In the original model, only TF-blocks could be final blocks. This limitation is removed, theoretically any type of block can be chosen to be the final block. In real examples, the final block should be normally a ST-block or a DEC-node. Allowing DEC-nodes to be the final node in the IFD has the advantage, that systems with multiple redundant final actuators like motors or valves can be modelled easier by adding a final DEC-node which is linked to the blocks representing the the actuators.

Graphically, the different kind of blocks and nodes are distinguished like that:

- ST- and SRC-blocks are drawn as boxes with a plain border with their name written inside the box. SRC- and ST-blocks can be simply distinguished by the fact that ST-blocks have incoming edges while SRC-blocks only have outgoing edges.
- WD-blocks are drawn as boxes with a dashed border with their name written inside the box.
- DEC-nodes are drawn as circles with their name written outside the circle.
- The final block or node is highlighted by a double border for the respective box or circle.

All edges are drawn as plain one-directional arrows. A whole IFD with this layout is shown in Figure 4.3.

Of course even a more drastic simplification and formalisation would be possible. WD- and ST-blocks could be unified, leading to blocks with expressions containing all three types of input-expressions. It would be even possible to allow arbitrarily many failure modes and stop limiting these to three. But for performance reasons, these changes were not made. This is explained in the next chapter.

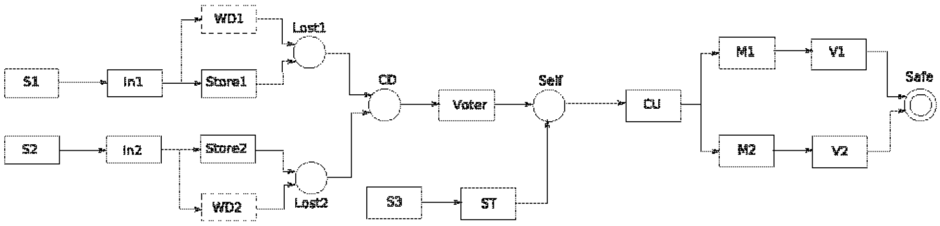
WD1
Lost1
S1
In1
Store1
CD
Voter
Self
CU
M1
V1
Safe
S2
In2
Store2
Lost2
WD2
S3
ST
M2
V2

Figure 4.3: A complete IFD of a chemical reactor. (Also see Section 7.2)

4.3 Overview of the Modifications

Old model	**New Model**	**Explanation**
lists	expressions	
D-Information F-information control instructions	Information	
SB-blocks TF-blocks ST-blocks	ST-blocks	Standard-blocks
SRC1-blocks SRC2-blocks	SRC-blocks	Source of information
CT-blocks	WD-blocks	Watchdog-Blocks
IP-blocks	DEC-nodes	Decision nodes
init(true) init(false)	input(S) input(D) input(L)	Initial values in the low level model
Transient failures bitflips test faults	failures	
d(c,D)	(c = D)	Component c in failure mode D
bf(e)	(e = S)	Bitflip in component e
ft(i)	$(f_i = \mathrm{S})$	Failure of test-component f_i
final(true) final(false)	removed	Leading always to the same state ⇒ redundant
"other"-events	removed	No influence on result
L1	S	Safe failure mode for single blocks
L2	D	Dangerous failure mode for single blocks
L3	L	Loss of information mode for single blocks
L_a	S	Spurious trip for whole system
L_d	D	Dangerous failure for whole system

Comparison between the old and the changed model

Both the low and the high level model were reduced quite a lot, while the main characteristics of the IFD-model were preserved. This subsection will present a table with a short overview of the changes which were made. In the left column, formalisms and vocabulary of the old model can be found. In the middle column, the respective expressions for the changed model are presented. The right column shows explanations.

Obviously, a lot of elements have been removed or were merged in the process of simplification and formalisation. This means that it is much easier to develop efficient algorithms for evaluating such a model. Still, it has kept its most important and unique characteristics. It is hierarchical and it supports several failure modes for single components in the system and for the global system itself.

Note that the following convention is used in this thesis: The names of basic components are written in small letters and the names of the blocks are written in capital letters.

In Chapter 7.2 a case study is presented in which the same system is modelled in both the old and the new way. That example will highlight the differences between the original and formalised model further.

4.4 Overview of the New Formalism

This section gives an overview about the modified formalism without referring to the original model. It is defined and explained both formally and using brief examples.

The modified formalism is able to describe a system's architecture and behaviour by using the internal information flow. In this context, the information flow can be electronic (e.g. data sent over a bus), electric (e.g. starting an electric motor), mechanic (e.g. an activated lever), or even physical (e.g. a chemical which is heated and changes the state of the system). An example is shown in Figure 4.4.

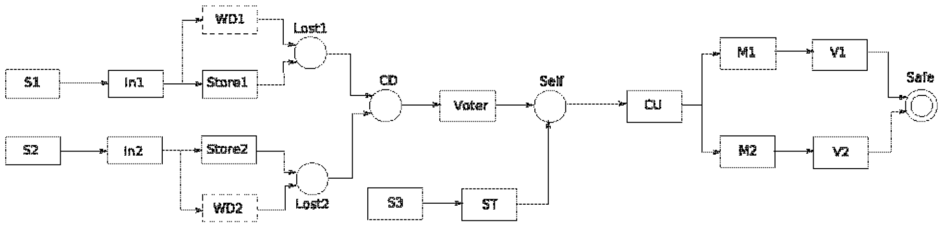

Figure 4.4: A complete IFD of a chemical reactor. (Also see Section 7.2)

4.4.1 The high level formalism

An IFD is a directed, acyclic graph. Its nodes represent either different functional entities or rules about the behaviour of the global system. Each node is always in one and only one of the following states:

- 0 (working correctly)

- S (spurious trip resp. safe failure mode)
- D (absence on demand resp. dangerous failure mode)
- L (Loss of information)

Each node passes its state to its successors. The state of a node can change due to failures and events. This is explained further in chapter 4.4.2. Besides that, every node can be identified with a name. By convention these names always start with an upper case letter.
Overall, there are four different kind of nodes:

- *Source blocks* (SRC-blocks) represent the sensors of the system. They have no incoming edge. They are drawn as boxes with a plain border and their name written in this box.
- *Standard blocks* (ST-blocks) represent all kind of functional entities in the IFD. They have always exactly one input. They are drawn as boxes with a plain border and their name written in this box.
- *Watchdog blocks* (WD-blocks) represent watchdog entities in the system. They have always exactly one input. They are drawn as boxes with a dashed border and their name written in this box.
- *Decision nodes* (DEC-nodes) represent the logical behaviour of the system. They can have multiple inputs. They are drawn as circles with their name written outside the circle.

Furthermore, one of the nodes in the IFD is marked as final node without an outgoing edge. It is represented graphically by drawing it with a double border. There are no restrictions about the type of this final node, although normally it should be either a standard block or a decision node.
The state of the final node is equivalent to the state of the global system. By calculating the probabilities for the states S and D of the final node it is possible to estimate the PFD and PFS of the system.

4.4.2 The low level formalism

For SRC-, ST- and WD-blocks special Pseudo-Boolean expressions are used to model state changes in these blocks. For each block B, three expressions are defined: S(B), D(B), and L(B), describing which events will lead to the failure modes S, D and L. These expressions have to be in DNF. Formally, they can be defined with the following Backus-Naur-Form(BNF)[36][51]:

```
<expr> ::= (<conj-expr>) [∧ (<conj-expr>)]+
<conj-expr> ::= (<conj-expr>) [∨ (<conj-expr>)]+| <input-expr>
| <failure-expr> | <bool-const>
<input-expr> ::= input(<fail-state>)
<failure-expr> ::= <comp-name> = <state>
<state> ::= S | D | L | 0
<fail-state> ::= S | D | L
<bool-const> ::= true | false
```

`<comp-name>` represents the names of the components. This means, that there a basically two different types of sub-expressions: failure expressions and input expressions.

Failure expressions represent basic components of the system and their current state (0, S, D, L). By convention components are always written in lower case. A component c in the state D is noted as follows: $(c = D)$. Input expression describe the state of the predecessor block resp. node. $input(S)$ means that the predecessor block has to be in the state S so that this expression can be satisfied. An example for such expressions is given here:

$$S(B) = (input(S) \wedge (a = 0) \wedge (b = S)) \vee (input(D) \wedge (a = S))$$
$$D(B) = (input(S) \wedge (a = D)) \vee (input(D) \wedge (b = S) \wedge (e = S))$$
$$L(B) = false$$

This means that the block B is in the state S if either the predecessor block is in the state S, the component a is in the state 0, and the component b is in the state S, or if the predecessor block is in the state D and a is in the state S. The block B can be never in the state L as this expression is simply the Boolean constant false.

Note that there are certain restrictions to input expressions. SRC-blocks can obviously do not have any input expressions as they lack predecessor nodes. For performance reasons, input expressions of ST-blocks can only contain the states S and D, while WD-blocks can can only contain input expressions in the state L.

For Decision nodes, another low level formalism was chosen. Unlike the blocks DEC-nodes do not include any hardware or software resources. They only describe in which state the system currently is given the states of the predecessor nodes. For this, we also use expressions, although with another syntax:

```
<dec-expr> ::= (<dec-expr>) [<bool-op> (<dec-expr>)]⁺
| <block-name> = <state> | <bool-const>
<bool-op> ::= ∧ | ∨
<state> ::= S | D | L
<bool-const> ::= true | false
```

`<block-name>` represents the names of the predecessor blocks.
As example, a system with a one out of two voter is selected, shown in Figure 4.5:
The expressions for the node *Dec* are as follows:

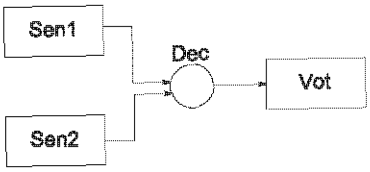

Figure 4.5: The IFD for an 1-out-of-2-voter

$S : (Sen1 = S) \vee (Sen2 = S)$
$D : (Sen1 = D) \wedge (Sen2 = D)$
$L : false$ — This failure mode is impossible

The sub-expressions like $(Sen1 = S)$ are also called *Decision expressions.* This means, that there will be a spurious trip if at least one of the two Sensors produce a spurious trip. A dangerous failure will occur if both sensors fail dangerously. A failure of type L is not possible in this system.
Note that failures of the voting process itself can not be handled in the DEC-node *Dec*, but they can be modelled by adding them to the block Vot.

4.4.3 Combining high and low level

With the new formalism, we want to calculate PFD(t) and PFS(t) for a given time t. To do so, it is necessary to extract the expressions D and S for the global system. For this purpose, the expressions D and S of the final node are used. Now Input expressions (in case the final node is of the type ST or WD) resp. Decisions expressions (in case the final node is of the type DEC) have to be substituted. For ST- and WD-blocks, the respective expressions S, D or L of the predecessor block are used to replace the input expressions.

This is done similar with decision expressions. This substitution process is continued recursively, until the source blocks are reached.
To illustrate this, a very simple model is chosen, the 1-out-of-2-voter. The expressions for the DEC-node is already given, the expressions for the other blocks look like this:

$S(Vot) : (input(S) \wedge (vot = 0)) \vee (vot = S)$
$D(Vot) : (input(D) \wedge (vot = 0)) \vee (vot = D)$
$S(Sen1) : (s1 = S)$
$D(Sen1) : (s1 = D)$
$S(Sen2) : (s2 = S)$
$D(Sen2) : (s2 = D)$

The expressions L are $false$ for all blocks. Now, the input expression in $S(Vot)$ can be substituted, leading to the following expression:

$S(Vot) : (((Sen1 = S) \vee (Sen2 = S)) \wedge (vot = 0)) \vee (vot = S)$

With a further substitution, we receive the following result:

$S(Vot) : (((s1 = S) \vee (s2 = S)) \wedge (vot = 0)) \vee (vot = S)$

In this expression, there is no substitution possible, as all sub-expressions only contain basic components. We have the expression S for the global system. If the failure probabilities of the basic components are known, we can calculate the PFS(t) based on this expression.

4.4.4 Modelling basic components

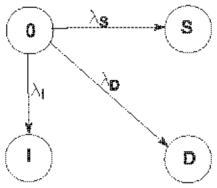

Figure 4.6: A Markov Chain for a non-repairable component.

In order to calculate PFD(t) and PFS(t) it is necessary to know the failure probabilities of the components at the time t. In order to estimate these values we used a simple Markov model for the components. We assume that no repairs will happen, and that a component will not switch from one faulty state to another faulty state. Furthermore, three failure rates for each component are defined, λ_S, λ_D, and λ_L. This leads to the Markov Chain depicted in Figure 4.6.

Note that it would not be a computational problem to use more complex Markov Chains for including effects like repairs or switching between different failure states. The calculation algorithms for analysing MCs could easily handle larger models.

Chapter 5

Developed Algorithms

The present chapter explains the algorithms used to transform the IFD-model into a compact Boolean form. Today, BDD-techniques are the state of the art for efficient treatment of large Boolean expressions. In our case, two different BDD-techniques were combined: ZBDDs and BEDs. The developed algorithm has already been presented in several articles (for example [46]). Now it will be explained in more detail.

The chapter is divided in several sections. First, the transformation of the expressions from Chapter 4 to Boolean expressions is explained. Afterwards the creation of BDDs is shown in several steps. At first a method is presented to transform a IFD without DEC-nodes into a BDD. For this case we assume that each component occurs only in the expressions of one block. Afterwards, the algorithm is extended to serial IFDs without the last assumption. The next section will deal with DEC-nodes and how these can be transformed into a BDD. Finally all these results will be combined for an algorithm powerful enough to handle all possible IFDs. The creation process will be demonstrated using a small example which contains all features of IFDs.

After explaining the creation of the BDDs, it is necessary to evaluate them for getting quantitative results. The last section will deal with this problem.

5.1 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. Basically, the local expressions contain two types of subexpressions: input expressions like *Input(S)* and failure expressions like $(a = S)$. For now, input expressions are ignored; they remain in the expression and will be handled later. For failures, we need a Boolean variable based on the component and its state. This would be easy if each component had only two states. Components have four states, though. So three different boolean variables c_0, c_S, and c_D are defined for every component c. These variables are true if c is in the state 0, S or D. Note that three variables are enough, a variable c_L is not necessary. As c can only be in one state at one time, the value of c_L can be deduced from the values of the other three variables. It holds: $c_{S1} \wedge c_{S2} = false$ for $S1 \neq S2$ for $S1, S2 \in \{0, S, D\}$.

Of course it would be possible to express four different states just with two Boolean variables. There are four possible permutations for two boolean variables, so every permutation could represent one state. But the main aim of reducing the number of variables is to effectively reduce the number of nodes in the final BDD. Surprisingly, it does not make any difference regarding the number of BDD-nodes if three or just two variables are used. Besides that the solution with three variables is less error prone to implement and has a major advantage for the quantified evaluation. We will give a detailed explanation later in this chapter.

The transformation is demonstrated with an example. Given are the following expressions of a block B:

$$S(B) = (input(S) \wedge (a = 0) \wedge (b = S)) \vee (input(D) \wedge (a = S))$$
$$D(B) = (input(S) \wedge (a = D)) \vee (input(D) \wedge (b = L) \wedge (e = S))$$
$$L(B) = false$$

The subexpressions representing the failures can now be transformed to Boolean variables. The input expressions remain unaltered, the next section will explain how these are handled. This leads to the following expressions:

$$S(B) = (input(S) \wedge a_0 \wedge \overline{a_S} \wedge \overline{a_D} \wedge \overline{b_0} \wedge b_S \wedge \overline{b_D}) \vee (input(D) \wedge \overline{a_0} \wedge a_S \wedge \overline{a_D})$$
$$D(B) = (input(S) \wedge \overline{a_0} \wedge \overline{a_S} \wedge a_D) \vee (input(D) \wedge \overline{b_0} \wedge \overline{b_S} \wedge \overline{b_D} \wedge \overline{e_0} \wedge e_S \wedge \overline{e_D})$$
$$L(B) = false$$

As each Boolean variable represents a state of a component, a probability for this variable can be defined. With these probabilities it is possible to evaluate the BDD quantitatively if the variables are stochastic independent. For this application this is not true, though. Luckily, section 5.6 will show that this causes no real problems as there is a workaround.

5.2 ZBDDs for Simple Serial Systems

To describe the algorithm for creating a ZBDD equivalent to the IFD, some restrictions regarding to the IFD are made. These restrictions are loosened in the next sections. At first, two assumptions are made:

- The IFD does not contain any DEC-blocks.
- Each Boolean variables only occurs in the expression of one block.

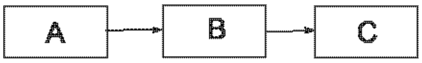

Figure 5.1: The high level model for a simple serial IFD

With these assumptions, it is very easy to use the structuring of the IFD in order to create a ZBDD very efficiently. First two ZBDDs for the final block based on the local expressions S and D are created in parallel. These two ZBDDs share equivalent nodes if possible.

The ZBDDs are created by decomposing the expressions S and D. As these expressions can contain input-subexpressions it is possible that the created ZBDD contains extra leaves besides the trivial one- and zero-node. For ST-blocks, three non-trivial leaves can appear: $Input(S)$, $Input(D)$ and $Input(S) \vee Input(D)$. For WD-blocks, only one extra leaf ($Input(L)$) is possible.

Note that the number of non-trivial leaves depends on the number of input-states which are allowed in one block. If in a type of block Input-expressions can take n different values, $2^n - 1$ non-trivial leaves are possible. This is the reason that WD- and ST-blocks are still distinguished and not unified to a type of block allowing all three failure modes for Input-expressions. In this case, up to seven non-trivial leaves would have been possible, leading to broader and probably larger BDDs.

We have chosen ZBDDs and not ROBDDs as they are better suited for our application. For a component c, three boolean variables c_0, c_S and c_D are created. There are eight different possibilities to set these variables true or false. Four of them ($c_0 \wedge c_S \wedge c_D, c_0 \wedge c_S \wedge \overline{c_0}, c_0 \wedge \overline{c_S} \wedge c_D, \overline{c_0} \wedge c_S \wedge c_D$) are always invalid, though, as only at most one of the variables can be true. So these four combinations will always lead to the zero-node which can be reduced in ZBDDs. Figure 5.2 shows the general structure of a ZBDD and a ROBDD for three generic variables c_0, c_S and c_D. Note that the rectangular nodes $x = 0$, $x = S$, $x = D$ and $x = L$ do not have to be really different nodes - it is possible that some or even all of them are in fact identical. While a

ROBDD needs up to seven nodes for these three variables, a ZBDD needs at most three. Besides, as the structure is known in advance, it is possible to optimise the BDD-construction-process. If c_0 has been set to true, setting c_S or c_D to true will lead to the zero node followed by a reduction of the ZBDD. Instead of creating a node which will be removed immediately afterwards, c_S and c_D will be set to false while c_0 is set to true. In the end this will lead to exactly the same ZBDD with less effort necessary for creating it.

Of course it would be possible to create a ROBDD with only three nodes representing four different states by using only two Boolean variables c_1 and c_2 for a component c. But this would only be equally good in comparison to three variables and a ZBDD regarding the performance. In the mean time it would be much less intuitive to program the algorithm probably leading to more bugs which can be hard to find. So this is the reason why three and not two Boolean variables were chosen to represent one component.

One disadvantage of ZBDDs still exists in the context of this work, though.

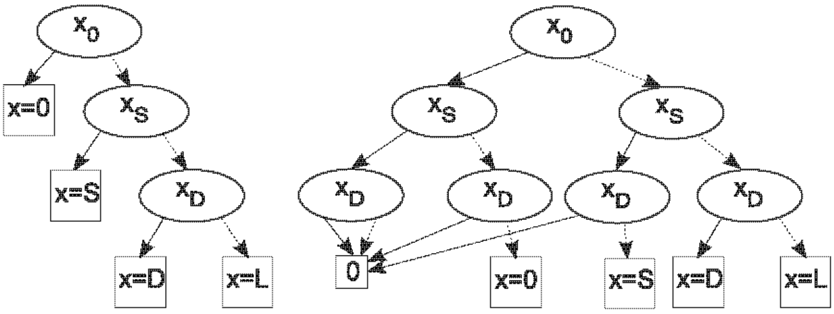

Figure 5.2: The structure of a ZBDD and an ROBDD for a generic failure expression

If an expression is reduced to the value *true* after a decomposition of a variable x, the node for x can not be linked directly to the node one-node. All variables with a lower variable order than x have to be included in the way to the node one-node. In a normal ZBDD, a chain of nodes is created, in which both outgoing edges of each node lead to the next node in the chain. For this work, this chain looks a little bit different, as dependencies have to be taken into account. The *true-chain* for n different components (x(1) to x(n)), leading to $3n$ boolean variables, is shown in Figure 5.3.

The true-chain has to take into account the problem that no component is allowed to be in more than one state, i.e $x(i)_{s1} \wedge x(i)_{s2} = false$ for $s1 \neq s2$ and $s1, s2 \in \{0, S, D\}$. So the depicted structure has to be used.

Regarding the needs for extra memory and computing power, this is not a large problem, though. Only one true-chain is needed for the whole ZBDD.

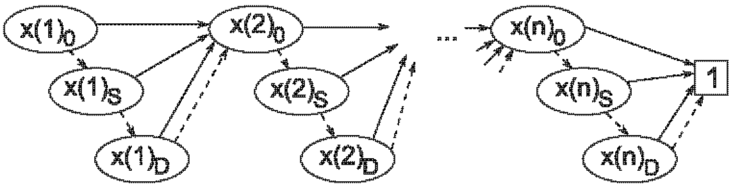

Figure 5.3: The true-chain

If the expression of a node v with a variable $x(i)_s$ is reduced to true, it can be linked to the node for the variable $x(i+1)_0$ in the true-chain.

In this work, the true-chain will not be shown in the figures explicitly, as it takes a lot of space without really giving any valuable information to the reader. Edges to nodes in the true-chain will end in rectangular boxes instead of ellipses. The name of the variable in the true-chain to which the incoming edges are leading is written inside the rectangle.

Now the decomposition itself is explained in more detail. An example, depicted in Figure 5.1, will demonstrate this process. Given is the following expression S for block C:

$$S(C) : (Input(S) \wedge x = S) \vee (Input(S) \wedge x = D \wedge y = S) \vee$$
$$(Input(D) \wedge x = s) \vee (Input(D) \wedge x = D \wedge y = D) \vee (x = S \wedge y = L)$$

This expression is transformed to the following Boolean form:

$$S(C) = (Input(S) \wedge \overline{x_0} \wedge x_S \wedge \overline{x_D}) \vee (Input(S) \wedge \overline{x_0} \wedge \overline{x_S} \wedge x_D \wedge \overline{y_0} \wedge y_S \wedge \overline{y_D}) \vee (Input(D) \wedge \overline{x_0} \wedge x_S \wedge \overline{x_D}) \vee (Input(D) \wedge \overline{x_0} \wedge \overline{x_S} \wedge x_D \wedge \overline{y_0} \wedge \overline{y_S} \wedge y_D) \vee (\overline{x_0} \wedge x_S \wedge \overline{x_D} \wedge \overline{y_0} \wedge \overline{y_S} \wedge \overline{y_D})$$

Now the decomposition can be used to create a ZBDD. Before starting the decomposition, the variables have to be ordered, for this example the following order was chosen: x_0, x_S, x_D, y_0, y_S, y_D. Note that the variable order is restricted: The three Boolean variables c_0, c_S and c_D representing one component c will always be ordered one after the other. An ordering like x_0, y_S, x_D, y_0, x_S, y_D would be invalid. We furthermore assume that the sequence of the three Boolean variables is always c_0, c_S, c_D.

Normally, in ZBDDs or ROBDDs only one variable is set at one time. For this special application the substitution process can be accelerated as it already has been explained, though. The decomposition starts with x_0. Once it is substituted with true, in the other case it is substituted with false. If x_0 is set to true, x_S and x_D is automatically set to false. After the substitution, the following two expressions remain:

$S(C|\overline{x_0}) = (Input(S) \wedge x_S \wedge \overline{x_D}) \vee (Input(S) \wedge \overline{x_S} \wedge x_D \wedge \overline{y_0} \wedge y_S \wedge \overline{y_D}) \vee (Input(D) \wedge x_S \wedge \overline{x_D}) \vee (Input(D) \wedge \overline{x_S} \wedge x_D \wedge \overline{y_0} \wedge \overline{y_S} \wedge y_D) \vee (x_S \wedge \overline{x_D} \wedge \overline{y_0} \wedge \overline{y_S} \wedge \overline{y_D})$
$S(C|x_0\overline{x_S}\overline{x_D}) = false$

In one case, the whole expression is unsatisfiable. The other case can be decomposed further. The next variable is x_S. For x_S, the same optimisation like for x_0 can be applied: if it is set to true, x_D is automatically set to false. This leads to the two following expressions:

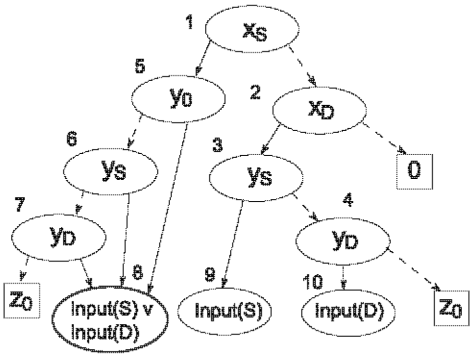

Figure 5.4: A local ZBDD for one block

$S(C|\overline{x_0}x_S\overline{x_D}) = Input(S) \vee Input(D) \vee (\overline{y_0} \wedge \overline{y_S} \wedge \overline{y_D})$
$S(C|\overline{x_0}\overline{x_S}) = (Input(S) \wedge x_D \wedge \overline{y_0} \wedge y_S \wedge \overline{y_D}) \vee (Input(D) \wedge x_D \wedge \overline{y_0} \wedge \overline{y_S} \wedge y_D)$

The decomposition process is continued recursively with the variable x_D for $S(C|\overline{x_0}\overline{x_S})$ and with y_0 for $S(C|\overline{x_0}x_S\overline{x_D})$.
The resulting ZBDD shown in 5.4 has three non trivial leaves. The zero-edge of node four leads to a node in the true-chain; its expression has been reduced to true by setting y_D to false, but as already explained it can not be linked directly with the node N_1. The next step is to eliminate the three input-nodes. The Input-expressions are basically placeholders for the expressions of the previous block, so the easiest way to eliminate the input-nodes is to replace them with the ZBDDs of the expressions they represent. Still equivalent nodes are shared. Let us assume that the expressions S and D of block B look like this:

$S(B) : (Input(S) \wedge z = S \wedge a = 0)$
$D(B) : z = D \vee (z = S \wedge a = D)$

These expressions are transformed into a Boolean form and afterwards the decomposition is continued just like for the block C. The result is shown in Figure 5.5. For this ZBDD, only one non-trivial leaf remains, Input(S). By continuing the process of replacing the non-trivial leaves with the local ZBDDs of the predecessor blocks, all of these leaves can be eliminated. The algorithm will stop if it reaches the SRC-block in which no input-expressions are allowed.

This algorithm can be described with the following pseudo code:

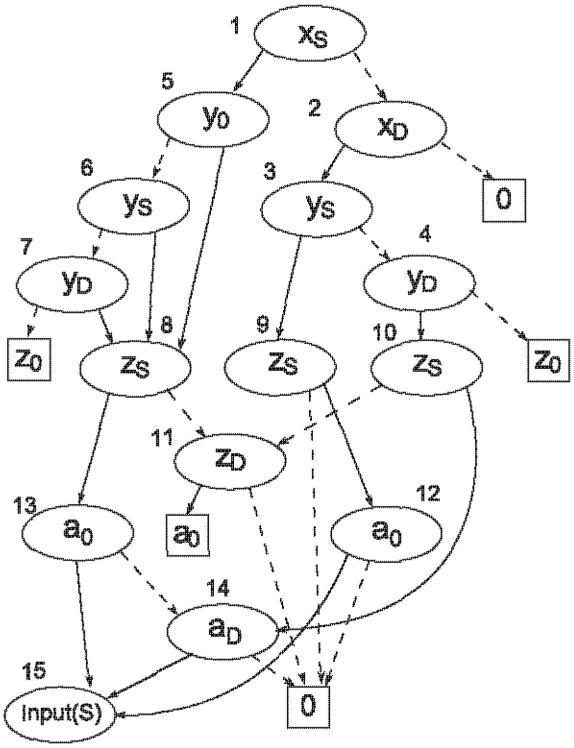

Figure 5.5: A ZBDD for two blocks

```
CreateSerialZBDD(Block finalBlock, Mode x)\\ $x \in \{S, D\}$
    ZBDD global = CreateZBDD(finalBlock.getExpression(x));
    Queue inputNodes = global.getInputNodes();
    While (!inputNodes.isEmpty())
        Node inputNode = inputNodes.getNext();
        ZBDD local = CreateZBDD(getExpression(inputNode));
        for all incoming edges e of inputNode:
            e.setTarget(local.getRoot());
        inputNodes.add(local.getInputNodes());
    return global;
```

CreateZBDD(Expression e) creates a local ZBDD for the given expression *e* by decomposition. This method will use already existing equivalent nodes in the global ZBDD if possible. *getInputNodes(ZBDD zbdd)* returns all Input-nodes of the ZBDD *zbdd*, and *getExpression()* delivers the applicable expression.
For each of the two global failure modes a own ZBDD is needed as these can only describe one kind of failure. But it is possible to create two different ZBDDs which share equivalent nodes if possible, leading to one ZBDD with two different roots.
Note that it is not necessary to implement a reduce-operator like in the original model. By using the decomposition, invalid combinations are removed automatically, and no variable can occur at two different levels of the BDD. This is another advantage of the BDD-technique.

5.3 ZBDDs for Generalised Serial IFDs

In Chapter 5.2 we assumed that no component will occur in more than one block. However, IFDs do not have such a limitation in general. This section explains how the algorithm presented in the last section can be extended in order to handle general serial IFDs. The assumption that no DEC-nodes exist still holds, though.
To be able to handle components occuring in multiple blocks of the IFD, the ZBDD needs to be extended. It is not possible to just add the ZBDDs for the local expressions if these local ZBDDs contain variables which already occurred in the global ZBDD. In this case there could be paths in which one variable could get two different values which is invalid. So two new attributes are added to the ZBDD:

- An array *localExpr* containing all local expressions.
- A hash table *compMap* mapping components to blocks in which they appear.

The construction of the ZBDD begins like 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 *localExpr*. All descendants of the current node will use these modified expressions 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 expressions which have to be used in the future. Children inherit these arrays from their parents and alter them only if their variable will also occur in other blocks of the IFD.
To clarify this idea, the example of Chapter 5.2 section is extended. The expression S of block A is defined as follows:

$$S(A) : (x = S \wedge b = 0) \vee (x = D \wedge b = S)$$

This means that the component x occurs not only in block C but also in b The equivalent Boolean form of this expression is as follows:

$$S(A) = (\overline{x_0} \wedge x_S \wedge \overline{x_D} \wedge b_0 \wedge \overline{b_S} \wedge \overline{b_D}) \vee (\overline{x_0} \wedge \overline{x_S} \wedge x_D \wedge \overline{b_0} \wedge b_S \wedge \overline{b_D})$$

The decomposition starts like in the simple case with the block C. The main difference is that before substituting any variables, *compMap* is checked. If x_0, x_S and x_D are substituted in $S(C)$, the same substitution is made for $S(A)$. The results of the decompositions are shown here:

$$\begin{aligned}
&S(A|x_0\overline{x_S x_D}) = false\\
&S(A|\overline{x_0}) = (x_S \wedge \overline{x_D} \wedge b_0 \wedge \overline{b_S} \wedge \overline{b_D}) \vee (\overline{x_S} \wedge x_D \wedge \overline{b_0} \wedge b_S \wedge \overline{b_D})\\
&S(A|\overline{x_0}x_S\overline{x_D}) = b_0 \wedge \overline{b_S} \wedge \overline{b_D}\\
&S(A|\overline{x_0 x_S}) = x_D \wedge \overline{b_0} \wedge b_S \wedge \overline{b_D}\\
&S(A|\overline{x_0 x_S}x_D) = \overline{b_0} \wedge b_S \wedge \overline{b_D}\\
&S(A|\overline{x_0 x_S x_D}) = false
\end{aligned}$$

If the decomposition reaches block A, the appropriate modified expressions have to be used for a correct result. To make sure that the right expressions are used, each ZBDD-node contains an array with pointers to the three local expressions of each block in the IFD. At the beginning of the decomposition, they point to the original expression-triples E_A, E_B and E_C, representing the three expressions S, D and L of the blocks A, B and C. But as soon as a variable is substituted in more than one block, the respective pointers are changed to the modified expressions.

Figure 5.6 shows the ZBDD for the presented example. In this figure, each node shows its pointer array, too. For example, node one contains a pointer to a triple which contains $S(A|\overline{x_0})$ instead of the original expression $S(A)$. These arrays are inherited by the children of the nodes and they are only modified if a substitution is done in more than one block. For example, node

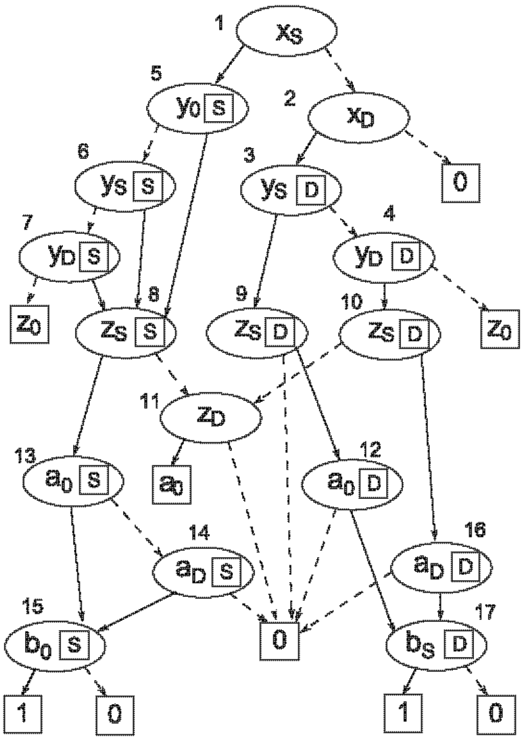

Figure 5.6: A ZBDD for a serial system with multiple occurrences of one variable

three contains the variable y_S which only occurs in block C. So it is not necessary to change the pointers, and the children of node three inherit the pointer array without any modifications. In the figure the changed arrays are visualised by including the state of x, as it is the only difference between the different arrays.

If an Input-node is reached in the ZBDD, it is replaced with the ZBDD equal to the appropriate expression in its own pointer array. So, the nodes 15 and 17 use the expressions $S(A|\overline{x_0}x_S\overline{x_D})$ respectively $S(A|\overline{x_0}\overline{x_S}x_D)$ instead of the original expression $S(A)$. In these modified expressions, x_0, x_S and x_D have been already eliminated, so these variables will not occur on two different levels of the ZBDD which would be invalid.

Note that two nodes with equivalent children can still be unified, even if their pointer arrays are different. If this is the case, their expressions are not affected by the settings of their pointer arrays anymore, like for node eleven in the example. Its remaining expression is z_D, so it does not matter how the expressions for block C look like.

5.4 BDDs for Non-Serial IFDs

The last sections only explained how to handle serial IFDs without any DEC-nodes. As these are very important, though, a solution is needed which can also include non-serial IFDs.

DEC-blocks are specified by different expressions than ST-, WD- and SRC-blocks. They do not contain any components which can be translated in Boolean variables. Instead they define how the results of the predecessor blocks have to be combined.

For this combination, a special form of BEDs can be used. The second reduction rule of BEDs have to be altered in order to use them with ZBDDs. This modified BED is called *Zero-suppressed Boolean Expression Diagram* (ZBED) The rules for ZBEDs are shown in Figure 5.7.

For DEC-nodes three ZBEDs are created, one for each expression (S, D and L) of the block. The leaves of the ZBED represent the predecessor blocks of the DEC-node in a certain state (S,D,L). Operator nodes are used to link the leaves according to the expressions of the DEC-block. An example for a rule of a DEC-node with three predecessor nodes X, Y and Z is shown in Figure 5.8. The ZBED is created like an operator tree for the represented expression.

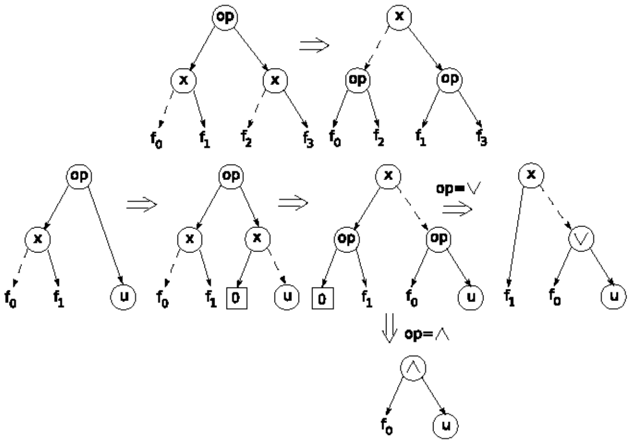

Figure 5.7: The reduction rules for ZBEDs

The linking process with the predecessors and the successor block is similar to the method described in 5.2. The leaf representing the predecessor non-DEC-block B in a certain state X is substituted with the appropriate ZBDD for the expression $X(B)$. A leaf representing the predecessor DEC-node D in state X is replaced by the ZBED for the expression X. If a non-DEC-block B is the successor of a DEC-node D, the ZBED for the expression X of D

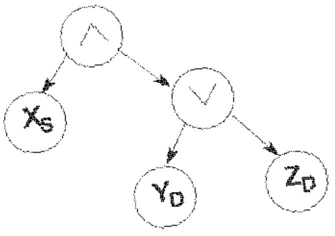

Figure 5.8: A local ZBED for the expression $X = S \wedge (Y = D \vee Z = D)$

substitutes the $Input(x)$-node of B. So in general, the following algorithm was 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 main difference between serial IFDs and IFDs with DEC-blocks can be recognized in the 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 ZBDD. For the application of the reduction rules there are two possibilities in general:

- Apply the reduction rules once after all local ZBDDs and ZBEDs have been linked.
- Apply the reduction rules iteratively after new local ZBDDs or ZBEDs have been added, and stop this reduction at *Input*-nodes or ZBED-leaves.

In this work, the second approach was used as the reduction rules can already simplify the diagram significantly while constructing it. As soon as operator nodes are linked to the terminal nodes N_0 or N_1, parts of the BDD can be removed totally.

It is possible to apply Boolean operators directly to ZBDDs, so theoretically it would not be necessary to use the BED-technique. In this case it has a major advantage, though. In order to apply Boolean operators directly, the two sub-ZBDDs on which the operator is used has to be known in advance. In this case, it would not be possible to build the BDD block by block which is a severe disadvantage. In contrast, the BED-technique allows us efficient local substitutions.

5.5 Illustrative Example

In this section, the presented techniques from the previous sections are combined. A detailed explanation is done with the help of an artificial and small, but still rather complex example including a DEC-node and a component which occurs in two blocks.

The high level model of the example is depicted in Figure 5.9. The expressions for the single blocks look like this:

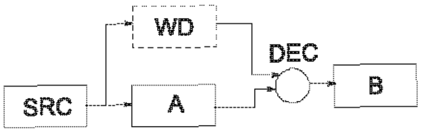

Figure 5.9: The high level model for the illustrative example

$S(B) : (Input(S) \wedge b = 0) \vee b = S$
$D(B) : (Input(D) \wedge b = D) \vee (Input(S) \vee b = S)$
$S(DEC) : A = S \vee WD = S$
$D(DEC) : WD = D$
$S(A) : (Input(S) \wedge a = 0) \vee a = S$
$D(A) : (Input(D) \wedge a = 0) \vee a = D$
$S(WD) : wd = S \wedge b = S$
$D(WD) : input(L) \wedge wd = D \wedge b = D$
$S(SRC) : s = S$
$D(SRC) : s = D$
$L(SRC) : s = L$

All the other expressions are set to $false$.

The aim is to create a ZBDD representing this model. At first a variable ordering has to be chosen. The following order is used: b_0, b_S, b_D, a_0, a_S, a_D, wd_0, wd_S, wd_D, s_0, s_S, s_D.

The construction starts at the final block B. Both $S(B)$ and $D(B)$ are transformed into their Boolean form. Afterwards, the decomposition commences for each expression, resulting in two ZBDDs. As these two ZBDDs share equivalent nodes, we will see them as one ZBDD with two different roots. Figure 5.10 shows the resulting ZBDD for the block B. As the component b occurs also in block WD, it is necessary to make also a substitution of the expressions of block WD like presented in Section 5.3. As the only Boolean variables affected by this are the variables b_0, b_S and b_D, not the whole pointer array is shown in the diagram, but only the state of the component b.

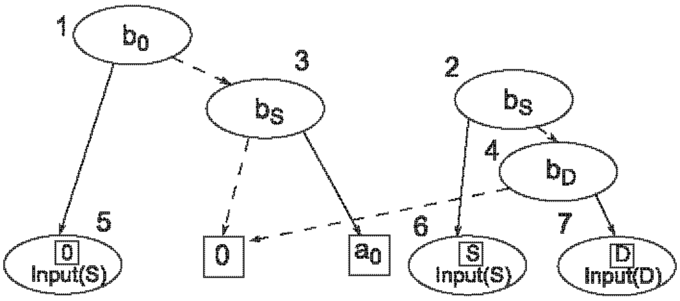

Figure 5.10: The ZBDD for the expressions $S(B)$ and $D(B)$

The nodes one and two are the roots for the expressions S resp. D. The input-nodes show not only their input-value, bot also the state of b. So five and six are not equivalent as in five b was set to 0 and in six it was set to S.

The next step is to replace the input-nodes with the appropriate ZBEDs of Dec. The resulting BDD is depicted in Figure 5.11. As no variables are changed in this step, the pointer arrays of the input nodes are taken over unaltered by the nodes of the added ZBEDs.

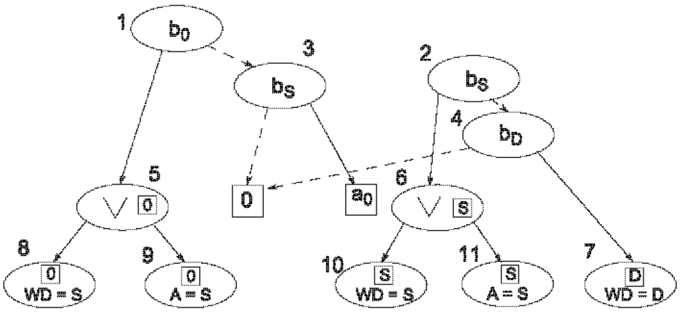

Figure 5.11: The BDD for the blocks B and Dec

Now it is necessary to replace the leaves again. The leaves for the nodes nine and eleven are replaced by the ZBDD for the expression $S(A)$. As neither

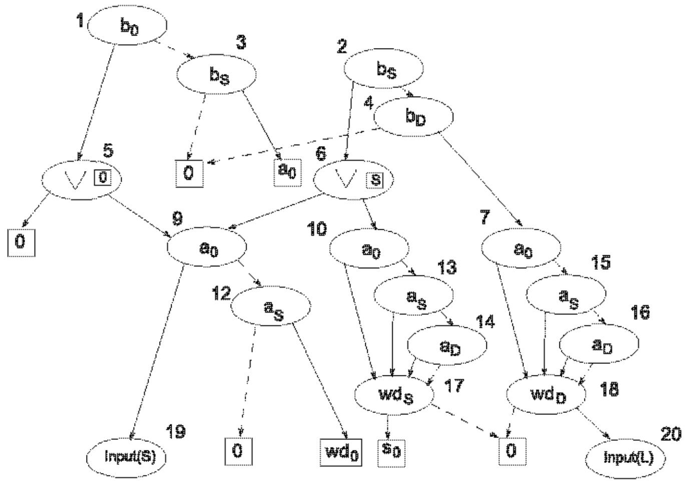

Figure 5.12: The BDD for the blocks B, Dec, Wd and A

$S(A)$ nor its predecessors contain the component b, the nodes nine and eleven can be unified. This is different for the nodes seven, eight and ten, though: as the block WD contains b, these three nodes do not represent the original expressions $S(WD)$ resp. $D(WD)$, but modified ones. Node seven uses the expression $D(WD|\overline{b_0}\overline{b_S}b_D)$, node eight is a place holder for $S(WD|b_0\overline{b_S}\overline{b_D})$ and node ten represents the expression $S(WD|\overline{b_0}b_S\overline{b_D})$. These three expression look like this:

$$D(WD|\overline{b_0}\overline{b_S}b_D) = Input(L) \wedge wd = D$$
$$S(WD|b_0\overline{b_S}\overline{b_D}) = false$$
$$S(WD|\overline{b_0}b_S\overline{b_D}) = wd = D$$

So, eight can be simply substituted with the Zero-node N_0, while seven and ten are replaced by non-trivial ZBDDs. For these ZBDDs, it is necessary to introduce nodes for the Boolean variables a_0, a_S and a_D although these do not appear in the expressions. As these variables have a higher order than the variable for the component wd, which are part of these expressions, they have to be included in the ZBDD. If these nodes would be missing it would mean that all three Boolean variables of a have to be false, i.e. the component a is in state L. But in fact it is not important in which state the component a is, as long as it is valid. So three nodes are introduced in a similar structure like the nodes of the true-chain. The resulting BDD is shown in Figure 5.12.

The next step is to apply the ZBED reduction rules. Node five can be removed immediately - it is an or-node with N_0 as child. So it is deleted,

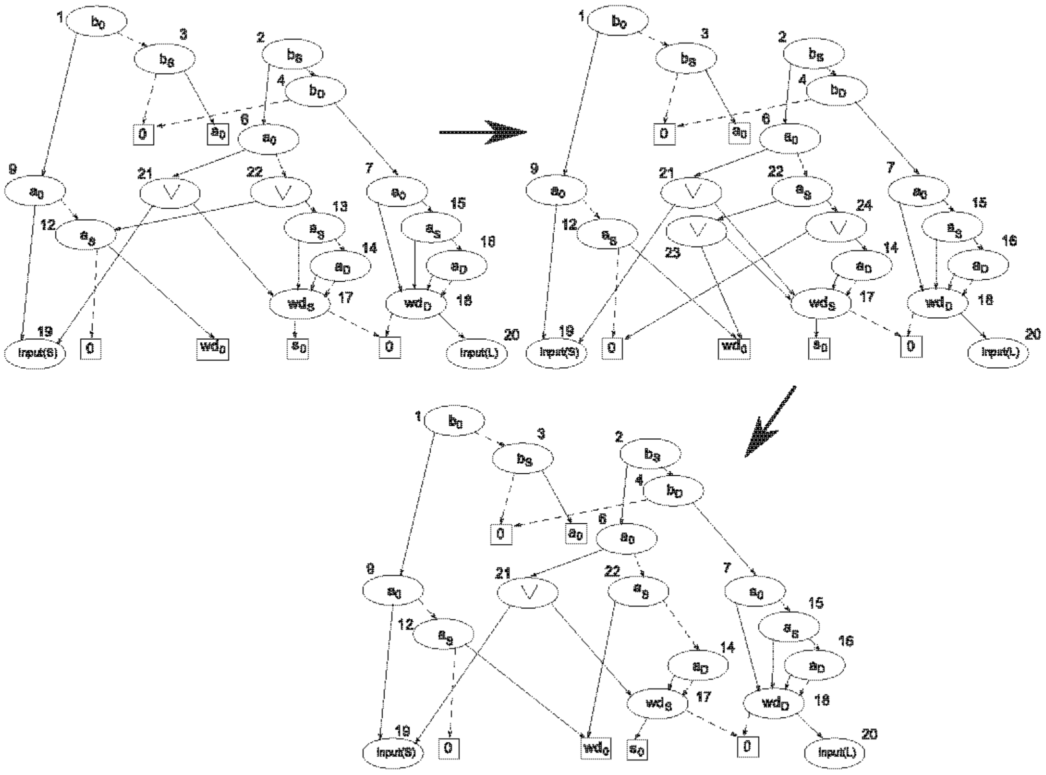

Figure 5.13: The ZBED-reduction rules applied on the BDD

and its parent, node one, is linked with the other child of node five instead. Node six can not be removed that simple. So it is necessary to stepwise apply the reduction rules for the ZBEDs. After the first application of the rules, two new or-nodes 21 and 22 are introduced, while node six has been transformed to a variable node. It is not possible to apply the rules at node 21 as one of its children is an Input-node - first the input node has to be substituted. So for now the ZBED-reduction rules are only applied for node 22. After a further step, two new or-nodes 23 and 24 appear. One of node 23's children is a node in the true-chain, representing the Boolean expression *true*. So node 23 can be removed, and its incoming edge is linked to the node in the true-chain instead. Node 24 can be removed, too, as one of its children is the zero node. The resulting BDD after applying the rules is depicted in Figure 5.13.

Now the remaining input-nodes have to be substituted. Afterwards, the ZBED-rules have to be applied a last time on the last or-node 21. After these steps, no input- or operator-nodes remain in the BDD, so the algorithm will finally terminate. The final version of the ZBDD is shown in

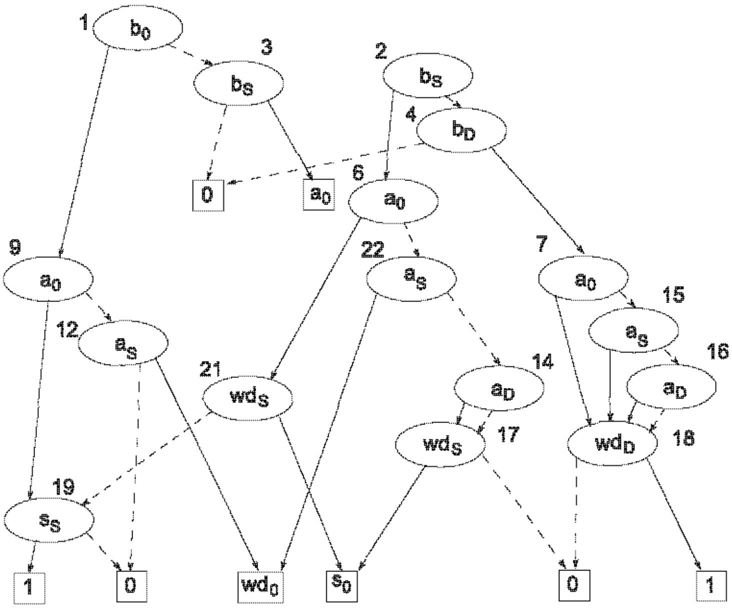

Figure 5.14: The final ZBDD

Figure 5.14.
This example reveals several interesting properties of the proposed algorithm. It is capable to create one BDD for two different expressions. So fewer BDD-nodes are needed for the overall system. Besides that, the algorithm only creates sub-BDDs for local expressions if these are really needed. For the example, two of the defined expressions, $D(A)$ and $D(Src)$ were never transformed into a sub-BDD as these expressions are neither a part of the global expression D nor does the global expression S contain them. Furthermore, the algorithm demonstrates that several BDD-techniques can be easily combined, in this case ZBDDs and BEDs.
The algorithm has been developed specially for IFDs. It uses several properties of the IFD-model, like restrictions regarding the variable ordering or the fact, that each component may only be in one state at one time. Without these properties the algorithm would not get correct results. This algorithm can not be used for other similar problems without an adaption. It is also impossible to use generic BDD-algorithms for solving the IFD-model without modifying them a lot.

5.6 Quantitative Evaluation

A basic assumption for the quantitative evaluation of ZBDDs in order to estimate the overall reliability is that all variables used in the ZBDD are stochastically independent. In the presented case this is not true, though. For each component c there are three variables c_0, c_S and c_D from which at most one can be true, so dependencies have to be taken into account.

The solution for this problem is to use conditional dependencies. Be c_t the state of the variable at the time t. It is assumed that the unconditional probabilities $P(c_t = 0)$, $P(c_t = S)$, $P(c_t = D)$ and $P(c_t = L)$ are known. It holds: $P(c_t = 0) + P(c_t = S) + P(c_t = D) + P(c_t = L) = 1$. But these probabilities are not used directly, instead the following probabilities are used for the variables c_0, c_S, and c_D:

- $P(c_t = 0)$ for c_0
- $P(c_t = S)|c_t \neq 0) = \frac{P(c_t=S)}{P(c_t=S)+P(c_t=D)+P(c_t=L)}$ for c_S
- $P(c_t = D|c_t \neq 0 \wedge c_t \neq S) = \frac{P(c_t=D)}{P(c_t=D)+P(c_t=L)}$ for c_D

For defining these probabilities, the restrictions regarding the variable ordering were taken into account.

Furthermore, the calculus for ZBDDs is altered a little bit. For the probabilities of the high edges of c_0 resp. c_S not c_S resp. c_D are seen as next variables in the variable ordering, instead the variable d_0 of the next component d is chosen.

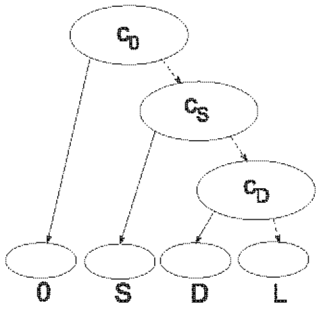

Figure 5.15: The structure of a ZBDD for one component c

The ZBDD in Figure 5.15 shows the general structure of a ZBDD for a single component. For a correct quantified evaluation of the ZBDD, the probabilities for the nodes 0, S, D and L have to be the same than the appropriate unconditional probabilities. This has to be proved, though. It is obviously true for the node 0. For the other nodes, some calculation has to be done.

The probability of the node S is calculated as follows:

$$P(S) = P(c_t = S)|c_t \neq 0) \cdot P(c_t \neq 0) = \frac{P(c_t{=}S)\cdot(P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L))}{P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L)} = P(c_t = S)$$

The same way the probability for the node D is designated:

$$P(D) = P(c_t \neq 0) \cdot P(c_t \neq S|c_t \neq 0) \cdot P(c_t = D|c_t \neq 0 \wedge c_t \neq S) =$$
$$\frac{P(c_t{=}D)|c_t{\neq}0{\wedge}c_t{\neq}S)\cdot P(c_t{\neq}0)\cdot(P(c_t{=}D)+P(c_t{=}L))}{P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L)} =$$
$$\frac{P(c_t{=}D)\cdot(P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L))\cdot(P(c_t{=}D)+P(c_t{=}L))}{(P(c_t{=}D)+P(c_t{=}L))\cdot(P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L))} =$$
$$P(c_t = D)$$

Finally, the probability for node L is calculated:

$$P(L) = P(c_t \neq 0) \cdot P(c_t \neq S|c_t \neq 0) \cdot P(c_t \neq D|c_t \neq 0 \wedge c_t \neq S) =$$
$$\frac{P(c_t{\neq}D)|c_t{\neq}0{\wedge}c_t{\neq}S)\cdot P(c_t{\neq}0)\cdot(P(c_t{=}D)+P(c_t{=}L))}{P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L)} =$$
$$\frac{P(c_t{=}L)\cdot(P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L))\cdot(P(c_t{=}D)+P(c_t{=}L))}{(P(c_t{=}D)+P(c_t{=}L))\cdot(P(c_t{=}S)+P(c_t{=}D)+P(c_t{=}L))} =$$
$$P(c_t = L)$$

For all four nodes the probabilities are correct. q.e.d.
The problem of the quantified evaluation is another reason that ZBDDs were chosen instead of ROBDDs. The simple structure of the ZBDDs makes it easy to define the values for probabilities of the Boolean variables with only a minor change of the calculus. In an ROBDD, the problem of dependencies between the different Boolean variables for one component can not be solved that elegantly.

Chapter 6

Implementation

This chapter deals with the details of the implementation of the proposed algorithm. It was implemented in Java. The implementation reads files describing the IFD and transforms it into an internal model, creates the appropriate BDD and evaluates it quantitatively. This chapter explains the structure of the implementation, chosen heuristics and other minor optimisations.
The aim of this implementation is mainly a proof of concept. It probably could be optimised much more, but obviously only a limited amount of time could be used for developing the software.

6.1 Defined Datatypes

At first it is necessary to define important datatypes used in the implementation. In general, these datatypes are either used to describe entities of the IFD-model or parts of the BDDs. This section will explain the datattypes which were defined for the implementation.

6.1.1 The IFD-model

The IFD-model has a high- and a low-level model. For the high-level model, a graph is used. In the IFD-software-package, this graph is represented by five classes.
The class IFD describes the IFD in general. It contains the attributes *blocks*, *edges* and *components*, which are lists of all blocks, edges and components which occur in the IFD. Each block and each edge gets a unique ID generated by the class itself. These IDs can be used to sort and access the blocks and edges more easily.

Important methods in the class IFD are *getEdges()* and *getBlocks()* which return the whole lists of edges and blocks. Besides that it is possible to add or remove single blocks or edges with the methods *addBlock*, *addEdge*, *removeBlock* and *removeEdge*. The method *getComponent(String name)* returns the component with the given name if such a component exists, and the methods *addComponent* and *removeComponent* can add or remove components to the IFD.

For modelling the blocks, three classes are used: The abstract class *AbstractBlock* as super class, the class *NonDecBlock* for SRC- and ST-blocks and the class *DecNode* for DEC-nodes.

AbstractBlock contains attributes for the name, the type, the ID and the parent IFD. Furthermore it stores if the block is the final block. Besides that, the class inherits the two lists for incoming and outgoing edges from the abstract class *Node*. For all attributes, getters and setters are defined.

The classes *DecNode* and *NonDecBlock* inherit all attributes and methods from *AbstractBlock*. Furthermore, they contain the three expressions for the states S, D and L and the appropriate getters and setters.

The edges are represented by the class *BlockEdge*. This class inherits the attributes *source* and *target* from the abstract class *Edge*. Furthermore, it has the attributes *id* and *ifd*. For all these attributes getters and setters exist.

Figure 6.1 depicts an UML-diagram[30] representing the class structure for the high-level model of the IFDs.

6.1.2 Expressions

The next important data structures are expressions. Expressions are used in three different variations:

- Boolean expressions
- Extended expressions for non-DEC-blocks
- Extended expressions for DEC-nodes

For all three kinds of expressions the super class *Expression* was created. It has the following attributes:

- int *type* (discriminating between disjunction terms, conjunction terms and atomic expressions)
- boolean *bool* (true if the expression is a Boolean expression, false otherwise)

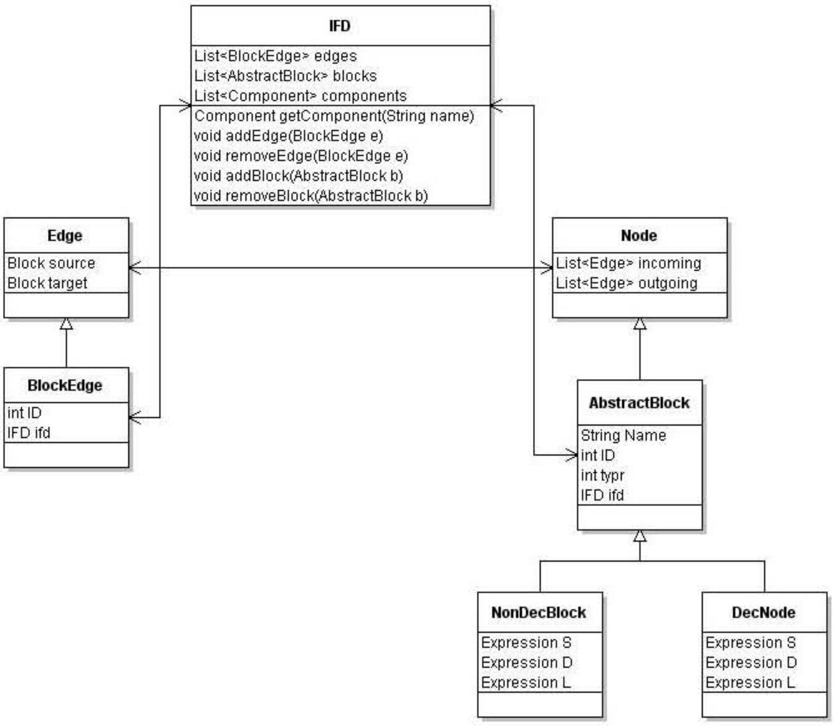
IFD
List<BlockEdge> edges
List<AbstractBlock> blocks
List<Component> components
Component getComponent(String name)
void addEdge(BlockEdge e)
void removeEdge(BlockEdge e)
void addBlock(AbstractBlock b)
void removeBlock(AbstractBlock b)
Edge
Block source
Block target
BlockEdge
int ID
IFD ifd
Node
List<Edge> incoming
List<Edge> outgoing
AbstractBlock
String Name
int ID
int typr
IFD ifd
NonDecBlock
Expression S
Expression D
Expression L
DecNode
Expression S
Expression D
Expression L

Figure 6.1: The UML-diagram for the high-level model for the IFD-approach (without getters and setters).

Besides that either two Expressions of type *Expression* (*subExpr1* and *subExpr2*) for disjunction and conjunction terms or an atomic expression *atom* of type *AtomExpr* are defined.
The class *AtomExpr* is an abstract super class for all kinds of atomic expressions. Its only attribute is the type, which can be either an Input, Fault, Boolean or Constant expression. For each type, a subclass is defined. Instances of the type *InputExpr* store the value of the input expression (S, D or L). The class *FaultExpr* represents components and their four different states: the faulty states S, D and L and the correct state 0. The class *BoolExpr* is used for Boolean variables. This class includes an attribute for the name of the variable and an attribute *negated* which shows if this variable is used as complement. The class *ConstExpr* represents the Boolean constants, its only attribute is the value of the constant (true or false). The UML-diagram for the expressions is depicted in Figure 6.2.

This structure of the expressions can be used for represent all three variations which occur in the implementation. Boolean Expressions are represented by instances of the type *Expression* in which the atomic expressions are either of the type *BoolExpr* or *ConstExpr*. The extended expressions

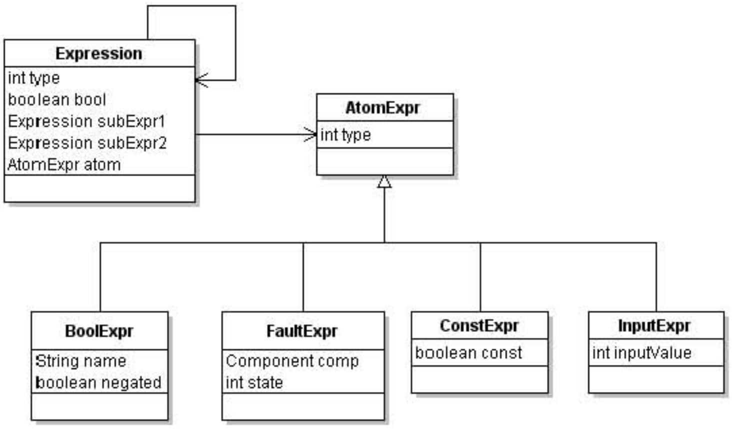
Expression
int type
boolean bool
Expression subExpr1
Expression subExpr2
AtomExpr atom
AtomExpr
int type
BoolExpr
String name
boolean negated
FaultExpr
Component comp
int state
ConstExpr
boolean const
InputExpr
int inputValue

Figure 6.2: The UML-diagram for the expressions (without getters and setters).

for non-DEC-blocks are instances of the type *Expression* where the atomic expressions are of the type *FaultExpr*, *InputExpr* or *ConstExpr*. Note that there are further limitations regarding input expressions depending on the type of block. For SRC-blocks, no input expressions are allowed, while for ST- and WD-blocks their values are limited to either S and D or L.
The extended expressions for DEC-nodes are represented like regular Boolean expressions. The name of the block and its state can be encoded in the name of the Boolean variable. An expression like $(A = S)$ is represented by the Boolean variable A-S. At first, this simple solution looks a little bit too sloppy as there is the possibility of stochastic dependencies. But such dependencies are only a problem if these expressions are used in the calculation of the probabilities. In this case, these variables are not used for this purpose, though. They are just placeholders for subexpressions, in which stochastic dependencies are taken into account correctly.

6.1.3 The BDD

For representing the BDD, three classes are used. The class *ZDD* is for the diagram in general. Important attributes are the *nodeMap*, a hash table which maps BDD-nodes to themselves; the Vector *orderedVariables* which represents an ordered list of all variables used in this BDD; the hash table *varMap* which maps the names of the used variables to sets of block in which they occur and finally the *ifd* which is a link to the IFD for which this BDD is created. Furthermore, two BDD nodes are defined as roots for the BDDs representing the failure modes S and D on a global scale.

The nodes in the diagram are represented by instances of the type *ZDDNode*. This class is a subclass of *Node* and adds the following attributes to the already existing ones (outgoing and incoming edges):

- ZDD main
- int type
- Expression[] listS, listD, listL
- AbstractBlock currentBlock
- int variableCount
- Expression currentExpr

The attribute *main* is a pointer to the central BDD with the global data needed for the construction. The *type* represents the possible types of nodes: Variable nodes like in standard ZBDDs, And- or Or-nodes like in BEDs or leaves for the Boolean constants true and false. *listS*, *listD* and *listL* are the pointer arrays of the nodes described in Chapter 5.3. Each failure mode gets its own array with pointers to the expressions which will be used for the next blocks. *currentBlock* is a pointer to the block in which the decomposition takes place for this node. *currentExpr* is an eventually reduced local expression of *currentBlock*. The attribute *variableCount* is an integer representing the variable of the block, *orderedVariables[variableCount]*.
Together the three lists, *currentBlock*, *currentExpr* and the current variable define if two different nodes are equivalent. If all these attributes are identical, the decomposition will lead to equivalent children nodes. So both nodes can be merged due to the reduction rules of ZBDDs. In order to check the equivalence efficiently, the hash code of each node is based on these attributes. Then the hash table *nodeMap* can be used to check in linear time [60] if there is an equivalent node to a newly created one. If this is the case, the existing node is used instead of the new one. With this technique it is possible to avoid the creation of many unnecessary nodes.
The edges in the BDD are instances of the class *ZDDEdge*. This class inherits its attributes *source* and *target* from its super class *Edge*. Besides that, an Boolean attribute *high* defines if the target of the edge is the one- or zero-child of the source. Figure 6.3 shows an UML-diagram for the classes used for representing the BDD.

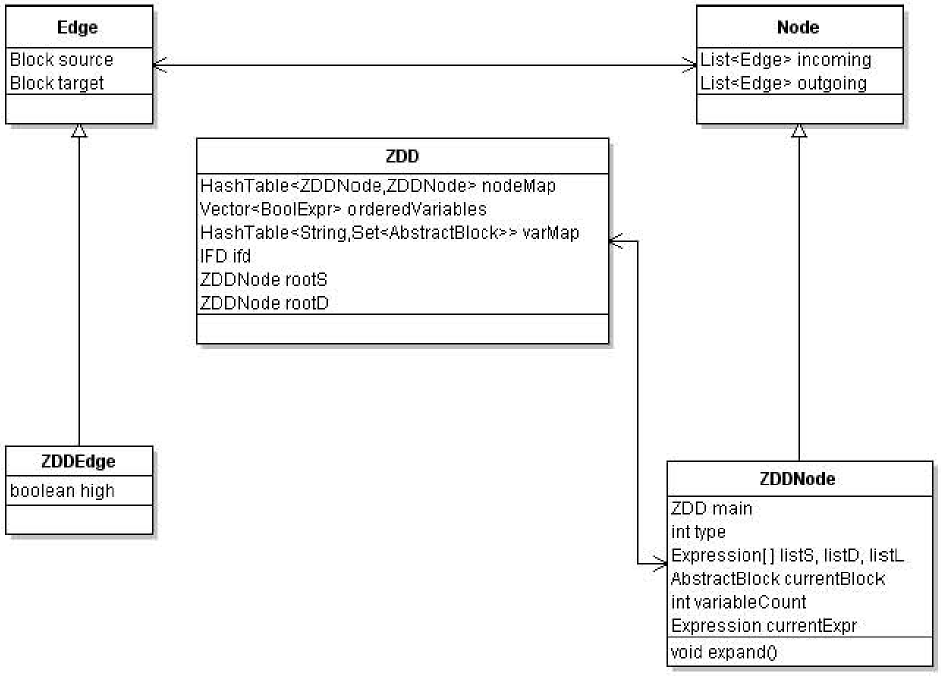
Edge
Block source
Block target
Node
List<Edge> incoming
List<Edge> outgoing
ZDD
HashTable<ZDDNode,ZDDNode> nodeMap
Vector<BoolExpr> orderedVariables
HashTable<String,Set<AbstractBlock>> varMap
IFD ifd
ZDDNode rootS
ZDDNode rootD
ZDDEdge
boolean high
ZDDNode
ZDD main
int type
Expression[] listS, listD, listL
AbstractBlock currentBlock
int variableCount
Expression currentExpr
void expand()

Figure 6.3: The UML-diagram for the BDD (without getters and setters).

6.2 Reading and Preparation of Models

This section explains how IFDs can be stored in files, and how these files are transformed into a computer readable model. This model is the base for the creation of the BDD and the following evaluation.

6.2.1 The File Format

IFD-models are stored in IFD-files, a file format developed especially for this thesis. One example for the Remote Redundancy Architecture (See Chapter 7.1) is shown in the Appendix D. IFD-files consist of four different sections: The file header, the component section, the block section and the edge section.

The header consists only of two lines. The first line denotes the version of the file format, which allows for changing the format in the future and still be able to load older examples. The second line defines the mission time of the system.

The component section begins with the line %*Components*. Afterwards, each line describes one component by giving its name, followed by an integer for its type and three float point values for the failure rates for the modes S, D and L. A component line looks like this:

r1 0 1E-09 1E-09 0

This line defines a component *r1* with the failure rates $1E-09$ for the modes S and D and the failure rate 0 for the mode L. Note that the type is a remnant from the original model where several component types (bitflip, transient, fault test) were possible. Now only one general component type is allowed, defined by 0.
The line %*Blocks* ends the component section and starts the block section. Each block is described in eight lines, for example:

(1) COM2R
(2) 5
(3) 0
(4) false
(5) input(S);d(bus1,S)#d(bus2,S);d(ecu2,S)
(6) input(D)#d(ecu2,0)#d(bus1,0);input(D)#d(ecu2,0)#d(bus2,0)
(7)
(8)———

The first line (COM2R) defines the name of the block. The second line (5) gives its ID. The third line (0) describes its type (ST: 0, WD: 1, DEC: 2, SRC: 3). The Boolean value in the fourth line defines if this block is the final block. The lines five to seven define the expressions S, D and L. For non-DEC-nodes, still the old format described in Chapter 3.2 is used. *input(S);d(bus1,S)#d(bus2,S);d(ecu2,S)* is equal to $input(S) \wedge ((bus1 = S) \vee (bus2 = S)) \wedge (ecu2 = S)$, an empty line is equal to false. The last line is only used to mark that the definition of this block is finished.
For DEC-nodes, the block's definition looks a little bit different:

(1) Sen2
(2) 7
(3) 2
(4) false
(5) $(\&P2\& = \#S\ OR\ \&R2\& = \#S)\ OR\ \&P2\& = \#D$
(6) $(\&P2\& = \#D\ AND\ \&R2\& = \#D)$
(7) $false$
(8) ———-

For these nodes, the expressions in the lines five to seven are defined in a different way. The names of the predecessor blocks used in the expressions are written between two ampersands, their state is defined by $\#S$, $\#D$ or $\#L$. Unlike for non-DEC-nodes the expression do not have to be in DNF, brackets and Boolean operators can be used freely.

The last section of the file begins with the line *%Edges*. In this section, the edges in the IFD are defined the following way:

(1) 3
(2) 1
(3) 4
(4) ———-

The first line is the id of the edge, the second line defines the id of the source-block and the third line gives the id of the target block. The fourth line is used to show that the definition of the edge is finished.

6.2.2 Creation of the Initial IFD-Model

Out of the IFD-file the IFD-model is created using the presented datastructures. This model is used as base for the evaluation. The reading process is implemented in the class *IFDReader*.

This class generates the IFD-model with the method *generateIFD*. It creates an empty IFD and adds components, blocks and edges like specified in the given file. Components and edges can be created directly by just reading the lines and creating appropriate Object of the type Component resp. ZDDEdge. For Blocks, this is more complicated as their descriptions contain expressions.

Non-DEC-blocks are not such a large problem as their expressions are always given in DNF. This allows for directly reading the lines describing the expression and creating the matching object of the type *Expression*. DEC-nodes are more complicated, though. Their expressions are not restricted. They can contain brackets and the structure of the expression is not known in advance.

Such expressions can be described with a *context-free grammar*[22][54]. To check if an expression can be generated by a given context-free grammar is done by using a *Parser*. Furthermore, it is necessary to do a lexical analysis in order to recognise different symbols used in the expression. For this lexical analysis a regular expression has to be defined, which can be recognised by the so called *Scanner*[10]. Together the scanner and the parser are called *compiler*. Programming compilers directly is quite error prone, though. So today generators are used to create the code for these programs. For the generation it is necessary to specify the language formally and then run the generator.

In this thesis, the *Compiler Construction Kit* (CCK, [1]) was used which creates compilers in Java. For creating the parser, a cup-file has to be created which contains both the specification of the expressions and instructions how to create the respective expressions in the software. The formal specification of the expression is based on the BNF-notation from Chapter 4.2. In order to describe the expressions, terminal and non-terminal symbols are defined. Terminals are returned by the scanner and contain the symbols for brackets, identifiers and Boolean operators. They are specified formally in a flex-file. The only non-terminal symbol used is the expression which we want to extract. For this purpose it is described in a formal notation similar to BNF. An extract is shown here:

```
expr ::= BO expr:e BC
{: RESULT = e; :}
| expr:e1 AND expr:e2
{: RESULT = new Expression(e1,e2,Expression.CONJUNCT,false); :}
| expr:e1 OR expr:e2
{: RESULT = new Expression(e1,e2,Expression.DISJUNCT,false); :}
| IDENTIFIER:id EQ S
{: RESULT = new Expression(new BoolExpr(id+"_S", false,0)); :}
| IDENTIFIER:id EQ D
{: RESULT = new Expression(new BoolExpr(id+"_D", false,0)); :}
| IDENTIFIER:id EQ L
{: RESULT = new Expression(new BoolExpr(id+"_L", false,0)); :}
| TRUE
{: RESULT = new Expression(Expression.trueExpression); :}
| FALSE
{: RESULT = new Expression(Expression.falseExpression); :}
;
```

There are eight different possibilities defined for the expressions. Each one is followed by the instruction in braces for creating an instance of *Expression* equivalent to the formal specification.

These formal specifications used for generating the classes *Lexer*, *Parser* and *Sym*. With these classes, the complex expressions can be transformed into the needed format. Using a generator allows for an easy and fast implementation of this transformation process.

6.3 BDD-Creation Process

After transforming the input files into an internal model, this model is used as base for the BDD-creation algorithm.
Before starting the construction of the BDD, the variable ordering has to be defined. This ordering is important as it influences the structure and the size of the BDD. For gaining an efficient variable ordering, the following steps are taken:
The blocks of the IFD are ordered based on a breadth-first search (BFS), beginning with the final block. An example for a possible order is depicted in Figure 6.4. Then the variables will be extracted out of the expressions of the non-DEC-blocks. The extraction order of the blocks is defined by the BFS-order, it starts at the non-DEC-block with the lowest number (In the example the block V2). If a variable is extracted from the expressions S, D or L of a block and not included in *orderedVariables*, it is added at the end. Finally *orderedVariables* will contain all variables occurring in the IFD, sorted by a BFS-order. By using this BFS-order we make sure that the variable order fits to the structure of the IFD which leads to a more compact BDD.
Furthermore, the triples of variables c_0, c_S and c_D for one component c will always follow each other directly. This allows several exploits which increase the efficiency.
To implement the algorithm discussed in the previous chapter, mainly one method was used. The method *expand* in the class *ZDDNode* takes a still childless ZDDNode and links it to the appropriate children based on the attributes of this node by substituting the variable like described in Chapter 5. If new nodes are created this way, the expand-method is applied on them, too. The global BDD is constructed by creating two nodes for both roots of the BDD and applying the expand-method on them. This way the whole BDD is created recursively.
Central elements for this construction process important for all nodes like the hash tables are stored in the main ZDD-object. Attributes only important

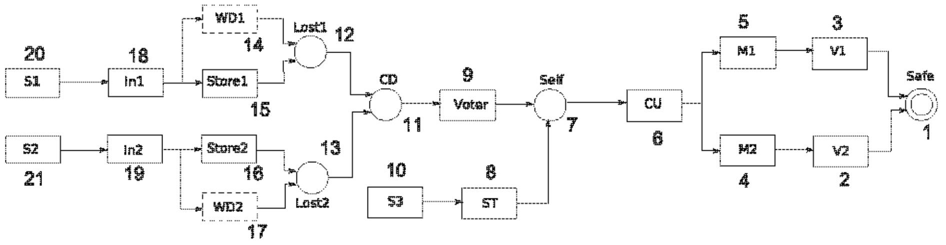

Figure 6.4: The BFS-order of an IFD

for the nodes itself like the current expressions are stored in the nodes itself. In order to reduce the memory requirements and decrease the computation time, all data which is not needed anymore is removed. For example a node with local expressions of a block B is taken. Pointers in the pointer array are needed only for the predecessor block's expressions. The expressions of the B and the successors do not affect the BDD anymore at this point. And if the expression of a component is equal to *true* then all pointers to other block's expressions are unnecessary. So unneeded pointers are replaced with the null-pointer.

This measure has several positive outcomes. At first, many of these expressions can be removed from the memory by the garbage collector if all pointers to them are set to null. Furthermore, it enhances equality tests of nodes. For these, the local expression, the variable and the pointer array have to be equal. If many of the pointers in the pointer array are just null pointers, it is much easier to do an equality test. Besides that it can happen that more node-equivalences are detected immediately. We assume that there are two nodes with equal expressions and the same variable. The pointer array differ for the pointers belonging to successor blocks of B, but are the same for the predecessors. If an equality test would take into account the whole pointer array, it would return the result that they are not equal. But if both nodes are expanded, they would have equal children. So from a BDD-point of view, these two nodes are equal and an equality test should take into account only predecessor blocks. The easiest way to ensure this is simply to replace all pointers to successor blocks with null.

In the implementation, known characteristics of the IFD-model and the resulting BDD can be taken into account for further optimisations and correct results. One problem in the BDD is that there are dependencies between the variables of the component triples. For a component c the three variables c_0, c_S and c_D are created, from which at most one can be true. Assume that the current variable of a node is c_0, but neither the local expression nor one expression in the pointer array contains this variable. In a standard ZBDD this would lead to a situation in which both outgoing edges are leading to the same child node. In this child node, the next variable is substituted, in this case c_S. But we can be sure that neither c_S nor c_D will appear if c_0 is missing, which would lead to a chain of three nodes, where both outgoing edges are going to the same child. Such a chain would allow an invalid combination like all three variables are true, too. So it has to be intervened in order to create a correct model. If the variable c_0 is not a part of the expression, the following structure depicted in Figure 6.5 is included instead.

So two unnecessary substitution steps are avoided, and the resulting ZBDD is a correct description of the system's behaviour.

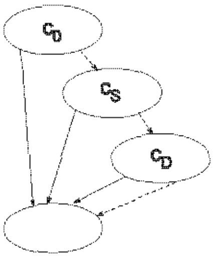

Figure 6.5: Structure of a ZBDD with an unused component c in the expression

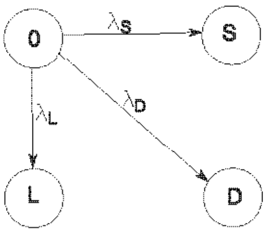

Figure 6.6: A Markov Chain modelling the component failures

The property that only one of these three variables is allowed to be true can be used also in another way. If an expression contains these variables and c_0 is set to true, c_S and c_D are automatically set to false. If c_S is set to true, c_D is automatically set to false. Without this measure still a correct ZBDD would be created, but for this purpose extra nodes would have been created which are reduced away directly afterwards.

6.4 Quantified Solving Process

The quantified evaluation of the BDD can be easily done like described in Chapter 5.6. But to do so, the probabilities of the different failure modes of each component must be known. To estimate these, Markov chains are used. For the implementation, the simplest possible model was chosen. We assume that there is no repair and that the state of a component will not change once it has failed. So only three different failure rates λ_S, λ_D and λ_L are needed. The Markov chain is shown in Figure 6.6.

To calculate the probabilities $P(c_t = s) = q_s(t)$ with $s \in \{0, S, D, L\}$, a differential equation system is created like it was explained in Chapter 2.1.4. The differential equation system (DES) looks like this:

$$\begin{aligned}
\frac{d}{dt}q_0(t) &= -\lambda_S q_0(t) - \lambda_D q_0(t) - \lambda_L q_0(t) \\
\frac{d}{dt}q_S(t) &= q_0(t)\lambda_S \\
\frac{d}{dt}q_D(t) &= q_0(t)\lambda_D \\
\frac{d}{dt}q_L(t) &= q_0(t)\lambda_L
\end{aligned}$$

As the Markov chain structure is used for every component, it is not necessary to create the MCs for all the components. It is sufficient to know the three failure rates and to insert them into the DES. To solve the DES, an already implemented package has been used [2]. Instead of this very simple model, more complicated Markov chains could be used, for example a complete Markov chain with four states (i.e. a Markov chain with transitions in both direction between each pair of states, like in a complete graph) or a Markov chain with macro states. It is even possible to use non-Markovian models, for example models with transition rates based on the Weibull-distribution. The only important thing is that there must be a method to gain the probabilities for the four different states 0, S, D and L.

Chapter 7

Case Studies

This chapter describes several case studies for which the IFD-model was used. It will demonstrate the strength and the limits of IFDs. Section 7.1 explains a novel approach to use existing redundancy in a system, leading to simpler and cheaper system design while still having a high grade of redundancy. Section 7.2 describes the safety functions of a control system for a chemical reactor. Both systems include components with multiple failure modes and complex control loops making them perfect examples for explaining and presenting the IFD-model.
In this chapter, we will not note down the expressions for all the blocks and nodes of the presented IFDs. Instead, the complete models including all expressions and definition of failure states can be found in the appendix.

7.1 Remote Redundancy

This section deals with the principle of remote redundancy presented in [27] and its analysis using the IFD-model. This novel approach will be applied to a steer-by-wire system. Both a conservative architecture and an architecture using the remote redundancy approach will be presented and analysed, followed by a comparison of the different results.

7.1.1 The Basic System Architecture

For a steer-by wire system, the following components are necessary:

- An electronic control unit *ECU*
- A bus *B*
- An electrical motor *M*

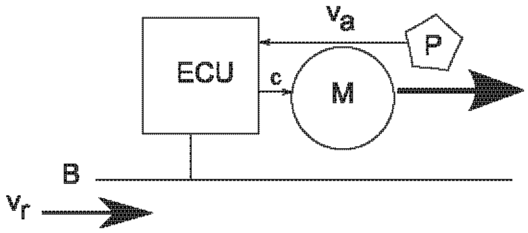

Figure 7.1: The basic steer-by-wire-system

- A position sensor P

This control loop works as follows: The reference value v_r is communicated to the ECU via the bus B. The actual value v_a is read from the sensor P. ECU calculates the difference v_r v_a and sends the correction value c to the motor M which is controlled by an electrical bridge (not shown). Figure 7.1 shows such a system. In this configuration, it is not fault tolerant, though. Each component can fail in different ways, and a failure of one component would lead to an overall system failure.

7.1.2 The Conservative Fault-Tolerant Architecture

In order to make this control loop fault-tolerant, its components have to be replicated. The motor and the bus are duplicated, the ECU and the position sensor triplicated. Additional electric links are added, besides two new rotation sensors and four output stages are added. Furthermore, a differential gear is needed so that both motors can control the steering mechanism. The whole architecture is depicted in Figure 7.2.

All components but the differential gear are redundant and thus all single component failures can be tolerated. The target value is sent via a double bus to ECU_1, ECU_2 and ECU_3, all of which calculate c. A triplication of the ECUs is necessary in order to be able to decide which ECU is faulty in case that their outputs are different. The actuator is duplicated only (M_1 and M_2), because it can be assumed that a faulty actuator is detected using the rotation sensors R_1 and R_2, and that there is sufficient time for passivation due to inertia. M_1 is controlled by ECU_1, and M_2 by ECU_2. Moreover, M_1 is passivated when ECU_2 and ECU_3 simultaneously stop the energy supply via their output stages O_{21} and O_{31}, respectively. Symmetrically, M_2 is passivated when ECU_3 and ECU_1 simultaneously stop the energy supply via O_{12} and O_{32}. If one of the ECUs fails and unjustifiably decides for passivation, the respective other ECU still preserves the provision of energy. The two actuators are coupled by a differential gear in order to tolerate blocking

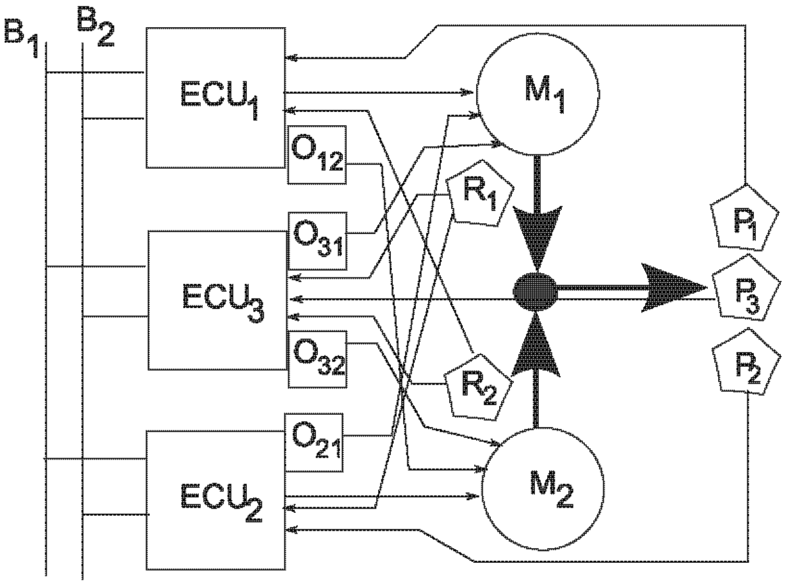

Figure 7.2: The conservative fault-tolerant steer-by-wire-system

motors. In theory, the differential gear or any other means to combine the output of both actuators represents a single point of failure. In practice this problem can be alleviated by using a very reliable differential gear. In many applications, the system under control itself is used to merge the output of the actors. For example, the actuators of an aeroplane might be two independent flaps which together have an influence on the plane's heading. As the implementation of the gear is highly application specific, it is outside the scope of this case study and it is assumed that no failures can occur in this component. Finally, the current position caused by the movement of the actuators is measured by an individual position sensor P_i for each ECU in order to close the control loop.

7.1.3 The Remote Redundancy Architecture

ECU_3 does not really give control instructions to the motors, its only duty is to monitor both and to passivate them if necessary. This task could be done by any ECU in the system, so it is possible to replace ECU_3 with an already existing ECU in the global system, not necessarily physical close to the steer-by-wire control system. Besides, electronic signatures will be used for the communication between all the components. This signature does not have to be very strong to resist an attack, it is only used to detect distorted messages. Using these measures, the architecture can be simplified a lot as can be seen in Figure 7.3. Similar to the system using the traditional ap-

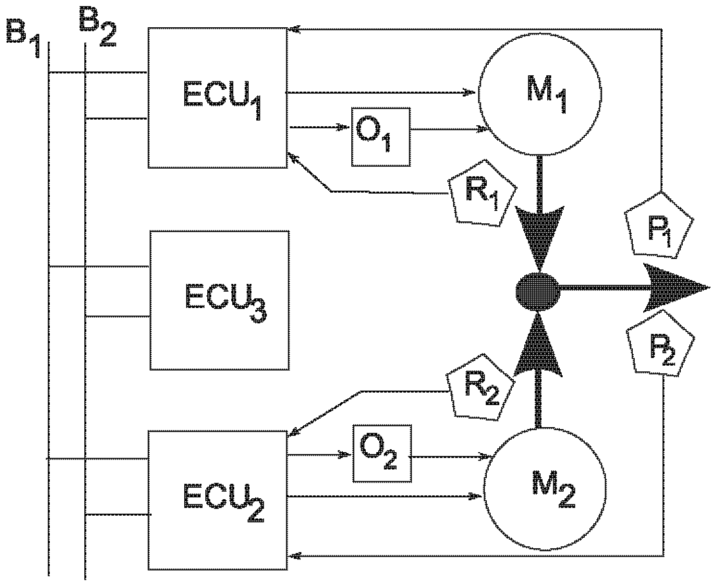

Figure 7.3: The steer-by-wire-system using remote redundancy

proach, it comprises three ECUs interconnected by a double bus system. Two of the nodes feature an actuator M, an output stage O for passivating this actuator, a rotational sensor R (for diagnosis) and a position sensor P (to close the control loop) which is attached to the outgoing shaft of a differential (that couples the rotation of both actuators). The third node, however, is not connected to any peripherals and only connected with the other ECUs via the network, thus representing *remote redundancy* as it can be placed anywhere in the network as long as communication latency requirements are met. The corresponding control software can share resources with other applications. The signals of both position sensors are cross-checked by each node with both rotational sensors, so that any wrong result of a single faulty sensor (rotational or positional) can be masked out. Consequently, a third position sensor is superfluous.

ECU_1 relays the information of the attached rotational sensor R_1 to ECU_2 and ECU_3 via the bus. It also forwards passivation signals from both ECU_2 and ECU_3 to the attached output stage O_1. Thus, in the case of an error, ECU_2 and ECU_3 jointly passivate actuator M_1 while ECU_1 and ECU_3 jointly passivate actuator M_2. Please be reminded that the forwarding of sensor values and passivation signals through "foreign" ECUs is only possible because signatures are being used, which would indicate a potential corruption of the relayed information.

Overall, this system has fewer components and no cross-wiring issues. But even after such a large reduction of the complexity, there are no single point of failures as shown in [27]. Still an important question remains: How does this reduction of redundancy influences the system safety? To answer this question, we made a qualitative and quantified analysis of both architectures.

7.1.4 The IFD-Models for the Two Architectures

Before analysing the architectures, it is necessary to define the possible component failures. We assume the following failure modes:

Component	Failure Mode S	Failure Mode D
Sensors	No result	Wrong result
ECUs	Crash Failure	Byzantine Failure
Output Stages	Spurious passivation	No passivation
Motors	Blocking or insufficient rotation	none
Communication	Loss or altering of signals	none

Note that there is no dangerous failure mode for the communication system as alternations of messages can be detected by signatures, so that these are handled the same way as lost messages.
With these assumptions, IFDs for both models can be created. We begin with the IFD for the traditional architecture.

The traditional architecture

At first, basic components have to be defined. The following components - named in small letters to distinguish them from the blocks - are included in the system and represented in the IFD: The motors m_1 and m_2, the output stages o_{12}, o_{21}, o_{31} and o_{32}, the ECUs ecu_1, ecu_2 and ecu_3, the buses bus_1 and bus_2 and the sensors p_1, p_2, p_3, r_1 and r_2.
In this system, ecu_1 and ecu_2 fulfil two different tasks: They are responsible for controlling m_1 resp. m_2 if there are no faults in the system. Besides this task they have to, together with ecu_3, monitor the system and detect failures. In case of an failure the ECUs have to shut down the faulty motor. For modelling purposes, it is better to distinguish between these two tasks. First, a model only for the controlling duty is presented, afterwards we will add the monitoring task.
Each ECU needs the reference value v_r which will be delivered via one of the buses and the actual value v_a which can received from the own positioning sensor. So the blocks Bus representing the two buses and P_i representing the position sensor i will be connected to the block ECU_i representing the ECU

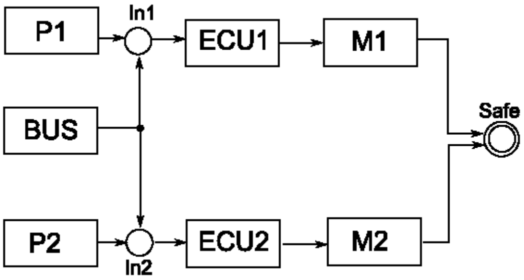

Figure 7.4: The IFD for the control system only in the traditional steer-by-wire-system

i via the decision node in_i. The control signal of the ECUs is forwarded to the respective motors, included as blocks M_1 and M_2. If none of the motors is turning in the wrong direction, and at least one of them is turning in the right direction, the steer-by-wire-system is working properly. Otherwise, a dangerous failure will occur. So we connect the two blocks M_1 and M_2 with the final decision node $Safe$ which defines the state of the whole system. The IFD for this simplified system is shown in Figure 7.4. Now it is necessary to add the monitoring and fault handling duties to the model. These tasks are added as own functional blocks to the model: ECU_{1S}, ECU_{2S} and ECU_{3S}. For their monitoring task the ECUs need not only v_r from one of the buses and v_a from their positioning sensor, but also the measurements of the appropriate rotation sensors. So, Bus, P_1 and R_2 are linked to the decision node in_{1S} leading to ECU_{1S}; Bus, P_2 and R_1 are connected with the decision node in_{2S} leading to ECU_{2S} and Bus, P_3, R_1 and R_2 are linked with the decision node in_{3S} leading to ECU_{3S}. The outgoing edges from the blocks ECU_{iS} are connected with the appropriate blocks for the output stages. The final step is to connect the output stages with the motors they can switch off. So the decision nodes inM_1 and inM_2 are introduced, which are put between the ECU_i- and M_i blocks. The IFD for the whole steer-by-wire-system using the traditional approach is shown in Figure 7.5.

After defining the high level model, the local expressions have to be chosen. In this example, they are quite simple. The block ECU_1 is chosen as example:

$S : (input(S) \wedge (ecu_1 = 0)) \vee (ecu_1 = S)$

$D : (input(D) \wedge (ecu_1 = 0)) \vee (ecu_1 = D)$

Decision nodes are also defined in an uncomplicated way. The final node $Safe$ looks like this:

$D : (M1 = D) \vee (M2 = D) \vee ((M1 = S) \wedge (M2 = S))$

The global steer-by-wire system will fail if either one of the motors turn in

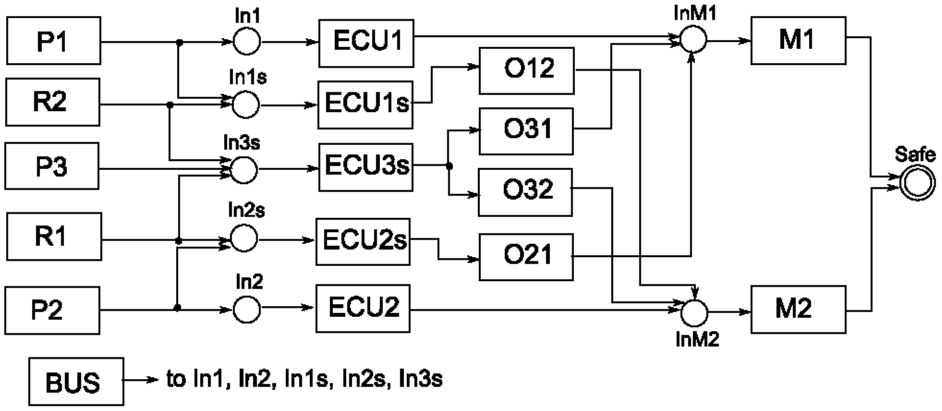

Figure 7.5: The complete IFD for the traditional steer-by-wire-system

the wrong direction, or if both motors do not turn at all. The expressions of the other blocks can be found in the appendix.

The remote redundancy architecture

Analogously, a model for the remote redundancy system has to be created. In this example, the ECUs do have a third task: They also have to forward measured values from the sensors to the other ECUs and activation or passivation signals from other ECUs to the output stage, e.i. that they are part of the communication channel. It is necessary to keep this in mind for modelling the system.

The components are defined analogously like in the conservative approach: The motors m_1 and m_2, the output stages o_1 and o_2, the ECUs ecu_1, ecu_2 and ecu_3, the buses bus_1 and bus_2 and the sensors p_1, p_2, r_1 and r_2.

In the remote architecture, the ECUs get the reference value v_r from both buses. Besides, all sensor values are sent to the three ECUs, either directly or via the buses. To represent this in the IFD, we create blocks for each Sensor and each ECU. Also, two blocks $COM1_R$ and $COM2_R$ are introduced. They represent the communication channel from the sensors r_i and p_i to the "foreign" ECUs, containing the buses and ecu_i. The blocks Pi and Ri for the according sensors are linked to the decision node $Seni$ which is connected to $COMi_R$. Both $COM1_R$ and $COM2_R$ are linked to the decision nodes In which is connected to the three ECU-blocks. In decides if the ECUs will get valid informations or not.

The control information for the motors is generated in ecu_1 and ecu_2 is forwarded to the motors. Before it can reach them, it is necessary to check if they have been passivated, though. So the other two ECU-blocks send their activation or passivation signal through the communication channel repre-

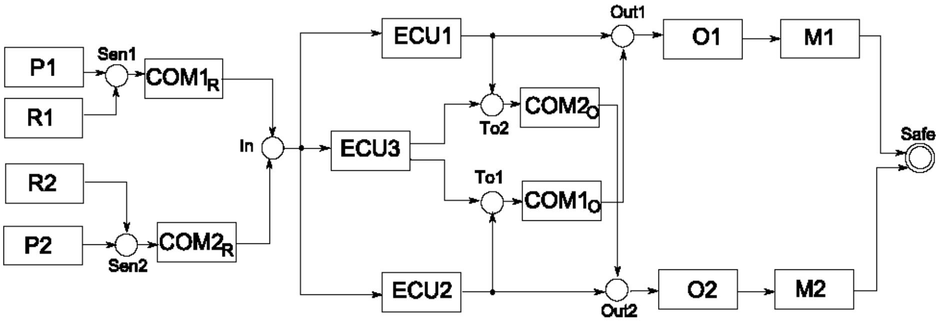

Figure 7.6: The IFD for the remote redundancy architecture

sented by the blocks $COM1_O$ and $COM2_O$ to the according output stages. The decision nodes $To1$ and $To2$ decide if both ECUs have made the same choice. The decision nodes $Out1$ and $Out2$ check if the output stages have to shut down the motors, taking in account the states of the signals from ECU_i and $COMi_O$. This result is forwarded to $O1$ and $O2$. The blocks $O1$ and $O2$ check, if the passivation or activation will work properly. The state of the outgoing state of the output stage blocks is further forwarded to the appropriate motor blocks. Afterwards, the final decision block $Safe$ estimates the global system state. The IFD for the whole model is shown in C.2.

At first, this structure seems to be slightly wrong. In the model, the sensors have always to go over the buses and one ECU while in reality this is not always necessary. Besides, the reference value gained via the buses seems to be omitted. But in fact these are no real problems, by defining the communication blocks the right way both can be handled. The failure of both buses will cause a failure for both communication blocks, finally leading to a dangerous failure of the whole system. In case of the failure of ecu_1 or ecu_2, it does not really matter what kind of input the block ECU_1 resp. ECU_2 will get. It will fail anyway. So it is possible to assume that ecu_1 resp. ecu_2 will never get the information of the sensors, just like the other ECUs. It will not change the resulting BDD. Sometimes it is possible to reduced the size of the IFD significantly by using knowledge about the modelling process.

Most of the local expressions are quite similar to the ones in the IFD for the traditional approach. One exception are the communication blocks. For example the expressions for $COM1_R$ are defined like this:

S: $(input(S) \vee (ecu_1 = S) \vee ((bus_1 = S) \wedge (bus_2 = S))$
D: $(input(D) \wedge (bus_1 = 0) \wedge (ecu_1 = 0)) \vee (input(D) \wedge (bus_2 = 0) \wedge (ecu_1 = 0))$

The block is in state S if communication is not possible any more. This can happen if either there is no incoming signal ($input(S)$), both buses fail ($(bus_1 = S) \wedge (bus_2 = S)$) or if the ECU fails and stops any messages forwarded through it ($ecu_1 = S$). The block switches to the state D if wrong messages are transported undetected through the communication channel. This can only happen if a faulty message like a wrong value from a sensor is sent through the communication channel. Alterings of messages in the ECU or in the bus are not included, as the messages are protected against this kind of failure with cyclic redundancy check (CRC, [64]) and signatures. Such an altering can be handled the same way like a loss of a message.
The expressions of the other blocks are given in the appendix.

7.1.5 Evaluation of the Models

After creating the models, they have to be evaluated. Two different kind of analyses were made: A qualitative and a quantitative evaluation.

The qualitative evaluation

To evaluate the models qualitatively, the possible paths through the BDDs have to be evaluated. Each path to the one-node represents a possible failure scenario. In such a path, some components can be in a faulty state. This means, that a failure of these components will lead to a failure of the overall system. By such a analysis, it is possible to show that in both models no single points of failure exist. But there are differences regarding dangerous two-component failures. For the traditional approach, there are five dangerous two-component failures:

- ecu_1 and ecu_2
- ecu_1 and ecu_3
- ecu_2 and ecu_3
- bus_1 and bus_2
- m_1 and m_2

As it is uses less redundancy, the remote architecture has more such dangerous combinations. Besides the five combinations valid for the traditional approach, the following pairs of components may not fail both:

- o_2 and m_1
- ecu_1 and m_2

- ecu_2 and m_1
- o_1 and m_2
- ecu_1 and o_2
- ecu_2 and o_1
- o_1 and o_2

Mainly the output stages lead to new dangerous combinations, as only two of them are used instead of four in the traditional approach. But to estimate the real impact of these new combinations, a quantified evaluation is necessary.

The quantified evaluation

For the quantified evaluation it was assumed that all component failures can be modelled with an exponential distribution. The failure rates were taken from the Military Handbook HDBK-217F [3]:

- Motors: 1,30E-05 h^{-1}
- Sensors: 2,00E-09 h^{-1}
- Buses: 2,00E-09 h^{-1}
- Output stage: 5,00E-08 h^{-1}
- ECUs: 5,00E-07 h^{-1}

Using these values and our models, the following unreliabilities could be calculated:

t/h	50000	100000	150000	200000
$U_{Traditional}$	8,30E-03	1,07E-01	3,18E-01	6,22E-01
U_{Remote}	1,39E-02	1,33E-01	3,59E-01	6,58E-01

The values of the novel approach are slightly higher, as expected. But most times they are in the same magnitude. The reason for this behaviour is that the failure of both motors has the most impact on the failure rate for both architectures, as the failure rate of the single motors is much higher than the failure rates of the other components. The additional dangerous combinations for the remote architecture are all combinations which are very unlikely. The main weak point of the traditional architecture remains unaltered. Only redundant components which could prevent accidents very rarely were removed.

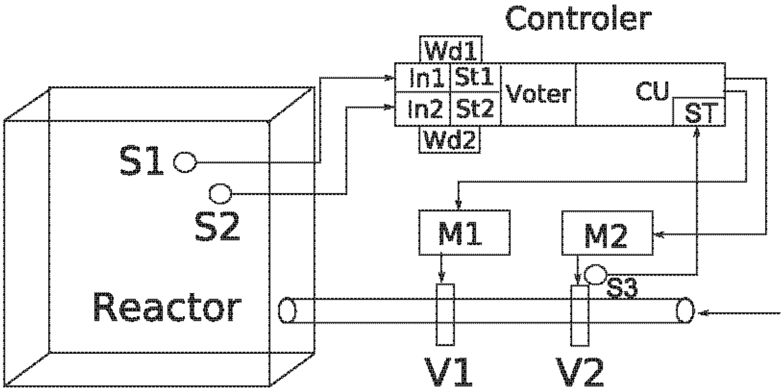

Figure 7.7: The architecture of the chemical reactor

7.2 Chemical Reactor

The next case study is an emergency stop system of a chemical reactor, based on the system already handled and modelled in [35] and presented in [46].This system should stop the reaction if the temperature in the reactor is getting too high by stopping the inflow of the chemicals. A new IFD-model including the modifications presented in this work is shown. This model is compared to the original model from [35] to demonstrate the differences between the original IFD-model and the modified version.

7.2.1 The System Architecture

The sensors $S1$ and $S2$ measure the current temperature of the chemicals in the reactor and communicate their results to the controller. The controller reads this result via his inputs $In1$ and $In2$ and will store these values for synchronization in different parts of 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 too late, the watchdog will pass a default value to the voter. Afterwards, the voter decides, if there is a dangerous situation. A shutdown will be ordered if at least one sensor is reporting a temperature exceeding the limit. If after the voting process the control unit CU decides to shut down the system, this information is forwarded 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. We assume that the communication and the memory in this system is protected by CRC, so faulty alterings can be detected.
To increase the overall safety of the system, the valve $V2$ is tested regularly in order to detect if the valve can close by a sensor $S3$. If such a test detects

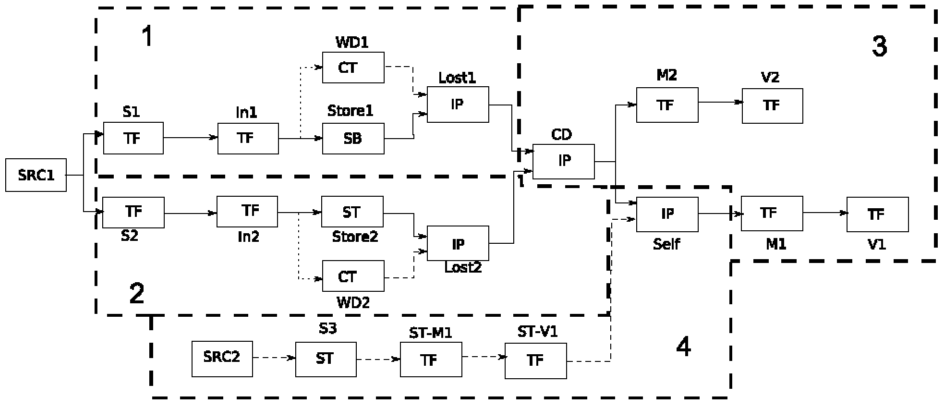

Figure 7.8: The old version of the IFD of the chemical reactor

a blockade of the valve $V2$ this result is forwarded to a part of the control unit (ST) which orders to close $V1$ immediately.

For the whole system, there are two possible kinds of failures in general. Either the emergency system is not available (dangerous failure), i.e. no valve will be closed if necessary, or it shuts down the system in a safe state leading to a unnecessary unavailability of the whole reactor (spurious shutdown), i.e. one of the valves is closed in a safe situation.

A failure of the whole emergency system can be caused by several different failures of its components. The sensors $S1$ and $S2$ can either measure a value which is too high (S), too low (D), or return no value at all (L). The Sensor $S3$ can detect a non existing blockade of $V2$ (S) or not detect an existing blockade (D). The input modules can either lose the data of the sensors (L) or change it to another value (S, D). In the memory the stored data can be distorted by a bitflip, leading to a detectable fault (S). 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 (L). 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 motors can fail to start (D). Finally, the valves can be stuck in an open (D) or closed (S) position.

7.2.2 The IFD-Models

In this subsection, two IFD-models are presented. At first, the original IFD-model presented in [35] is used. Afterwards, the modifications of Chapter 4 are applied.

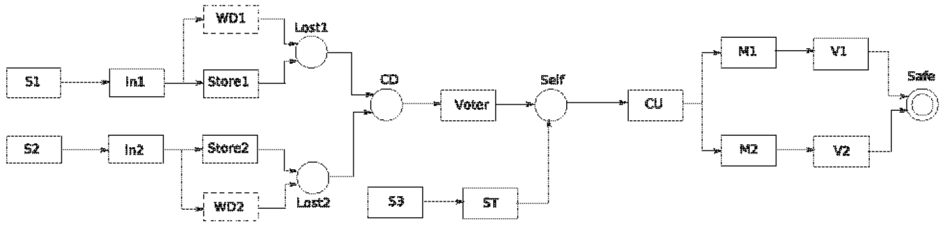

Figure 7.9: The modified version of the IFD of the chemical reactor

The original model:

The original IFD using the old version of the model is depicted in Figure 7.8. Basically, there are four different parts of the model. The parts one and two represent the sensors $S1$ and $S2$ and the way of their values using the input modules *In1* resp. *In2* and the memory represented by the blocks *Store1* and *Store2* to the voter. Also the watchdogs are included as their own blocks. The IP-blocks *Lost1* and *Lost2* describe watchdogs send their results to the voter if necessary. The voter represented by an IP-block unifies these two sensor values and transforms them to an order for steering the valves. Part three of the model represents the forwarding of these orders from the voter over the motors *M1* and *M2* to the valves represented by the blocks *V1* and *V2*. The fourth part of the model is the self-test functionality. It describes the self-tests and the closing of valve $V1$ using the motor $M1$ if necessary. The block *ST* describes the selftest itself, the block *ST-M1* the motor and the block *ST-V1* the valve to close. The IP-block *Self* unifies the information flow for closing Valve 1 by the selftest-unit and the regular control unit.

For each block, an automaton is defined. These automata describe lists like in section 3.2. For example the automaton of block *In1* defines the following list:

$$L_a(i) = Init(True).d(x5,0).a.Fin(True); Init(False).d(x5,S).Fin(True)$$
$$L_b(i) = Init(True).bf(e2).Fin(False); Init(False).d(x5,0).b.Fin(False)$$
$$L_c(i) = Init(True).d(x5,I); Init(False).d(x5,I)$$

$x5$ represents the input module *In1*, $e2$ represents the bus. The whole low level model of this example can be found in the appendix of [35].

The modified model: The modifications are applied in two steps. First, the high level model is altered so that it fits to the new methodology. Afterwards the lists for the low level model are changed to the appropriate expressions.

Large parts of the high level model can be created by simply changing the type of blocks. TF-, SB- and ST-blocks are changed to standard blocks, SRC1- and SRC2-blocks are replaced by source blocks and IP-blocks are substituted by DEC-nodes. Still, at some points the topology of the model has to be changed.

In the formalised version of the model, DEC-nodes can not describe any failures unlike IP-blocks. The DEC-node *CD* replacing the IP-block in the original model only defines which control decision should be taken by the voter, but does not include any possible hardware failures of the Voter or the control unit. So it is necessary to add the standard blocks *Voter* and *CU*. Also a new final DEC-node *Safe* is added for defining the global system state. Furthermore, the selftest unit was simplified. The block *ST*, which takes it decision based on the result of the sensor represented in the block *S3*. It forwards its results to the new DEC-node *Self*, which takes both the decision of the voter and the result of the selftest into account for deciding if the valves have to be closed or not.

The low level model can be changed in a similar way. Each event in the original model can be translated to an expression in the altered model. For example the lists of the block *In1* presented in the previous paragraph is changed the following way:

At first, all *other*- and *final*-events are removed as they do not carry any information. Afterwards, initial events are replaced with Input-expressions, bitflips and transient failures are transformed to general failure expressions. In the case of bitflips only the failure mode S will occur, the probability for the other failure modes is 0. Furthermore, the components are renamed to increase intuitivity. $x5$ is called $in1$ and $e2$ is called bus.This leads to the following three expressions:

$S : (Input(S) \wedge (in1 = 0)) \vee (Input(D) \wedge (in1 = S))$
$D : (Input(D) \wedge (bus = S)) \vee (Input(S) \wedge (in1 = D))$
$L : (Input(S) \wedge (in1 = L)) \vee (Input(D) \wedge (in1 = L))$

Note that these expressions can be simplified even further. For the expression L it is not really important in which state the signal is before entering the block. If $in1$ is in state L, the block will be in state L. So the expression for L can be simplified to:

$L : (in1 = L)$

For DEC-nodes, more work is necessary. As these are total different from IP-blocks, their expressions have to be defined taking into account the logic of the architecture. For example, the expressions for the final node *Safe* look like this:

$S : (V1 = S) \vee (V2 = S)$
$D : (V1 = D) \wedge (V2 = D)$
$L : false$

The system will fail dangerously if both valves can not be closed. A spurious trip will occur if at least one of the valves will close without demand. The failure mode L is not defined for the global system.
The rest of the expressions of the new model can be found in the appendix.

7.2.3 Comparison of the Two Models

Both models describe the architecture of the reactor regarding its safety functions in a very detailed way. But there is a huge difference between the two models: The original model contains much more types of events and blocks than the modified one. Still both models are equally powerful, the modified model includes all failure modes and functionalities like the original model.
While for the original model a huge Markov Chain has to be created which can only be evaluated with a lot of computing power, the modified model is easy enough to be transformed in an easy solvable BDD.
Besides the faster computation, the BDD has also another advantage: It is possible to extract the most probable failure scenarios including all single points of failure for both failure modes easily without creating all possible failure scenarios explicitly by extracting the most probable paths through the BDD. For the failure mode S, a spurious trip of the following components will lead to an overall spurious trip: $V1$, $V2$, CU, $S1$, $S2$, $S3$, ST and $Voter$. The dangerous failure of CU or $Voter$ will cause a dangerous failure of the whole system. Such lists of the most probable failure scenarios are much more useful than very immense lists with all possible, but a lot of very improbable failure scenarios if a system should be evaluated qualitatively.

Chapter 8

Evaluation

The algorithms presented in this dissertation are evaluated in two different ways. At first, a theoretical complexity analysis is made. Afterwards, the performance of the algorithms is tested with measurements taken from the implemented software. Both approaches demonstrate the strengths and weaknesses of the developed methods.

8.1 Complexity Analysis

Here the theoretical complexity analysis is discussed. It consists of three parts. At first, the analysis is made for simple serial systems like in Chapter 5.2. Afterwards the analysis is extended for general serial systems (Chapter 5.3) and IFDs with DEC-nodes (Chapter 5.4).

8.1.1 Simple Serial Systems

In many cases, BDDs are very efficient for representing Boolean expressions. Still it is possible to find worst case examples with n Boolean variables with a space complexity of $O(2^n)$ for any variable ordering[20][42]. So basically the overall space complexity of the algorithms should be $O(2^{3n})$ for an IFD with n components, as each component is represented by three Boolean variables. A more precise estimation is possible though. At first, three variables based on one component always occur together in a fixed structure consisting out of at most three, not eight nodes. This means that the complexity is reduced to $O(2^n)$. Furthermore, for systems in which no component occurs in the expressions of two different blocks, the complexity can be estimated even more precisely. In Chapter 5.2 the algorithm creates a BDD for one block first. This BDD has at most three non-trivial leaves (Input(S), Input(D) and

Input(S) $\wedge$ Input(D)), which are replaced by the BDDs of the predecessor block. These three BDDs share equivalent nodes and have themselves at most three non-trivial leaves. So, while the BDDs for the blocks can grow exponentially, they will always shrink back to three nodes at the end of the block. Be m the number of blocks and k the maximum number of components in one block. Then the overall complexity of the BDD is $O(m \cdot 2^k)$. If k is quite small, which is a reasonable assumption - normally modellers try to minimise the size of low level models - the actual complexity is much better than the originally expected one.

8.1.2 General Serial Systems

If components can occur in several blocks, the analysis is more complicated. One major difference to the simple case is that a BDD describing the expression of one block can have more than three non-trivial leaves. Each leaf contains additional information regarding the variables occurring in other blocks, and this information is used in the equivalence check. Each component already set in the expressions used in the pointer arrays quadruples the amount of possible leaves.

As illustrative example we take a serial IFD with four blocks A, B, C and D. A component c_1 occurs in B and D, a component c_2 occurs in B and C. Furthermore there are other components, but these are not interesting for now.

If the BDD for the final block D is created, it can contain up to 12 non-trivial leafs. The three possible input values can be combined with the four possible settings for c_1, which have to be forwarded to the local BDD for block B. So, up to twelve local BDDs for block C can be created. Three groups of four local BDDs are practically identical, they differ only in the value for c_1 which is needed for block B. Besides that, the BDDs also have to forward the value of c_2 to B which increases the number of possible non-trivial leaves to 48.

In the local BDDs for block B, all the different settings of c_1 and c_2 influence the local expression which is already simplified a bit. These settings do not influence the block A though, as it does neither contain c_1 nor c_2. So the block B has only at most three non-trivial leafs like in the standard case.

With these considerations the complexity of the BDD-creation can be estimated. Be l the number of components occurring in more than one block. Then the complexity of the BDD-creation for a generalised serial IFD is $O(4^l \cdot m \cdot 2^k)$ which is much worse than the simple case if l is large.

Unluckily, multiple occurrences of components in one model happen quite often. For example the components *bus*1 and *bus*2 in the IFD-model for remote redundancy occur in four different, the components *ecu*1 and *ecu*2 in three different blocks. Still the effect is normally not so drastic as the theoretical complexity implies. Most of the possible non-trivial leafs never occur in practical systems, so even the IFD-model for the remote redundancy architecture with four components in multiple blocks can be solved very fast.

8.1.3 Non-Serial Systems

For non-serial models, i.e. models which contain DEC-nodes, BEDs have to be analysed. BEDs are based on the apply algorithm. $apply(op, G, H)$ for a Boolean operator op and two BDDs G and H has the complexity $O(|G| \cdot |H|)$[20]. The creation of BEDs has the same complexity as the apply algorithm [11].

In [11] the authors show that the worst case complexity of BEDs has to be exponential as only polynomial sized BEDs for n variables can not be used to describe an exponential number of Boolean expressions with n variables. Luckily almost all practical examples can be expressed with polynomial sized diagrams. In fact it is extremely difficult to find an example for an exponential sized BED. So, while the theoretical complexity is exponential, practically the usage of the BED-technique should lead to fairly compact and handy results.

8.2 Measurements

For evaluating the presented algorithms, several things can be measured. It is possible to measure the solution time and memory usage or just to count the number of nodes in the BDD.

The solution time and the memory usage depends on multiple factors, for example: The architecture of the platform on which the test were made, the optimisation of the implementation (especially regarding the architecture), the chosen programming language or the operating system. On the other hand, the number of nodes only depends on the original algorithm itself and chosen heuristics. The architecture on which the tests are executed does not influence the size of the BDD.

As in this work the proposed algorithm itself should be evaluated, not its implementation, the number of BDD-nodes is chosen as measure.

This section is divided in two parts. First, the practical examples from Chapter 7.1 and Chapter 7.2 are analysed. These examples show that the proposed method can deal with complex systems and several failure modes efficiently. Afterwards, some artificial, easily scalable examples are evaluated. These examples are used to compare our method with standard BDD-approaches.

8.2.1 Case Studies

Three different models were presented for the case studies: an IFD for a chemical reactor (16 components, 21 blocks and nodes), an IFD for a classical steer-by-wire system (16 components, 25 blocks and nodes) and the model for the remote redundancy architecture (13 components, 23 blocks and nodes). The number of nodes in the respective BDDs are given in the following table:

Example	Chemical Reactor	Classical Red.	Remote Red.
BDD-Nodes	146	19261	3582

Although all these models have approximately the same number of components and blocks, there are large differences between the sizes of the BDDs. To find an explanation for this, the models itself have to be investigated more closely.

In the model for the chemical reactor, no component occurs in more than one block, and the expressions for the blocks are very small. This will lead to a very efficient representation of the serial parts of the IFD as already demonstrated in the present chapter. Furthermore, the IFD contains only five DEC-nodes, each of them with only two inputs. As BEDs can lead to a complexity worse than for simple serial diagrams it is good that only a few of them are in the model. Besides that, the expressions of the DEC-nodes are fairly simple, so that the resulting original BEDs contain only one operator node each. As each operator node represents one apply-operation with the complexity $O(|G| \cdot |H|)$, keeping the number of these down leads to smaller BDDs. Taking into account these facts it is not so surprising that the whole BDD is quite small, even if the explicit representation in [35] is very large.

The model for the classical redundant architecture contains the same amount of components and only four blocks more than the IFD for the chemical reactor. Still its BDD is much larger. The reason for this is the topology of its IFD. It contains more DEC-nodes, from which four have three incoming edges. The DEC-node In3s (see 7.5) has even four. This leads to more operator nodes in the BDD-creation process. Furthermore, it is a very "broad" IFD. While it has more blocks and nodes than both the IFDs for the chemical reactor or the remote redundancy architecture, a path from any source block to the final node can contain at most seven blocks and nodes. For the other examples, up to eleven blocks and nodes are possible for such a path. For "broad" IFDs, the proposed algorithms do not work as well as for IFDs with longer serial sub-diagrams.

The BDD of the model for the remote redundancy architecture is smaller than the BDD for the classical redundancy, but larger than the BDD for the chemical reactor. It has fewer components than the other examples, so a

smaller size can be expected. Its major difference is that it is the only model in which several components occur in multiple blocks. Furthermore, it contains more DEC-nodes than the IFD of the chemical reactor. So it contains several factors which explain the larger size of its BDD compared to the BDD of the chemical reactor. Still, the IFD is not so broad like the IFD of the classical architecture, and its DEC-node-expressions are much simpler. So the size of the BDD is significantly smaller than for the BDD of the classical architecture.
Using these examples, the following points have to be respected if an IFD for a large system should be solvable:

- Avoid too many DEC-nodes.
- Avoid DEC-nodes with many incoming edges.
- Prefer a "slim" topology, avoid "broad" topologies.
- Avoid components occurring in multiple blocks.

Of course, in many cases it is not possible to follow all these points exactly. Still normally a certain degree of freedom exists for modelling systems. If it is used wisely, the IFDs can be modelled in a way to avoid too large BDDs which can not be handled anymore.

8.2.2 Scalable Models

IFDs can not only be used for modelling systems with several failure modes. Theoretically it is possible to model classical models which can be represented by Fault Trees or RBDs. It is not really optimal to do so as many of the features of the IFDs are not needed. The resulting model would only be unnecessary complicated. But by creating such models it is at least possible to compare the algorithms presented in this dissertation with classical BDD-approaches. For simple examples the algorithms should not have a worse complexity than standard BDD-solutions.
Two examples were chosen. One example called $SerPar(n, m)$ is serial system of n subsystems, in which each subsystem consists out of m parallel components $c_{i,j}$. The other example called $ParSer(n, m)$ is a parallel system of n serial subsystems. Each subsystem consists out of m serial components $c_{i,j}$. Figure 8.1 shows the RBDs for both systems.
Now it is necessary to create equivalent IFDs for these systems. In general, there are two possible designs. One possibility (Variant A) is to create a serial IFD with n blocks. For SerPar, each block i gets the following expression:

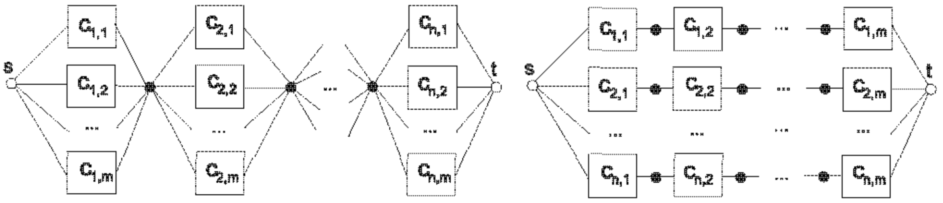

Figure 8.1: RBDs for $SerPar(n,m)$ (left) and $ParSer(n,m)$ (right)

$Input(S) \vee (c_{i,1} = S \wedge \cdots \wedge c_{i,m} = S)$

For ParSer, the expressions look like this:

$(Input(S) \wedge c_{i,1} = S) \vee \cdots \vee (Input(S) \wedge c_{i,m} = S)$

Note that we only need one failure mode, we could use the failure mode D instead with basically the same result.
The other possibility (Variant B) to describe such serial-parallel systems with an IFD is more complex. It is based on the high level model depicted in Figure 8.2.
The IFD consists of n DEC-nodes D_i and $n \cdot m$ non-DEC-blocks $B_{i,j}$. The

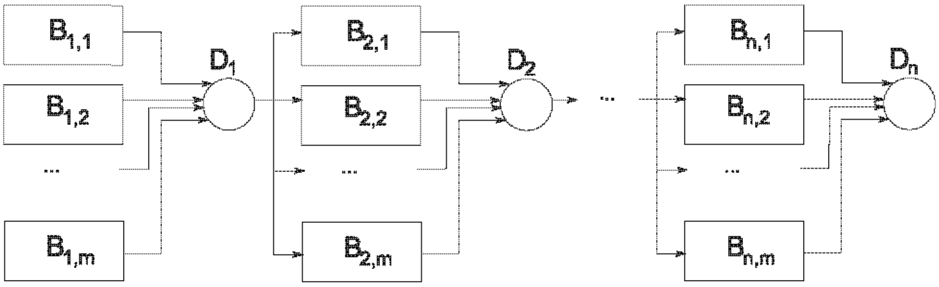

Figure 8.2: The IFD-high-level-model for SerPar and ParSer (Variant B)

expressions for the DEC-nodes are defined as follows:

$B_{i,1} = S \wedge \cdots \wedge B_{i,m} = S$ for SerPar
$B_{i,1} = S \vee \cdots \vee B_{i,m} = S$ for ParSer

The expressions for the blocks $B_{i,j}$ look like this:

$Input(S) \vee c_{i,j} = S$ for SerPar
$Input(S) \wedge c_{i,j} = S$ for ParSer

The two presented variants can be created automatically. In this work both variants are used, as we want to test both serial and non-serial IFDs. The following table shows the number of BDD-nodes needed for the example SerParA for $1 \leq n \leq 5$ and $1 \leq m \leq 5$:

		n				
	SerParA(m,n)	1	2	3	4	5
m	1	1	2	3	4	5
	2	7	9	11	15	19
	3	13	16	19	22	25
	4	19	23	27	31	35
	5	25	30	35	40	45

For this example, the number of BDD-nodes grows linearly regarding both n and m, leading to an overall complexity of $O(n \cdot m)$. A better complexity is not possible as at least each of the $n \cdot m$ variables in the original RBD has to be included in any representation. The next example is ParSerA:

		n				
	ParSerA(m,n)	1	2	3	4	5
m	1	1	7	13	19	25
	2	2	17	31	45	59
	3	3	26	48	70	92
	4	4	35	65	95	125
	5	5	44	82	120	158

For this example the BDD grows also linearly with m and n, but more nodes are needed than for the example SerParA. The reason for this is that in SerParA the reduction rules of ZBDDs can be applied more often than in ParSerA. Still the result is as expected. As both examples are serial IFDs without components occurring in multiple blocks, the complexity may only be linear regarding the number of blocks. And the local ZBDDs for the blocks can also grow only linear as the local expressions have a very simple structure which is favored by BDD-techniques.

For non-linear IFDs, the results are different. The next table shows the number of nodes of the example SerParB(m,n):

		n				
	SerParB(m,n)	1	2	3	4	5
m	1	1	7	13	19	28
	2	2	19	57	142	307
	3	3	31	105	290	716
	4	4	43	153	436	1267
	5	5	55	201	584	1976

In this example, the BDD-size grows linearly with m, but super-linearly with n. For the last variant, it is even worse:

		n				
	ParSerB(m,n)	1	2	3	4	5
m	1	1	2	3	6	7
	2	7	21	49	114	348
	3	13	45	125	340	1846
	4	19	69	196	1225	2409
	5	25	93	271	4436	10398

The size of the BDDs grows superlinearly for both n and m. The reason that the variants B are worse than the variants A is simple. The structure of the IFD for the variants A forced the BDD to adapt the variable order to the structure of the underlying RBD. In SerParA the parallel subsystems had to be analysed before the serial system was evaluated. In ParSerA, the serial subsystems were handled before the parallel global system. In both cases this leads to an optimal variable order.

For the variants B, the variable order is determined by a BFS-ordering of the blocks. For SerParB, the ordering is still optimal. But this is not enough for a linear growth of the BDD-size regarding n. The cause for the superlinearity is that for each block a local ZBDD is created. For a block $B_{i,j}$ it contains the variables for the components $c_{i,1}, c_{i,2}, ...c_{i,n}$. As we are using ZBDDs, each variable has to really appear as node unless it must be false. So the size of each local ZBDD is linear to n. As there are $n \cdot m$ different local ZBDDs, this leads to a complexity of $O(m \cdot n^2)$. In this case, ZBDDs are a disadvantage. ParSerB is even worse than SerParB. The reason for this behaviour is that the variable order based on our chosen heuristic is really bad. For an optimal sized BDD the variables of each serial subsystem should be decomposed directly afterwards. In this case the next variable is always from the next serial subsystem. This leads to the extreme explosion of the BDD-size.

These measurements showed that the BDDs of serial IFDs grow linearly not only in theory. Non-serial IFDs are worse. How large this effect is depends mainly on the structure of the IFD. In some cases, at least polynomial complexity is possible which is acceptable. In the worst case, exponential complexity can still happen. This is expected, though. It is impossible to create a BDD which only needs a polynomial amount of nodes for describing any possible Boolean expression, as there is an exponential number of possible permutations regarding the number of Boolean variables.

Part III

Conclusions

Chapter 9

Summary

Now it is time to summarise the work which has been presented in this dissertation. We distinguish four main parts of the work: The modifications of the IFD-model, the developed algorithms, the implementation and the case studies. The next sections will deal with them in more detail.

9.1 Modifications

The modifications were made with several targets in mind. At first, the IFD approach should be formalised so that it is easier to handle, especially for the BDD-creation. Furthermore, the intuitivity should be increased.

To formalise the model, the original low level model was changed a lot. Instead of automata defining characteristic lists for different failure modes expressions were introduced. Furthermore the six possible list elements in the original model were reduced to two different kinds of expressions in the new model. Also the high level model has been altered. Several types of blocks were unified, and IP-blocks were changed to DEC-nodes with completely different characteristics.

To increase intuitivity, several naming conventions were changed and standardised. For example, the failure modes S,D and L occur now in every layer of the model. Furthermore, the reduction of blocks and list elements in the original model makes it easier to understand, as less elements have to be handled.

Although the model has been altered a lot, it has not lost any of its important properties. It still can describe the behaviour of safety critical systems in a very compact and unique way. All old models can be easily transformed into the new format without losing any of its expression power. Overall, the formalisation was the first important step for the efficient evaluation of the

IFD-model.

9.2 Algorithms

The evaluation is done in multiple steps. First, the expressions of the blocks are transformed to Boolean expressions. This is necessary in order to use Binary decision diagrams. Afterwards, the gained local Boolean expressions are decomposed, beginning with the final block of the IFD. This way a BDD is created. Once a block is finished, the decomposition is continued with the expressions of the predecessor block. With this recursive creation process the properties of the IFD are used in an optimal way.
The algorithms support also components which occur in multiple blocks of the IFD, furthermore the constructed BDD can be used for a quantitative analysis, if the failure probabilities of the components are known. It can take into account the inherent dependencies in the model, i.e. that components are always in exactly one state.
This algorithm was evaluated theoretically and by measurements. In both ways it proved that it can be very efficient, although in some cases a combinatorial explosion still can occur. But for most "natural" examples, it works very well.

9.3 Implementation

The presented algorithm was implemented in Java to give a proof of concept. The implementation reads a file in a defined format, creates an internal representation of an IFD and transforms it into a BDD. This BDD is evaluated and returns the probabilities for the failure modes S and D of the modelled system.
The implementation could handle small, artificial examples as well as real life applications. It has no GUI yet, but it can be extended easily. Overall the implementation proved that the presented ideas can work in reality, too.

9.4 Case Studies

Finally two case studies were made. The first case study was a comparison between a classical redundancy structure of a steer-by-wire-system and a novel architecture called remote redundancy. With IFDs it was possible to show that the novel approach, in which much less redundancy was needed, still was almost as good as the classical system.

The second example was a chemical reactor, which demonstrated the differences between the original and the modified IFD-model. It proved that the new formalisms can express the same systems than the original approach. These case studies also demonstrated how IFD-models can be created, and that they are a practical alternative to other modelling tools. The models for both quite complex systems were very compact and could be evaluated in a few seconds on a standard computer.

Chapter 10

Outlook

The work done in this thesis can be continued in many different ways. In this chapter we show a few possibilities how to extend the achievements presented in this dissertation.

10.1 Performance Improvement

The current implementation is mainly a proof of concept. But as the used algorithms are quite close to original BDD-algorithms, it should be possible to use techniques to minimize memory usage for the implementation. This would enable the evaluation of larger models.

10.2 The Bertholon Model

Currently a student project is going on with the target to use the *Bertholon Model* instead of the exponential distribution for modelling the behaviour of the components. The Bertholon Model [21] is a combination of the exponential and the Weibull distribution. It can be approximated by dividing the mission time in several time slices and define a fitting exponential distribution for each. This is no exact solution, but upper and lower bounds can be determined. Furthermore, this approach allows for an analytical solution.

10.3 Further Generalisation

It is possible to generalise the IFD-model even further. Instead of three special failure modes an arbitrary number of failure modes can be allowed. Besides that, also ST-, SRC- and WD-blocks could be merged to one standard block. It is not so hard to define such a generalised model theoretical, also the algorithm can be adapted fairly easy. It is harder to implement and test such an algorithm, though.

10.4 Graphical User Interface

At the moment, the models have to be entered by writing a text-file. This is very difficult and error prone. Adding a GUI would make the modelling process much easier. Adding it should not be a large problem: A graphical model of the IFD can be easily created and transformed to the internal representation which is already used for the evaluation.

10.5 Dependencies

At the moment we assume that there are no dependencies between the components. But there are systems for this is not a valid assumption as there are effects like common cause failures or failure propagation. The IFD-model could be extended that such effects are included.
In order to evaluate such a model, a state based model for dependent components has to be created. The results of these models have to be combined in a valid way. In [47] we presented a method how to evaluate OpenSESAME-models with BDD-techniques. Instead of creating BDD-nodes for single components, nodes for sets of n interdependent components were created with 2^n outgoing edges, one for each possible permutation. The probability of each permutation can be calculated with a state based model.
This approach could be adapted for IFDs, too. So they could model even more systems in a very detailed way.

10.6 LARES

In [32] the authors present LARES, an interface for connecting different high level models with different state based models. These high level models may support dependencies, different repair behaviours, multiple failure modes or other stochastic dependencies. Instead of transforming a high level model directly into a state based model, it is transformed into the LARES-formalism. This formalism can be used to create different equivalent low level models like SPNs or Markov Chains. So for n supported high level models and m supported low level models LARES would need only $n + m$ transformations compared to $n \cdot m$ for direct transformations.

It would be possible to transform the IFD-model into LARES, which would immediately allow for using much more state based models for calculating the probabilities of component failures. Besides that, the IFD-model could be extended with dependencies fairly easily, as LARES will take care of the correct calculus.

Part IV

Appendix

Appendix A

Abbreviations

A	Steady State Availability
A(t)	Availability
BDD	Binary Decision Diagram
BDT	Binary Decision Tree
BED	Binary Expression Diagram
BFS	Breadth First Search
BNF	Backus Naur Form
CNF	Conjunctive Normal Form
CTMC	Continuous Time Markov Chain
D	Dangerous Failure Mode
DAG	Directed Acyclic Graph
DEC	Decision Nodes
DES	Differential Equation System
DNF	Disjunctive Normal Form
FT	Fault Tree
IFD	Information Flow Diagram
L	Loss of Information Failure Mode
MC	Marcov Chain
MTTF	Mean Time to Failure
MTTR	Mean Time to Repair
PFS(t)	Probability of a Spurious Trip
PFD(t)	Probability of Failure on Demand
R(t)	Reliability
RBD	Reliability Block Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
S	Safe Failure Mode
S(t)	Safety
SPN	Stochastic Petri Net
SRC	Source Blocks
ST	Standard Blocks
U(t)	Unreliability
WD	Watch Dog Blocks
ZBDD	Zero Suppressed Binary Decision Diagram
ZBED	Zero Suppressed Boolean Expression Diagram

Appendix B

Glossary

Apply Algorithm: Algorithm to create BDDs efficiently by applying reduction rules immediately.

Availability A(t): The probability that a component or system is working correctly at the time t.

Binary Decision Diagram (BDD): An efficient way to represent Boolean expressions by using the *Shannon Decomposition* and a binary DAG. Several kinds of BDDs exist like *Binary Expression Diagrams* (BED), *Reduced Ordered BDDs* (ROBDD) and *Zero Suppressed BDDs* (ZBDD).

Binary Decision Tree (BDT): A tree representing a Boolean expression by showing all possible decomposition paths. It grows exponentially with the number Boolean variables in the expression, but it can be reduced dramatically by using BDD-techniques.

Binary Expression Diagram (BED): An ROBDD extended with nodes for Boolean operators. A BED can be reduced to a standard ROBDD by applying special reduction rules.

Block: A functional entity in an IFD. It is described in more detail by either a finite automaton (in the original formalism) or by three local expressions S, D, and L (in the modified formalism). By convention blocks are named in capital letters. There are several types of blocks in the original (SRC1, SRC2, FT, TF, CT, ST, SB, IP) and in the modified formalism (SRC, ST, WD).

Boolean Formalisms: See *Combinatorial Formalisms*

Byzantine Failure: Occurs if a component or system gives faulty orders or results.

Combinatorial Formalisms: Modelling formalisms which can model complex systems. Afterwards, a Boolean expression representing the modelled system can be extracted and evaluated. The two most important combinatorial formalisms are Fault Trees and Reliability Block diagrams.

Conjunctive Normal Form (CNF): A Boolean expression x with the variables $x_{ij}, 1 \leq i \leq n, 1 \leq j \leq m$ in the following form:

$$\wedge_{i=1}^{n}(\vee_{j=1}^{m} x_{ij})$$

is in CNF. This means that a Boolean expression in CNF is a conjunction of disjunctive terms.

Crash Failure: Occurs if a component of system shuts down in an unspecified way.

Dangerous Failure: A failure which leads to a violation of the specified safety requirements of a component or system. This can lead to incidents endangering humans or the environment.

Disjunctive Normal Form (DNF): A Boolean expression x with the variables $x_{ij}, 1 \leq i \leq n, 1 \leq j \leq m$ in the following form:

$$\vee_{i=1}^{n}(\wedge_{j=1}^{m} x_{ij})$$

is in DNF. This means that a Boolean expression in DNF is a disjunction of conjunctive terms.

Exponential Distribution: A probability distribution with only one parameter, the failure rate $\lambda > 0$. Its probability density function is defined as follows:

$$f(t) = \lambda e^{-\lambda t}$$

Failure: A behaviour of a system which violates its specification.

Fault: A state of a system which violates its specification. Can lead to a failure if not tolerated.

Fault Tree: A *Combinatorial Formalism* to model the failure behaviour of systems as a tree with AND-gates, OR-gates and basic events.

Global Lists: Lists for the global system of an IFD which describe the possible scenarios so that a system will generate a spurious trip (list S) or a dangerous failure (list D).

Hardware Resources: The basic components in the IFD-model with four possible states (S, D, L, and 0). By convention they are named in small letters.

Information Flow Diagram (IFD): Diagram to model a system's failure behaviour by its logical information flow to estimate the PFD and PFS.

Local List: Describes how hardware resources of a functional entity have to behave so that the according block will switch to a certain state. For each block there are three local lists: S, D, and L.

Markov Chain: A *State Based Formalism* to model complex systems. The different system states are represented by random variables with the Markov property, i.e. that the possible state transitions only depend on the current state.

Mean Time to Failure (MTTF): The expected value of the lifetime of a component or system. It can be calculated by the following formula:

$MTTF = \int_0^\infty R(t)dt$

Mean Time to Failure (MTTR): The expected repair time for a failed repairable component or system.

Timing Failure: Occurs if a component or system gives its results or orders too late, leading to a violation of the specification.

Omission Failure: A failure in which a component or system does not forward its results or orders as specified.

One-Edge: The outgoing edge of a BDD-node n with a variable x leading to the sub-BDD representing the expression of n in which x has been substituted with true.

One-Node: The node in a BDD representing the Boolean constant true.

Probability of Failure on Demand PFD(t): The Probability that a failure on demand will occur in the time interval $[0, t]$.

Probability of a Spurious Trip PFS(t): The Probability that a spurious trip will occur in the time interval $[0, t]$.

Reduce Operator: An operator in the original IFD-approach by Karim Hamidi. It eliminates duplicate and impossible sequences from the aggregated lists. In the modified IFD-model, this reduction is done by applying BDD-techniques.

Reliability Block Diagram: A *Combinatorial Formalism* to model the failure behaviour of systems based on its redundancy structure.

Reliability R(t): The probability that a component or system will work satisfying for a specified time t. Be T the random variable representing the lifetime of the system, $u(t)$ its probability density function and $U(t)$ its cumulative density function. Then the reliability, denoted as $R(t)$, is defined as follows:

$$R(t) = Pr(T > t) = 1 - U(t) = \int_t^\infty f(x)dx$$

Reduced Ordered Binary Decision Diagram (ROBDD): A BDD in which equivalent nodes are merged and nodes with two outgoing edges to the same node are eliminated. Normally created by using the *Apply Algorithm*.

Safety S(t): The probability that a system or component is working correctly or is shut down in the time interval $[0, t]$. It holds: $S(t) \geq R(t)$.

Shannon Decomposition: A method to transform Boolean expressions. Be Φ_{X_i} resp. $\Phi_{\overline{X_i}}$ the expression which results from substituting the variable X_i in the Boolean expression Φ with true resp. false. For a Boolean function $\Phi : \mathbb{B}^n \rightarrow \mathbb{B}$ it holds:

$\Phi = (X_i \wedge \Phi_{X_i}) \vee (\overline{X_i} \wedge \Phi_{\overline{X_i}})$

where X_i is a variable of Φ.

Spurious Trip: Unwanted and unspecified activation of safety mechanisms (like an alarm or a safety shutdown) of a system in a non-dangerous state.

State (for IFDs): There are four possible states for hardware resources, blocks and the global system in IFDs:

- S: Safe failure or spurious trip
- D: Dangerous failure
- L: Loss of information (Omission failure)
- 0: Working correctly

State Bases Formalisms: Modelling formalisms which use different system states and state transitions to represent the modelled systems. Examples are Markov Chains and Stochastic Petri Nets.

Stochastic Petri Net (SPN): A *State Based Formalism* to model complex systems. A SPN is a graph with two different kinds of nodes (places and transitions), edges (called arcs) which link either places to transitions or transitions to places. Places contain marks which can be moved by firing the transitions in order to model the system behaviour and state changes.

Unreliability U(t): The probability that a component or system will fail in a specified time interval $[0, t]$. It holds:

$U(t) = 1 - R(t)$

Weibull Distribution: A probability distribution with a shape parameter $\beta > 0$ and a scale parameter $\eta > 0$. Its probability density function is defined as follows:

$f(t) = \frac{\beta t^{\beta-1}}{\eta^{\beta}} e^{-(t/\eta)^{\beta}}$

Zero-Edge: The outgoing edge of a BDD-node n with a variable x leading to the sub-BDD representing the expression of n in which x has been substituted with false.

Zero-Node: The node in a BDD representing the Boolean constant false.

Zero Suppressed Binary Decision Diagram (ZBDD): A BDD in which equivalent nodes are merged and nodes with an one-edge leading to the Zero-node are eliminated. Normally created by using the *Apply Algorithm.*

Appendix C

Examples

C.1 Conservative Redundancy

1. Components:
p1, p2, p3: position sensors
r1, r2: rotation sensors
bus1, bus2: data busses
ecu1, ecu2, ecu3: ECUs
o12, o21, o31, o32: output stages
m1, m2: motors

2. Sub-models for SRC-, ST- and WD-blocks:
S(P1) = (p1=S)
D(P1) = (p1=D)
S(P2) = (p2=S)
D(P2) = (p2=D)
S(P3) = (p3=S)
D(P3) = (p3=D)
S(R1) = (r1=S)
D(R1) = (r1=D)
S(R2) = (r2=S)
D(R2) = (r2=D)
S(BUS) = (bus1=S) $\wedge$ (bus2=S)
S(ECU1) = (Input(S) $\wedge$ (ecu1=0)) $\vee$ (ecu1=S)
D(ECU1) = (Input(D) $\wedge$ (ecu1=0)) $\vee$ (ecu1=D)
S(ECU1s) = (Input(S) $\wedge$ (ecu1=0)) $\vee$ (ecu1=S)
D(ECU1s) = (Input(D) $\wedge$ (ecu1=0)) $\vee$ (ecu1=D)
S(ECU2) = (Input(S) $\wedge$ (ecu2=0)) $\vee$ (ecu2=S)

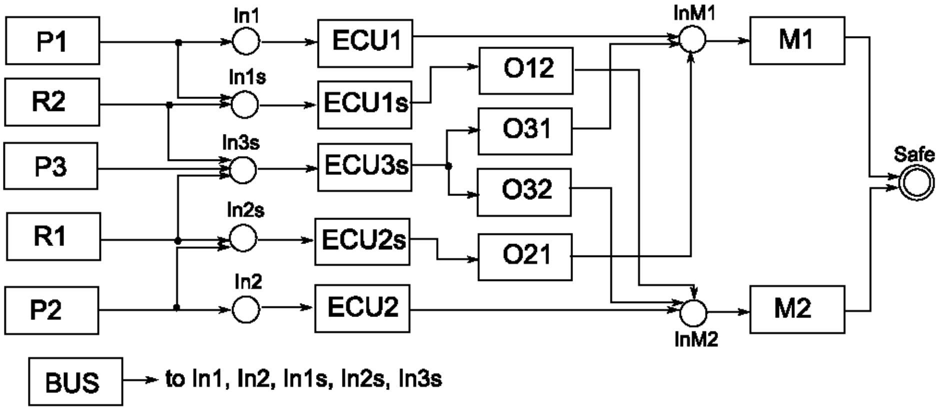
P1
R2
P3
R1
P2
In1
In1s
In3s
In2s
In2
ECU1
ECU1s
ECU3s
ECU2s
ECU2
O12
O31
O32
O21
InM1
InM2
M1
M2
Safe
BUS
to In1, In2, In1s, In2s, In3s

Figure C.1: IFD for the conservative steer-by-wire-system

$D(ECU2) = (Input(D) \land (ecu2=0)) \lor (ecu2=D)$
$S(ECU2s) = (Input(S) \land (ecu2=0)) \lor (ecu2=S)$
$D(ECU2s) = (Input(D) \land (ecu2=0)) \lor (ecu2=D)$
$S(ECU3s) = (Input(S) \land (ecu3=0)) \lor (ecu3=S)$
$D(ECU3s) = (Input(D) \land (ecu3=0)) \lor (ecu3=D)$
$S(O12) = (Input(S) \land (o12=0)) \lor (o12=S)$
$D(O12) = (Input(D) \land (o12=0)) \lor (o12=D)$
$S(O21) = (Input(S) \land (o21=0)) \lor (o21=S)$
$D(O21) = (Input(D) \land (o21=0)) \lor (o21=D)$
$S(O31) = (Input(S) \land (o31=0)) \lor (o31=S)$
$D(O31) = (Input(D) \land (o31=0)) \lor (o31=D)$
$S(O32) = (Input(S) \land (o32=0)) \lor (o32=S)$
$D(O32) = (Input(D) \land (o32=0)) \lor (o32=D)$
$S(M1) = Input(S) \lor (m1=s)$
$D(M1) = Input(D) \land (m1=0)$
All missing expressions are false.

3. Sub-models for DEC-nodes:
$S(In1) = (BUS = S) \lor (P1 = S)$
$D(In1) = (P1 = D)$
$S(In1) = (BUS = S) \lor (P2 = S)$
$D(In1) = (P2 = D)$
$S(In1s) = (BUS = S) \lor ((P1 = S) \land (R2 = S))$
$D(In1s) = (P1 = D) \land (R2 = D)$
$S(In2s) = (BUS = S) \lor ((P2 = S) \land (R1 = S))$
$D(In2s) = (P2 = D) \land (R1 = D)$
$S(In3s) = (BUS = S) \lor ((P3 = S) \land (R2 = S) \land (R1 = S))$

D(In3s) = $(\mathrm{P3} = \mathrm{D}) \wedge (\mathrm{R2} = \mathrm{D}) \wedge (\mathrm{R1} = \mathrm{D})$
S(InM1) = $(\mathrm{O21} = \mathrm{S}) \vee (\mathrm{O31} = \mathrm{S})$
D(InM1) = $(\mathrm{ECU1} = \mathrm{D}) \wedge (\mathrm{O21} = \mathrm{D}) \wedge (\mathrm{O31} = \mathrm{D})$
S(InM2) = $(\mathrm{O12} = \mathrm{S}) \vee (\mathrm{O32} = \mathrm{S})$
D(InM2) = $(\mathrm{ECU2} = \mathrm{D}) \wedge (\mathrm{O12} = \mathrm{D}) \wedge (\mathrm{O32} = \mathrm{D})$
D(Safe) = $(\mathrm{M1} = \mathrm{D}) \vee (\mathrm{M2} = \mathrm{D}) \vee ((\mathrm{M1} = \mathrm{S}) \wedge (\mathrm{M2} = \mathrm{S}))$
All missing expressions are false.

C.2 Remote Redundancy

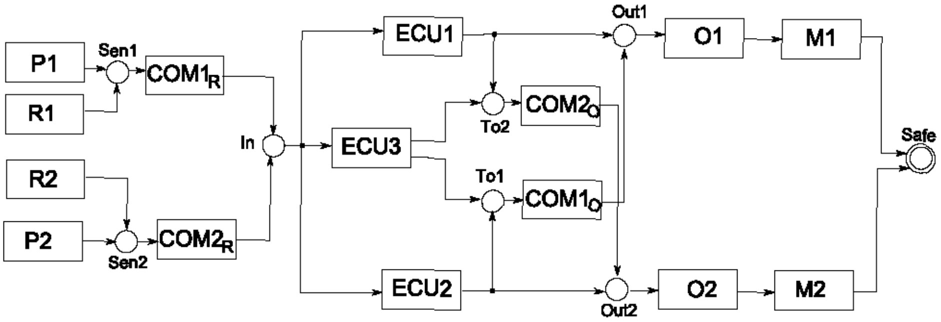
P1
R1
R2
P2
Sen1
Sen2
COM1R
COM2R
In
ECU1
ECU3
ECU2
To2
To1
COM2O
COM1O
Out1
Out2
O1
O2
M1
M2
Safe

Figure C.2: The IFD for the remote redundancy architecture

1. Components:

p1, p2: position sensors
r1, r2: rotation sensors
bus1, bus2: data busses
ecu1, ecu2, ecu3: ECUs
o1, o2: output stages
m1, m2: motors

2. Sub-models for SRC-, ST- and WD-blocks:

S(P1) = (p1=S)
D(P1) = (p1=D)
S(P2) = (p2=S)
D(P2) = (p2=D)
S(R1) = (r1=S)
D(R1) = (r1=D)
S(R2) = (r2=S)
D(R2) = (r2=D)
S(ECU1) = (Input(S) $\wedge$ (ecu1=0)) $\vee$ (ecu1=S)
D(ECU1) = (Input(D) $\wedge$ (ecu1=0)) $\vee$ (ecu1=D)
S(ECU2) = (Input(S) $\wedge$ (ecu2=0)) $\vee$ (ecu2=S)
D(ECU2) = (Input(D) $\wedge$ (ecu2=0)) $\vee$ (ecu2=D)
S(ECU3) = (Input(S) $\wedge$ (ecu3=0)) $\vee$ (ecu3=S)
D(ECU3) = (Input(D) $\wedge$ (ecu3=0)) $\vee$ (ecu3=D)
S(COM1R) = Input(S) $\vee$ ((bus1=S) $\wedge$ (bus2=S)) $\vee$ (ecu1=S)
D(COM1R) = (Input(D) $\wedge$ (bus1=0) $\wedge$ (ecu1=0)) $\vee$ (Input(D) $\wedge$ (bus2=0) $\wedge$ (ecu1=0))
S(COM2R) = Input(S) $\vee$ ((bus1=S) $\wedge$ (bus2=S)) $\vee$ (ecu2=S)

D(COM2R) = (Input(D) ∧ (bus1=0) ∧ (ecu2=0)) ∨ (Input(D) ∧ (bus2=0) ∧ (ecu2=0))
S(COM1O) = Input(S) ∨ ((bus1=S) ∧ (bus2=S)) ∨ (ecu1=S)
D(COM1O) = (Input(D) ∧ (bus1=0) ∧ (ecu1=0)) ∨ (Input(D) ∧ (bus2=0) ∧ (ecu1=0))
S(COM2O) = Input(S) ∨ ((bus1=S) ∧ (bus2=S)) ∨ (ecu2=S)
D(COM2O) = (Input(D) ∧ (bus1=0) ∧ (ecu2=0)) ∨ (Input(D) ∧ (bus2=0) ∧ (ecu2=0))
S(O1) = (Input(S) ∧ (o1=0)) ∨ (o1=S)
D(O1) = (Input(D) ∧ (o1=0)) ∨ (o1=D)
S(O2) = (Input(S) ∧ (o2=0)) ∨ (o2=S)
D(O2) = (Input(D) ∧ (o2=0)) ∨ (o2=D)
S(M1) = Input(S) ∨ (m1=s)
D(M1) = Input(D) ∧ (m1=0)
All missing expressions are false.

3. Sub-models for DEC-nodes:
S(Sen1) = (R1 = S) ∨ (P1 = S) ∨ (P1 = D)
D(Sen1) = (P1 = D) *wedge* (R1 = D)
S(Sen2) = (R2 = S) ∨ (P2 = S) ∨ (P2 = D)
D(Sen2) = (P2 = D) *wedge* (R2 = D)
S(In) = (COM1R = S) ∨ (COM2R = S)
D(In) = ((COM1R = S) ∨ (COM2R = D)) ∧ ((COM1R = D) ∨ (COM2R = S)) ∧ ((COM1R = D) ∨ (COM2R = D))
S(To1) = (ECU2 = S) ∧ (ECU3 = S)
D(To1) = (ECU2 = D) ∨ (ECU3 = D)
S(To2) = (ECU1 = S) ∧ (ECU3 = S)
D(To2) = (ECU1 = D) ∨ (ECU3 = D)
S(Out1) = (ECU1 = S) ∨ (COM1O = S)
D(Out1) = (ECU1 = D) ∧ (COM1O = D)
S(Out2) = (ECU2 = S) ∨ (COM2O = S)
D(Out2) = (ECU2 = D) ∧ (COM2O = D)
D(Safe) = (M1 = D) ∨ (M2 = D) ∨ ((M1 = S) ∧ (M2 = S))
All missing expressions are false.

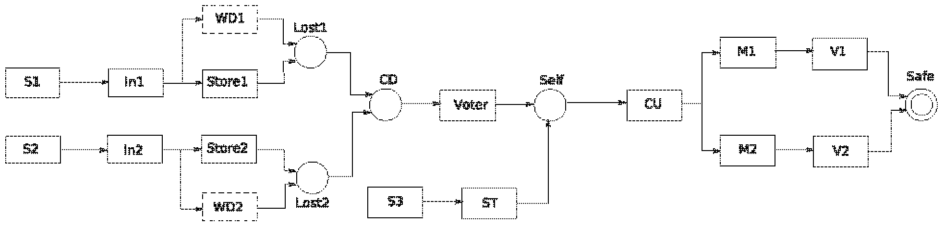
WD1
Lost1
S1
In1
Store1
CD
Voter
Self
CU
M1
V1
Safe
S2
In2
Store2
M2
V2
Lost2
WD2
S3
ST

Figure C.3: IFD of the chemical reactor

C.3 Chemical Reactor

1. Components:
s1, s2: zemperature sensors
s3: fault detector
in1, in2: input modules
mem1, mem2: memory
wd1, wd2: watchdogs
vot: voter
st: control module for fault detection
cu: central control unit
m1, m2: motors
v1, v2: valves

2. Sub-models for SRC-, ST- and WD-blocks:
D(S1) = (s1=D)
S(S1) = (s1=S)
D(S2) = (s2=D)
S(S2) = (s2=S)
D(S3) = (s3=D)
S(S3) = (s3=S)
D(In1) = Input(D) $\wedge$ (in1=0)
S(In1) = Input(S) $\wedge$ (in1=0)
L(In1) = (in1=L)
D(In2) = Input(D) $\wedge$ (in2=0)
S(In2) = Input(S) $\wedge$ (in2=0)
L(In2) = (in2=L)
D(WD1) = Input(L) $\wedge$ (wd1=D)
S(WD1) = Input(L) $\wedge$ (wd1=S)
D(WD2) = Input(L) $\wedge$ (wd2=D)
S(WD2) = Input(L) $\wedge$ (wd2=S)
D(Store1) = Input(D) $\wedge$ (mem1=0)

S(Store1) = (Input(S) $\wedge$ (mem1=0)) $\vee$ (mem1=S)
D(Store2) = Input(D) $\wedge$ (mem2=0)
S(Store2) = (Input(S) $\wedge$ (mem2=0)) $\vee$ (mem2=S)
D(Voter) = (Input(D) $\wedge$ (vot=0)) $\vee$ (vot=D)
S(Voter) = (Input(S) $\wedge$ (vot=0)) $\vee$ (vot=S)
D(ST) = (Input(D) $\wedge$ (st=0)) $\vee$ (st=D)
S(ST) = (Input(S) $\wedge$ (st=0)) $\vee$ (st=S)
D(CU) = (Input(D) $\wedge$ (cu=0)) $\vee$ (cu=D)
S(CU) = (Input(S) $\wedge$ (cu=0)) $\vee$ (cu=S)
D(M1) = Input(D) $\wedge$ (m1=0)
S(M1) = (Input(S) $\wedge$ (m1=0)) $\vee$ (m1=S)
D(M2) = Input(D) $\wedge$ (m2=0)
S(M2) = (Input(S) $\wedge$ (m2=0)) $\vee$ (m2=S)
D(V1) = (Input(D) $\wedge$ (v1=0)) $\vee$ (v1=D)
S(V1) = (Input(S) $\wedge$ (v1=0)) $\vee$ (v1=S)
D(V2) = (Input(D) $\wedge$ (v2=0)) $\vee$ (v2=D)
S(V2) = (Input(S) $\wedge$ (v2=0)) $\vee$ (v2=S)
All missing expressions are false.

3. Sub-models for Dec-nodes:
S(Lost1) = (WD1 = S) $\vee$ (Store1 = S)
D(Lost1) = (WD1 = D) $\vee$ (Store1 = D)
S(Lost2) = (WD2 = S) $\vee$ (Store2 = S)
D(Lost2) = (WD2 = D) $\vee$ (Store2 = D)
S(CD) = (Lost1 = S) $\vee$ (Lost2 = S)
D(CD) = (Lost1 = D) $\wedge$ (Lost2 = D)
S(Self) = (Voter = S) $\vee$ (ST = S)
D(Self) = (Voter = D) $\wedge$ (ST = D)
S(Safe) = (V1 = S) $\vee$ (V1 = S)
D(Safe) = (V1 = D) $\wedge$ (V2 = D)
All missing expressions are false.

Appendix D

IFD-File Example

```
Version 0.3
1000
%Components
m1 0 1.3E-05 0 0
m2 0 1.3E-05 0 0
r1 0 1E-09 1E-09 0
r2 0 1E-09 1E-09 0
p1 0 1E-09 1E-09 0
p2 0 1E-09 1E-09 0
bus1 0 2E-09 0 0
bus2 0 2E-09 0 0
ecu1 0 2.5E-07 2.5E-07 0
ecu2 0 2.5E-07 2.5E-07 0
ecu3 0 2.5E-07 2.5E-07 0
o1 0 2.5E-08 2.5E-08 0
o2 0 2.5E-08 2.5E-08 0
%Blocks
P1
0
3
false
d(p1,S)
d(p1,D)

_____________
P2
1
3
```

```
false
d(p2,S)
d(p2,D)

—————
R1
2
3
false
d(r1,S)
d(r1,D)

————-
R2
3
3
false
d(r2,S)
d(r2,D)

—————
COM1R
4
0
false
input(S);d(bus1,S)#d(bus2,S);d(ecu1,S)
input(D)#d(ecu1,0)#d(bus1,0);input(D)#d(ecu1,0)#d(bus2,0)

—————
COM2R
5
0
false
input(S);d(bus1,S)#d(bus2,S);d(ecu2,S)
input(D)#d(ecu2,0)#d(bus1,0);input(D)#d(ecu2,0)#d(bus2,0)

—————
Sen1
6
2
false
```

```
(&P1&=#S OR &R1&=#S) OR &P1&=#D
(&P1&=#D AND &R1&=#D)
false
————-
Sen2
7
2
false
(&P2&=#S OR &R2&=#S) OR &P2&=#D
(&P2&=#D AND &R2&=#D)
false
————-
In
8
2
false
&COM1R&=#S AND &COM2R&=#S
(&COM1R&=#D AND &COM2R&=#D) OR (&COM1R&=#S AND &COM2R&=#D)
OR (&COM1R&=#D AND &COM2R&=#S)
false
————-
ECU2
9
0
false
input(S)#d(ecu2,0);d(ecu2,S)
input(D)#d(ecu2,0);d(ecu2,D)

————
ECU3
10
0
false
input(S)#d(ecu3,0);d(ecu3,S)
input(D)#d(ecu3,0);d(ecu3,D)

————
ECU1
11
0
false
```

```
input(S)#d(ecu1,0);d(ecu1,S)
input(D)#d(ecu1,0);d(ecu1,D)

——————
To1
12
2
false
&ECU2&=#S AND &ECU3&=#S
&ECU2&=#D OR &ECU3&=#D
false
——————
To2
13
2
false
&ECU1&=#S AND &ECU3&=#S
&ECU1&=#D OR &ECU3&=#D
false
——————-
Out1
14
2
false
&ECU1&=#S OR &COM1O&=#S
&ECU1&=#D AND &COM1O&=#D
false
——————-
Out2
15
2
false
&ECU2&=#S OR &COM2O&=#S
&ECU2&=#D AND &COM2O&=#D
false
——————-
COM1O
16
0
false
input(S);d(bus1,S)#d(bus2,S);d(ecu1,S)
```

input(D)#d(ecu1,0)#d(bus1,0);input(D)#d(ecu1,0)#d(bus2,0)

COM2O
17
0
false
input(S);d(bus1,S)#d(bus2,S);d(ecu2,S)
input(D)#d(ecu2,0)#d(bus1,0);input(D)#d(ecu2,0)#d(bus2,0)

O1
18
0
false
input(S)#d(o1,0);d(o1,S)
input(D)#d(o1,0);d(o1,D)

M1
19
0
false
d(m1,S);input(S)
input(D)#d(m1,0)

O2
20
0
false
input(S)#d(o2,0);d(o2,S)
input(D)#d(o2,0);d(o2,D)

M2
21
0
false
d(m2,S);input(S)
input(D)#d(m2,0)

Safe
22
2
true
false
&M1&=#D OR (&M1&=#S AND &M2&=#S) OR &M2&=D
false

-
%Edges
0
0
6

1
2
6

2
6
4

3
3
7

4
1
7

5
7
5

6
4
8

7
5

8
8 8 10
9 8 11
10 8 9
11 9 12
12 9 15
13 10 12
14 10 13
15 11 13
16 11 14
17 13 17

18 12 16
19 16 14
20 17 15
21 14 18
22 18 19
23 19 22
24 15 20
25 20 21
26 21 22

Bibliography

[1] http://www2.informatik.tu-muenchen.de/projekte/cck/.

[2] http://www.jens-langner.de/dessolver/.

[3] Military Handbook Reliability Prediction of Electronic Equipment. Technical report, United States Department of Defence, 1991.

[4] Report on the Accident to Airbus A320-211 Aircraft in Warsaw. Technical report, Main Commission Aircraft Accident Investigation Warsaw, 1993.

[5] *Oxford Dictionary of Computing.* Oxford University Press, 2004.

[6] System Safety Handbook. Technical report, United States Federal Aviation Administration, 2008.

[7] Interim report on the accident on 1 june 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris. Technical report, Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile (BEA), 2009.

[8] *Collins English Dictionary - Complete & Unabridged 10th Edition.* HarperCollins Publishers, 2011.

[9] Report of Japanese Government to the IAEA Ministerial Conference on Nuclear Safety - The Accident at TEPCO's Fukushima Nuclear Power Stations. Technical report, Nuclear Emergency Response Headquarters of the Government of Japan, June 2011.

[10] Alfred V. Aho, Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques & tools.* Pearson Addison-Wesley, 2007.

[11] H. Andersen and H. Hulgaard. Boolean Expression Diagrams. In *12th Annual IEEE Symposium on Logic in Computer Science*, pages 88–111. IEEE, 1997.

[12] J. D. Andrews and L. M. Bartlett. Efficient Basic Event Orderings for Binary Decision Diagrams. In *1998 PROCEEDINGS Annual RELIABILITY and MAINTAINABILITY Symposium*, pages 61–68, 1998.

[13] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The AltaRica Formalism for Describing Concurrent Systems. *Fundamenta Informaticae*, 34:109–124, 2000.

[14] Jens Bachmann, Martin Riedl, Johann Schuster, and Markus Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. *Lecture Notes in Computer Science*, 5404/2009:485–496, 2009.

[15] H. Belhadaoui, O. Malassé, G. Buchheit, and J.F. Aubry. Evaluation de la disponibilité d'un processeur tolérant aux fautes. In *8ème Congrès international pluridisciplinaire en Qualité et Sûreté de Fonctionnement (Qualita 2009)*, 2009.

[16] M. Boiteau, Y. Dutuit, A. Rauzy, and J.-P. Signoret. The AltaRica dataflow language in use: modeling of production availability of a multi-state system. *Reliability Engineering and System Safety*, 91:747–755, 2006.

[17] M. Bouissou and J. L. Bon. A new formalism that combines advantages of fault-trees and markov models: Boolean logic driven markov processes. In *Reliability Engineering and System Safety*, pages 149–163, 2003.

[18] Marc Bouissou. A generalization of Dynamic Fault Trees through Boolean logic Driven Markov Processes (BDMP). In *Proceedings of the European Safety and Reliability Conference (ESREL 2007)*, pages 1051–1058, 2007.

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

[20] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:667–691, 1986.

[21] A. Cabarbaye and J. Faure. Apport d'un outil d'optimisation globale à l'ajustement des modèles statistiques. In *41têmes Journées de Statistique*, 2009.

[22] Noam Chomsky. Three Models for the Description of Language. *IRE Transactions on Information*, 2:113–123, 1956.

[23] D. Coppit and K. Sullivan. Galileo: A Tool Built From Mass-Market Applications. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 750–753, 2000.

[24] J. Dugan, B. Venkataraman, and R. Gulati. DIFtree: A software package for the analysis of dynamic fault tree models. In *Reliability and Maintainability Symposium. 1997 Proceedings, Annual*, pages 64–70. IEEE, 1997.

[25] S. J. Dugan and F. Patterson-Hine. A Combinatorial Approach to Modeling Imperfect Coverage. *IEEE Transactions on Reliability*, 44(1):87ff., 1995.

[26] S. J. Dunnett and J. Andrews. A binary decision diagram method for phased mission analysis of non-repairable systems. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 220(2):93–104, 2006.

[27] K. Echtle, T. Kimmeskamp, S. Jaquet, O. Malassé, M. Pock, and M. Walter. Reliability Analysis of a Control Systems Built Using Remote Redundancy. In *Advances in Risk and Reliability Technology Symposium*, pages 335–346, 2009.

[28] M. Jallouli et al. Dependability Consequences of Fault-Tolerant Technique Integrated in Stack Processor Emulator using Information Flow Approach. In *2008 International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, pages 1–6, 2008.

[29] M. Jallouli et al. Evaluation of Important Reliability Parameters using VHDL-RTL modelling and Information Flow Approach. In *Proceedings of the European Safety and Reliability Conference (ESREL 2008)*, pages 2549–2557, 2008.

[30] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language 3rd Edition.* Addison-Wesley, 2003.

[31] N. Gaitanis. The design of totally self-checking TMR fault-tolerant systems. *IEEE Transactions on Computers*, 37:1450–1454, 1988.

[32] A. Goubermann, M. Riedl, J. Schuster, M. Siegle, and M. Walter. LARES - A Novel Approach for Describing System Reconfigurability

in Dependability Models of Fault-Tolerant Systems. In *Proc. of the European Safety and Reliability Conference (ESREL 2009)*, pages 153–160. Taylor and Francis, 2009.

[33] A. Griffault, S. Lajeunesse, G. Point, A. Rauzy, J.-P. Signoret, and P. Thomas. The AltaRica Language. In *Proceedings European Safety and Reliability Associates Conference (ESREL 98)*, 1998.

[34] C. Grinstead and J.L. Snell. *Introduction to Probability.* American Mathematical Society, 1997.

[35] K. Hamidi. *Contribution à un modèle dévaluation quantitative des performances fiabilistes de fonctions électroniques et programmables dédiées à la sécurité.* PhD thesis, Institut National Polytechnique de Lorraine, 2005.

[36] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7:735–736, 1964.

[37] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[38] Georg Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras.* PhD thesis, Universität Nürnburg-Erlangen, 2006.

[39] Way Kuo and Ming J. Zuo. *Optimal Reliability Modeling - Principles and Applications.* John Wiley and Sons, 2003.

[40] K. Lampka and M. Siegle. Analysis of Markov Reward Models using Zero-suppressed Multi-terminal BDDs. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, 2006.

[41] Christoph Lindemann. DSPNexpress: A Software Package for the Efficient Solution of Deterministic and Stochastic Petri Nets. *Performance Evaluation*, pages 3–21, February 1995.

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

[43] Y. Papadopoulos, C. Grante, L. Grunske, and B. Kaiser. Continuous Assessment of Designs and Re-Use in Model-Based Safety Analysis. In *16 th IFAC World Congress*, 2005.

[44] Y. Papadopoulos and J. A. McDermid. Hierarchically Performed Hazard Origin and Propagation Studies. *Lecture Notes in Computer Science*, 1698/1999:688–701, 1999.

[45] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[46] M. Pock, H. Belhadaoui, O. Malassé, and M. Walter. 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)*, pages 1829–1838. Taylor & Francis Ltd., 2008.

[47] Michael Pock and May Walter. Efficient Extraction of the Structure Formula from Reliability Block Diagrams with Dependent Basic Events. In *Proc. of the European Safety and Reliability Conference (ESREL 2007)*, pages 1015–1023. Taylor and Francis, 2007.

[48] D. Presscott and J. Andrews. A Cause Consequence Analysis Approach to Modelling Multi-Platform Phased Missions. In *Advances in Risk and Reliability Technology Symposium*, pages 419–437, 2009.

[49] Arno Puder, Kay Römer, and Frank Pilhofer. *Distributed systems architecture: a middleware approach.* Morgan Kaufmann, 2006.

[50] Uwe Kay Rakowsky and Nicole Richardson. *Wörterbuch der Zuverlässigkeit.* LiLoLe-Verlag, 2001.

[51] A. Ralston and E. Reilly, editors. *Encyclopedia of Computer Science.* Chapman and Hall, 1993.

[52] Antoine Rauzy. New Algorithms for Fault Tree Analysis. *Reliability Engineering and System Safety*, 40(3):203–211, 1993.

[53] Antoine Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1–12, 2002.

[54] E. Reilly, editor. *Concise Encyclopedia of Computer Science.* Wiley and Sons, 2004.

[55] Harold E. Roland and Brian Moriarty. *System safety engineering and management.* Wiley and Sons, 1990.

[56] R.A. Sahner, K.S. Triverdi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems.* Kluwer Academic Publishers, 1996.

[57] Thomas Schickinger and Angelika Steger. *Diskrete Strukturen 2.* Springer Verlag, 2002.

[58] Winfried Schneeweiss. *Die Fehlerbaum-Methode.* LiLoLe Verlag, 1999.

[59] Winfried Schneeweiss and Max Walter. *The Modeling World of Reliability/Safety Engineering.* LiLoLe Verlag, 2005.

[60] Uwe Schöning. *Algorithmik.* Spektrum Akademischer Verlag, 2001.

[61] Claude Elwood Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 28:379–423 623–656, 1948.

[62] M. L. Shoomann. *Reliability of Computer Systems and Networks.* Wiley and Sons, 2002.

[63] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[64] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 2003.

[65] M. Walter and C. Trinitis. 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)*, pages 47–54. VDE-Verlag Berlin, 2007.

[66] Max Walter and Carsten Trinitis. Simple Models for High-Availability Systems with Dependent Components. In *Proceedings of the 2006 European safety and reliability conference, vol. 3*, page 1719ff. Taylor and Francis, 2006.

[67] Walodi Weibull. A statistical distribution function of wide applicability. *Journal of Applied Mechanics*, 18:293–297, 1951.

[68] Poul Frederick Williams, Macha Nikolskaa, and Antoine Rauzy. Bypassing BDD Construction for Reliability Analysis. *Information Processing Letters*, 75(1-2), July 2000.

[69] X. Zang, D. Wang, H. Sung, and K. Trivedi. A BDD-Based Algorithm for Analysis of Multistate Systems with Multistate Components. *IEEE Transactions on Computers*, 52:1608–1618, December 2003.

UNIVERSITÉ
DE LORRAINE

AUTORISATION DE SOUTENANCE DU DOCTORAT DE L'UNIVERSITE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur SIEGLE Markus, Professeur, Universität der Bundeswehr München,

Monsieur RAUZY Antoine, Professeur, Ecole Polytechnique Palaiseau.

L'Administrateur Provisoire de l'Université de Lorraine, autorise :

Monsieur Michael POCK

à soutenir devant un jury de l'UNIVERSITE DE LORRAINE, une thèse intitulée :

"Evaluation technique for Safety Critical Systems"

en vue de l'obtention du titre de :

DOCTEUR DE l'UNIVERSITE DE LORRAINE

Intitulé du doctorat : **"Automatique, Traitement du Signal et des Images"**

Fait à Vandoeuvre, le **14 Mars 2012**
Pour l'Administrateur Provisoire par délégation,
Le Chargé de Mission,
François LAURENT

UNIVERSITÉ DE LORRAINE
34 COURS LÉOPOLD - CS 25233
54052 NANCY CEDEX
TÉL : 03 54 50 54 00
CONTACT@UNIV-LORRAINE.FR
WWW.UNIV-LORRAINE.FR

Keywords: Binary Decision Diagrams, Safety Critical Systems, System Modelling, Hierarchical Models

This thesis presents a novel approach for modelling safety critical systems which takes into account several failure modes both for components and the global system. The so called Information Flow Diagrams (IFDs) were originally developed in a previous PhD-thesis. In this work, the evaluation if the IFD-approach should be made more efficient by using Binary Decision Diagrams (BDDs).
This thesis will explain why such a model is necessary and practical, followed by a detailed explanation of the IFD-model. This includes its hierarchical structure and how this model can be applied.
The next step is to formalise the original IFD-model in order to enable more efficient evaluation techniques. It will be explained why these formalisation steps were taken and what was gained by using them.
Afterwards a detailed explanation of the developed algorithms is presented. These algorithms are based on a combination of different BDD-techniques. Zero Suppressed BDDs (ZBDDs) are combined with Boolean Expression Diagrams (BEDs). Furthermore, the structure of the IFDs is used in order to construct a large BDD out of several smaller BDDs. This increases the efficiency of the evaluation process.
The presented techniques are evaluated by analysing several use cases which are explained in this work.

Mots clés : Binary Decision Diagrams, Systèmes de sécurité fonctionnelle, Modélisation des systèmes, Models hiérarchiques

Cette thèse présente une nouvelle approche pour la modélisation des systèmes de sécurité fonctionnelle qui prend en compte plusieurs modes de défaillance pour les composants et le système global. Les diagrammes de flux d'information (IFDs) ont été initialement développé dans un thèse précédent. Dans ce travail, l'évaluation si l'approche flux d'information être rendue plus efficace par utiliser les diagrammes de décision binaires (BDD).
Cette thèse sera d'expliquer pourquoi ce modèle est nécessaire et pratique, suivie d'une explication détaillée des IFDs. Cela inclut sa structure hiérarchique et comment ce modèle peut être appliqué.
La prochaine étape est la formalisation du modèle IFD original pour permettre l'utilisation des techniques d'évaluation plus efficaces. Il sera expliqué pourquoi ces étapes de formalisation ont été prises et les avantages de leur utilisation.
Ensuite une explication détaillée des algorithmes développés est présenté. Ces algorithmes sont basés sur une combinaison de différentes techniques de BDD. Zero Suppressed BDDs (ZBDDs) sont combinées avec des Boolean Expression Diagrams (BEDs). En outre, la structure des IFD est utilisé pour construire un BDD global sur plusieurs petits BDDs. Cela augmente l'efficacité du processus d'évaluation.
Les techniques présentées sont évaluées par l'analyse de plusieurs cas d'utilisation qui sont expliqués dans cet travail.