University of Twente
Enschede - The Netherlands

A Bayesian network reliability software tool

Master thesis

ing. P.F.Th. Zandbergen

April 29, 2008

Committee

dr. H. Boudali
dr. M.I.A. Stoelinga
Ir. A.F.E. Belinfante

Research Group

University of Twente
Faculty of Electrical Engineering,
Mathematics and Computer Science
Department of Computer Science
Formal Methods and Tools

Abstract

Fault Trees (FT) are widely used for reliability analysis. Static FT who have no temporal and/or functional failure dependencies are solved by using Binary Decision Diagrams. A FT extended with gates capable of modelling temporal and/or functional failure dependencies is called a Dynamic FT. A Dynamic FT is usually solved by a conversion to a Markov Chain (MC). Unfortunately a MC will have an effect called state space explosion. This means that the number of states of the MC can become extremely large. This is an undesirable effect. Bayesian Networks (BNs) is a different method to calculate unreliability. This method requires less states than a MC and is an attractive alternative to compute unreliability. But since most systems are drawn as (D)FT, it would be nice if the user can still use (D)FTs but calculate the unreliability via BNs. The constructed tool (called IDIC) provides this functionality. The user can draw his FT in the GUI of the tool and by pressing a button he lets the program calculate the unreliability of his system by translating the FT to a BN and analysing it.

IDIC consists of two main parts, the conversion and the GUI. In the GUI the user can draw several FT constructs (eg. several types of gates and basic events) with the help of the package Jgraph. These constructs can be edited, deleted, moved and added to a central drawing area. Saving and loading functionality is present, therefore drawn FTs can be stored and loaded for future use. The conversion is a step-by-step process. The first step is reading a specific formatted file with the help of the ANTLR package. ANTLR uses the information from the file to construct a standard formatted BN. This BN will be analysed by the package SMILE. The BN is translated from standard format to SMILE format and analysed.
IDIC will read the drawn FT, convert it to a BN and analyse it. The results, which is the unreliability per state, is displayed in the GUI.

IDIC has been tested via two case studies. The results of these case studies are good. Compared with the tool Galileo, which uses MCs to analyse the FT, the results are similar. The differences are smaller than a thousandth in one case and a millionth in the other. But the time needed is the key factor here. The first case required more than 8 minutes for Galileo to solve using MCs, IDIC only uses 10 seconds to solve this case. The other case study was similar in computation time.

IDIC is a prototype, the algorithms, data structures and packages used are not the fastest, least memory/space consuming. They are however easy to use, intuitive and well documented. Therefore although IDIC is functional, there are several ways to improve several components. This thesis lists a few of those possible improvements.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reliability analysis is an important concept. Commonly used in areas as space, medical or chemical industries where it is costly (in money or lives) to correct failed systems.

You want to know how reliable a system is before you send it to Mars, or use it to replace a person's heart. Fault Tree modelling is a way to model a system and perform analysis on it. Fault Trees (FTs) are widely used in this field. This is because they are easy to use and can be effectively solved with the help of Binary Decision Diagrams (BDDs). These standard or Static Fault Trees (SFTs) are unable to deal with temporal and functional failure dependencies.FT that include these temperal and functional failure dependencies are called Dynamic Fault Trees (DFTs). These DFTs are analysed using the method of Markov Chains (MCs). However Markov Chains are prone to the state space explosion problem [1]. Bayesian Networks (BNs) are a relative new formalism in this field. Its popularity is starting to grow among system analysts. Bayesian Networks are capable to model probabilistic models and analyse them. This can be done in a more efficient way than Markov Chains.

1.1 Motivation

Dynamic Fault Trees are solved using Markov chains. Unfortunately this method has a serious problem when dealing with large to very large networks. The conversion from DFTs to MCs requires a lot of space to deal with the additional nodes and edges the MC uses, to correctly deal with the functionality given in the DFT. This problem is called the 'state space' problem.
Bayesian Networks require less space than a Markov chain. This is already evident with small examples and only increases with the larger DFTs.
Markov Chains will add new nodes and edges with each new variable or component. Bayesian Networks work differently. They display the functionality similar to Fault Trees. With an (almost) equal number of nodes, the 'state space' is small compared to Markov Chains.
So why not solve DFTs with BNs.

1.2 Goal of the Project

The goal of the project is to deliver a tool and a Graphical User Interface (GUI) in which the user can create static and dynamic Fault Trees. These trees should represent the desired system model. Therefore the GUI must incorporate the means to place individual gates and events, combine them, provide means to change properties of these constructs and finally provide means to set some properties of the network itself. The FT constructs should automatically be converted to Bayesian Network constructs upon the pressing of a button.

[1] The size of the model doubles with addition of each new atomic proposition[1].

The tool should then provide several analysis capabilities. Such as the probability the system should fail. The tool will use several existing packages, algorithms and libraries.

1.3 Approach

In this work, an analysis is done of Fault Trees, MCs and BNs. The theoretical background will be laid out. The tool will consist of two main parts; The conversion algorithm that converts the Fault Trees to Bayesian network and analyses it and the GUI in which the user draws the FT. The tool will be a prototype with minimal functionality, yet fully functional.

The user can draw a FT in the tool's GUI, this FT needs to have save and load functionality. After setting parameters the tool can starts the conversion. With the help of ANTLR generated JAVA files a textfile containing the FT is read and parsed. The result, a ANTL tree, is then selected and used for the next step. This next step is the creation of a Bayesian Network using the information stored in the ANTLR tree. This Bayesian Network exists only in the computers memory at this point and is a standard network. To get some probability results from this network it needs to be analysed. The SMILE package from GENIE is used for this step. After transforming the BN, to a format that SMILE can read and understand, the analysis is started. The results from this will go back to the tool's GUI.

1.4 Result / Contributions

The result of this all is a tool in which the user can draw Fault Trees, set their properties and watch the probability of the network. Internally the tool converts the FT to a BN. Some case studies done with the tool, deliver the same probability result as tools which use Markov Chains to calculate the results. The tool has the standard methods to calculate the probability, but can be extended to included several[2] of the capabilities of tools as Galileo and even more functionality not present in Galileo like setting evidence. Also the tool uses generally less memory and time compared to tools which use Markov Chains to calculate the probabilities. This is determined by using a common watch and memory usage statistics of the computer itself. The precise amount of time and memory the tool is using is not tested. This is left as future work, as the implementation must be extended to incorporate the most efficient algorithms.
The tool is a prototype. Most of the functionality is present, but there are several extensions and improvements still left to be implemented. The tool is called IDIC, which stands for Infinite Diversity in Infinite Combinations (IDIC). The tool is a combination of several unlikely packages brought together to form a union, which is the core of the Vulcan philosophy of IDIC.

1.5 Layout of this thesis

In the background section some basic theory of Fault Trees is given, both static (section 2.1.1) and dynamic (2.1.2). Tools like Galileo uses Markov Chains to solve dynamic Fault Trees. Some theory about Markov chains is given in 2.1.3. Bayesian Networks are introduced in section 2.2. The main chapter 3 deals with the implementation of the tool. It begins with an introduction on the design and continues with a step-by-step description of all the steps needed to begin drawing a FT and ending with the probability of the systems unreliability back in the GUI.
Two case studies implemented in IDIC are discussed in 4. The conclusion is chapter 5.

[2] Analysis of the capabilities of Galileo is in order to determine the extend of the capabilities to be included.

Chapter 2

Background

This thesis covers the conversion of Fault Trees to Bayesian Networks. But what kind of Fault Trees, and what constitutes a Bayesian Network?
This Background chapter is divided into 2 main sections. Section 2.1 is on FT and 2.2 is on BN. Section 2.1 is further divided in section Static Fault Trees 2.1.1, Dynamic Fault Trees 2.1.2 and Markov Chains 2.1.3.
Section 2.2 has a example on Bayesian Networks and the concept of Discrete Time Bayesian Networks is introduced.

2.1 Fault Trees

Fault Tree modelling is a way to model a system and perform analysis on it. Fault Trees are widely used. This is because they are easy to use and can be effectively analysed with the help of Binary Decision Diagrams (BDD). Fault Tree models are used, to provide a framework, to analyse quantitatively and qualitatively the failure behaviour of a system. If the failure probability of all the individual components is known then the probability of the failure of the entire system can be calculated.
For instance in Figure 2.1, it is useful to determine the probability of all 3 processors to fail (subsystem of the entire computer system). If the probability is high then extra (redundant) processors should be incorporated. Fault Tree analysis is also important for systems with high dependencies. Systems designed to leave this planet should be quite reliable, because replacing parts is a costly task. In medical systems it is also important. A surgeon replacing a men's heart with an artificial one would like his equipment as well as the replacement organ to be highly reliable.

By analysing the model, one can precisely see which failure events have been considered. This can be useful in management decisions and risk assessments. There are two versions of Fault Trees discussed in this document; Static and Dynamic Fault Trees. Static Fault Trees is the standard Fault Tree. These static FT lack the ability to model dynamic behaviour (e.g. order in which events occur). Dynamic Fault Trees (DFT) is a extension of static FT with new gates to incorporate dynamic behaviour of system components.

2.1.1 Static Fault Trees

Fault Trees are not normal trees as in a graph-theoretical sense [9]. A normal tree consists of nodes and edges. All nodes have exactly one parent, but can have multiple children. A FT is read from bottom to top. If arrows are used the node without **out**going arrows is called 'top node' or, to use the tree analogy, the 'root'. The nodes without **in**coming arrows are called 'leafs'. Because a FT is read from leaf to root, there can exist multiple paths. Thus a node can have multiple parents, in contradiction with normal trees. Fault Trees usually have a root representing the 'full

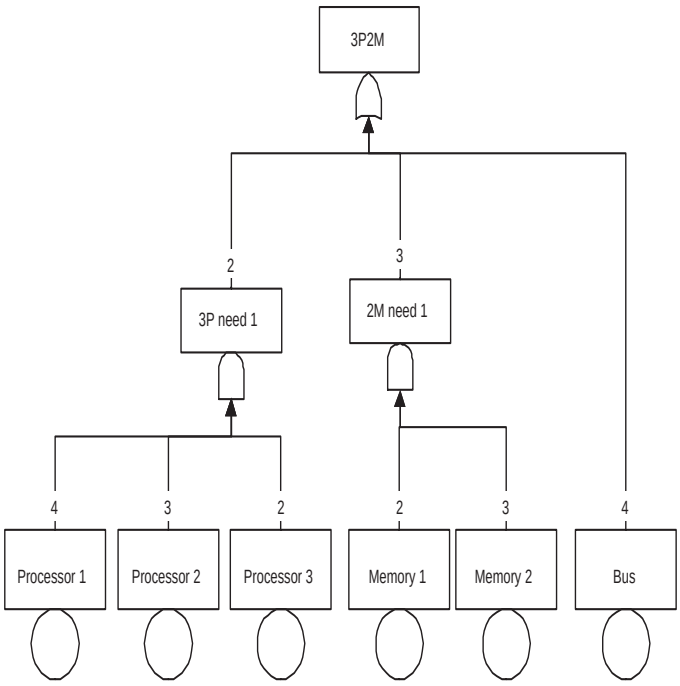


Figure 2.1: Computer system

system' with the underlying sub branches being (sub) components, or subsystems, of the whole. An example is that of a computer system. The tree, with the Computer system failure as root node (represented by the text "3P2M" Meaning 3 Processors, 2 Memories), and its subcomponents below is displayed in Figure 2.1. The whole computer fails if all of the three processors fails, both memories fail or when the bus fails.

OR

It is known that the whole system fails, if *one of* the subcomponents fails. This situation is displayed in Figure 2.1 by means of a logical *OR*, shown in Figure 2.2.

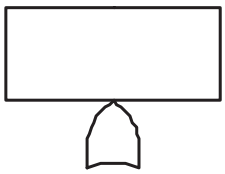


Figure 2.2: *OR* gate

If any of the subcomponents fails, then the *OR* will provide a logical '1' to the box "3P2M". Consequently the system fails. A component failure or failure event is denoted by a logical '1'.

AND

Further down the tree we see three processors. Only when *all* 3 have failed, this subcomponent fails. Therefore a logical *AND* (Figure 2.3) is needed here.

Only when all processors have failed (and therefore become '1') will the logical '1' be transmitted further up the tree.

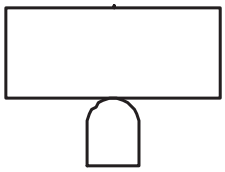


Figure 2.3: *AND* gate

K/M

Besides the *OR* and *AND* gate there is one more gate to choose from in a SFT. The *K/M* gate; the output for this gate is only true (i. e. equals '1') when K or more inputs from M inputs occur. In Figure 2.4 the output occurs when 2 or more inputs (of total 3) occur.

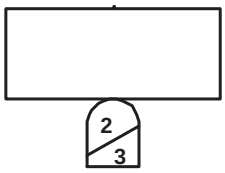


Figure 2.4: K/M gate

This is an excellent way of showing all sorts of (logical) relationships between several system components. The tree can be easily extended. For example, the processors could be mounted on a circuit board; 2 on one board and the third on a second board. This extension is displayed in Figure 2.5.

Now when either boards fails, the system will fail (*OR*). A circuit board fails when all the processors on it have failed (*AND*).
This can be extended as far as necessary. As long as there is a logical relation between system components, and therefore between individual system components, and the system failure.

Basic Events

Of course a FT with just gates says absolutely nothing. You need basic building blocks. In case of the computer system these are the actual processors, memories and bus.
These basic building blocks are called Basic Events Figure 2.6.

A processor does not fail or not. It has a chance of failing. This is called the *failure rate*. During its operation the component will have a certain chance of failing. If you would calculate the failure rate of the entire system, the failure rates of the leafs have to be propagated to the root.

2.1.2 Dynamic Fault Trees

Static Fault Trees use Boolean gates to represent how component failures combine to produce system failure. Dynamic Fault Trees add temporal and functional dependencies: system failures can depend on the ***order*** of component failures and component failures can depend on other component failures [12]. Dynamic Fault Trees use several (additional) gates to enforce this ordering.

Priority AND

The *Priority AND* (PAND) displayed in Figure 2.7 behaves almost as a normal *AND*. The output occurs only when all inputs occur. In addition the output only occurs when the inputs i. e. failure events occur in the given order (usually from left to right).

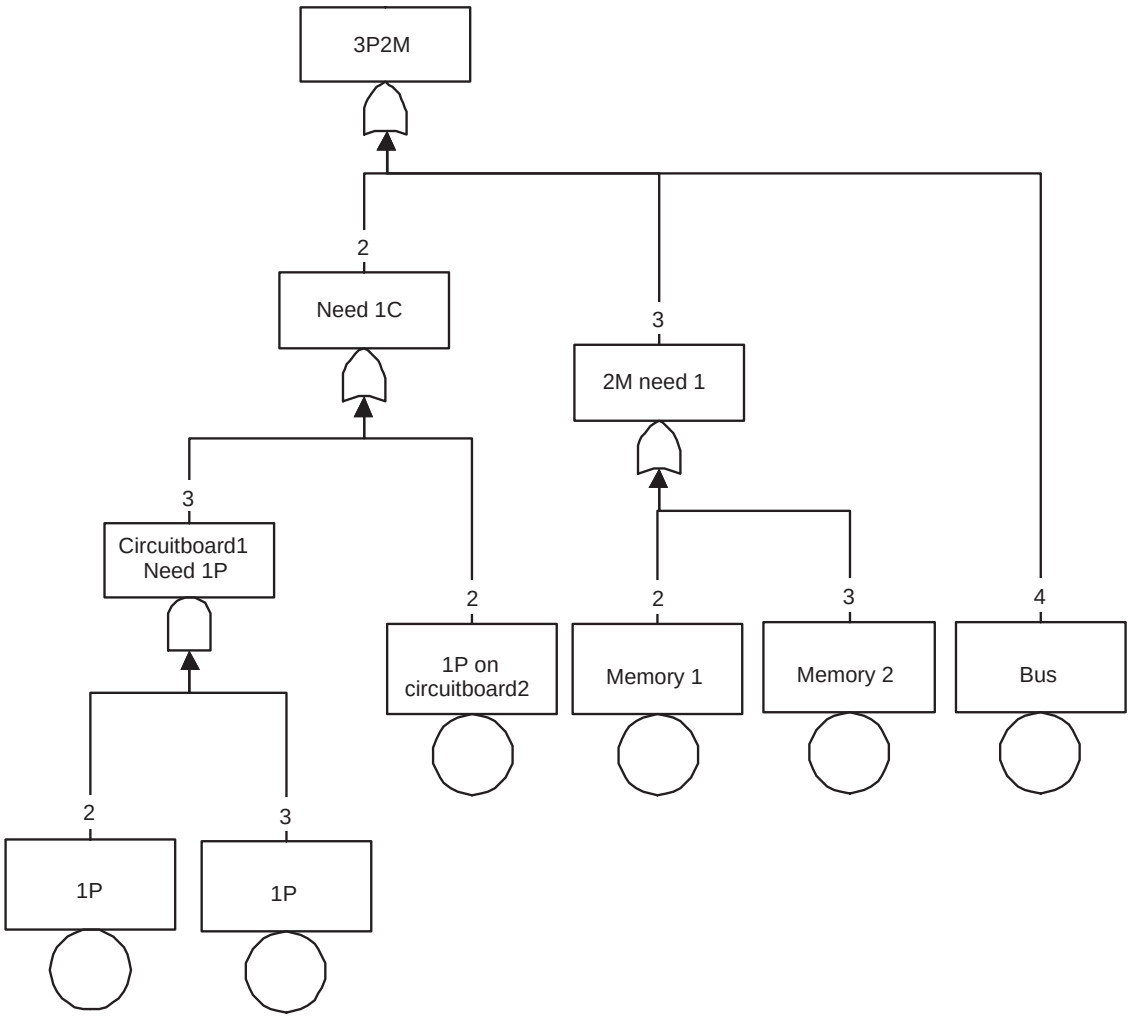


Figure 2.5: Computer system extended with circuit boards

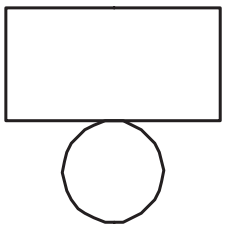


Figure 2.6: Basic Event

SEQuence enforcing

The SEQuence enforcing gate (SEQ) asserts that failures can only occur in a given order. In Figure 2.8 the gate is shown.

As will be shown later, the behaviour of the *SEQ* gate can also be modelled by a *Cold Spare* (CSP)

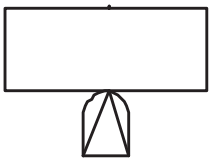


Figure 2.7: Priority AND

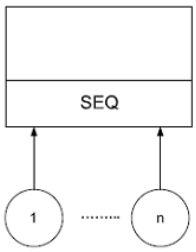


Figure 2.8: SEQuence enforcing

Functional DEPendency

The *Functional DEPendency gate* (FDEP) has no real output. A *FDEP* consists of a trigger event and one or more depended events. Only when the trigger event is activated will the depended events occur. The gate is shown in Figure 2.9

The trigger event can be a basic event or outputs of any *AND* gate, *OR* gate, or dynamic gate

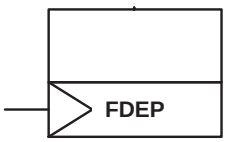


Figure 2.9: FDEP

(*PAND*, *SPARE*) [18].
For Example in Figure 2.11 the *FDEP* models the memory part of the system. There are 2 Memory Interface Units (MIU). Each MIU has one or more memory to interface with. Memories 1 and 2 depend solely on MIU1 and 4 and 5 solely on MIU2. Memory 3 depends on both MIU's. When a MIU i. e. the trigger fails the depended memories fail as well. This is modelled by an *FDEP*. Note that the memories can also fail independently from the MIU.

Cold, Warm, Hot Spare

The *Cold*, *Warm* and *Hot Spare* gates (respectively *CSP*, *WSP* and *HSP*) are identical in appearance. In Figure 2.10 the *CSP*, *WSP* and *HSP* are drawn.

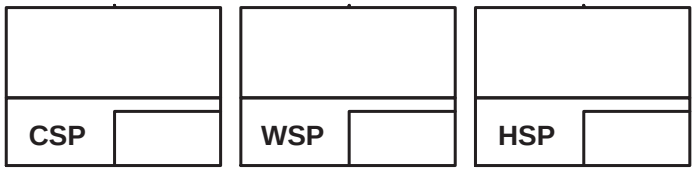


Figure 2.10: Cold SPare gate, Warm SPare gate, Hot SPare gate

A spare gate has only one primary input and has one or more spare inputs. When the primary input fails, available spare inputs are used in order until none are left, at which time the gate fails. The "temperature" of the gate selects the degree to which the failure rate of the spares is attenuated.
In the case of the *CSP* the spare can not fail until the primary input fails, this can be compared with a car. The spare wheel has no tear and wear when in the trunk and is therefore fully functional when needed.
In Figure 2.11 two *CSP*'s are shown. They share a spare component 'A'. This means that when 'A2' fails the cold spare will be used, but 'A1' can no longer use the spare.

The *WSP* spare will fail with a reduced failure rate as compared to the primary. A sort of 'standby' state; it isn't operational, but is standing ready to be activated.

The *HSP* spare has the same failure rate as the primary component. The spare is operational, just like the primary component. The *HSP* will fail when the primary component fails and the spare has failed earlier or when the spare component fails after the primary has failed.

Static Fault Trees are unable to model the order in which components fail. If the bus from the computer system example (Figure 2.1) had a spare bus; the system would only fail if both primary and spare bus fail. So there has to be a construction that would allow the system to switch to the spare bus. However if this switchconstruction fails first, the spare bus cannot be activated although it could still be operational and the system should fail. Static Fault Trees cannot model this behaviour.

An Example of a Dynamic Fault Tree can be seen in Figure 2.11. This Hypothetical Example Computer System (HECS) depends on 4 subsystems. If either one fails, then the entire system fails.
The busses or the operator console are standard (or static) components, respectively an *OR* and *AND* gate. The operator console depends on its operator, its software and its hardware. Either one fails then this subcomponent fails. There are two buses, when both fail, then this subsystem fails.

Each of the two processors shares a spare processor. To share a spare component means that if one component fails and the spare component is available it can be used. However the spare component is no longer available to the other component. If either processor fails then the spare processor kicks in. The spare processor is not active before use, so it can not fail before activation; therefore it is modelled using a cold spare.

The computer system needs at least 2 of its 5 memories. If 3 or more fails, the system fails. The memory units themselves rely on one or two Memory Interface Units (MIU). If MIU1 fails, then this triggers the *FDEP* and consequently the failure of M1 and M2.

To analyse such a system it could be converted to a Markov Chain (MC). This is done by making nodes for each possible state of the system and arrows to indicated each possible transition between these states. In 2.1.3 a small example is converted to a MC. To convert this entire HECS system to a MC would result in an extremely large MC. Therefore current tools, such as Galileo [6] make use of modularization. Modularization means that the Fault Tree is divided into Static and Dynamic independent modules. The Static modules are solved using BDD based techniques. For more information about BDD's see (for example) [14].

Dynamic modules are solved using MC based techniques. Both results are then combined to compute the overall system failure. The next section explains how to create a Markov Chain from a System model and shows an example of a MC.

2.1.3 Markov Chains

A Markov chain consists of states (or nodes) and transitions. The example of the computer system (Figure 2.1 on page 14) could be converted to a MC. The MC (Figure 2.12 on page 20) has one 'start' node. This node (3,2,1) will have only outgoing arrows. The notation (3,2,1) means that the 3 processors, the 2 memories and the 1 bus can still fail. With each transition either component will fail and the corresponding number will decrease. All outgoing arrows (or possible transitions) will go to other possible system states. There are several possible transitions. One of the three

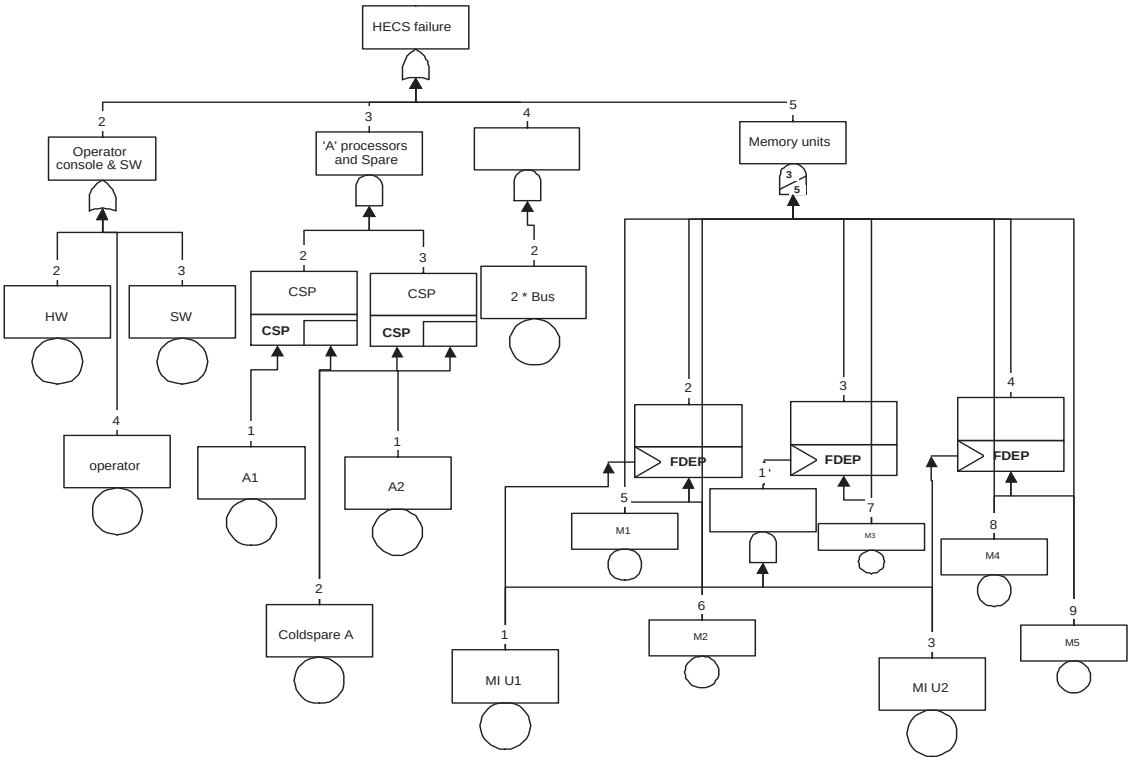


Figure 2.11: Hypothetical Example Computer System (HECS)

processors can fail, this is displayed by the arrow with 3λ. There are 3 processors; each can fail with a failure rate of λ. The Memory can also fail in the start position. There are two memory units, each with a failure rate of μ. The transition with 2 μ displays this possibility. The bus has a failure rate of σ and is displayed by the transition labelled σ. Three 'end' nodes (or absorbing nodes) exist. All three represent valid system failure states. FB, FP and FM which stands for respectively Failure of Bus, Failure of Processor and Failure of Memory. Note that this example is static (i. e. no dynamic components are present). Therefore this example is usually solved with BDD and not, as has been done here, by MC.

To show the importance of the order in which the components fail more clearly, the following figure (Figure 2.13) is the Markov chain of a *PAND* gate with 2 inputs. For convenience the PAND is displayed here again.
From operational state '11' Either 'B' or 'A' can fail and therefore end up in respectively states '10' or '01'. However since the component only fails when first 'A' fails and then 'B' fails, only the upper path will lead to a failure state. Now the importance of the order in which components fail is clearly visible.

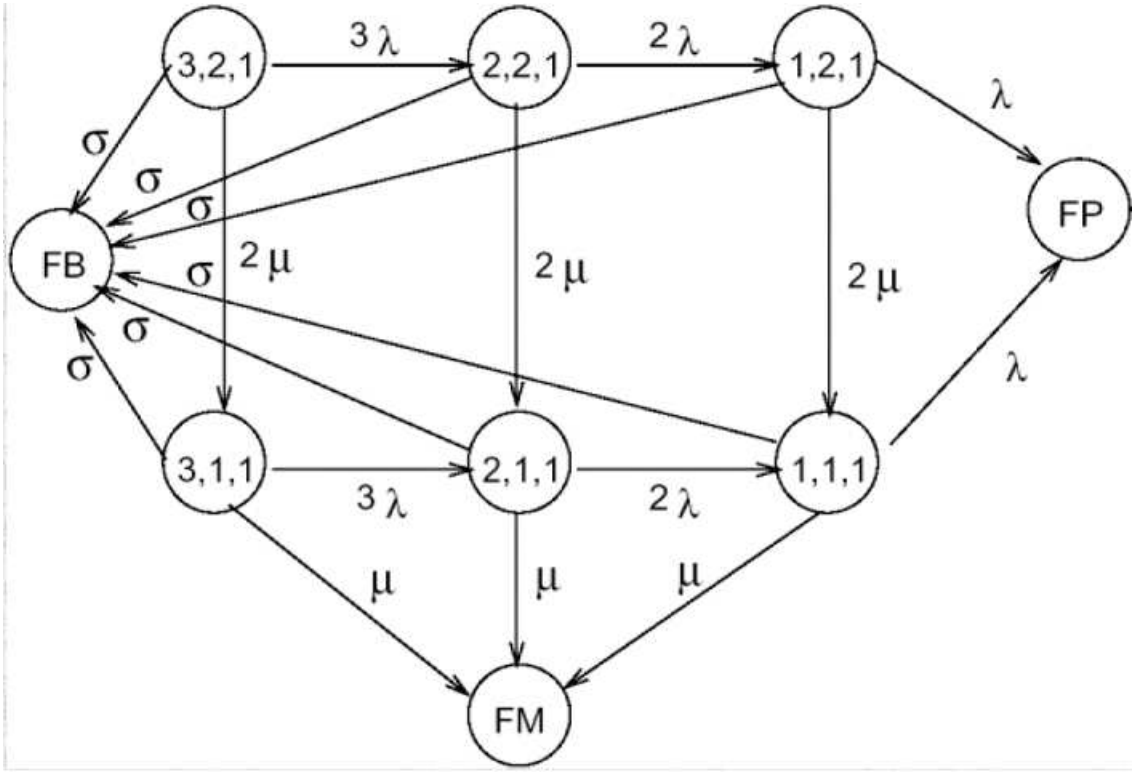


Figure 2.12: Markov Chain of Computer system (3P2M) example

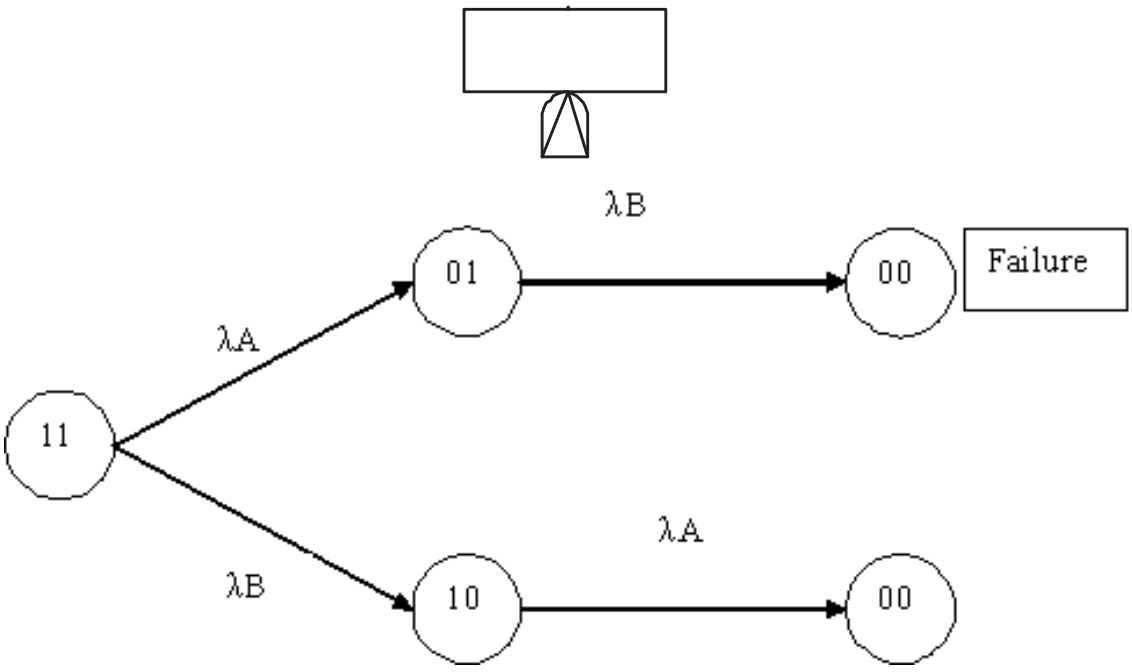


Figure 2.13: A PAND and its corresponding MC

The general algorithm for converting dynamic Fault Trees to Markov Chains is as follows:

- Start with initial state in which all components are operational.
- For each operational state, enumerate all child states by considering effects of one component (call it j) failure at a time
- Establish transition rate from parent to child state as failure rate of component j
- Determine if new state is operational or failed.
- Continue until all operational states are enumerated.

This algorithm has been taken from [9].
Following from this conversion algorithm and the above example it is not difficult to see that the number of states will exponentially grow, thus causing a "state space explosion". Because the example of the computersystem (Figure 2.1) had only six components. Where the FT only needed 3 gates and 3 basic events to display the system. The MC already needs 9 nodes. If there would have been an additional variable, for example, the system is extended with an external HD. The MC would need to grow to accommodate this new variable.
A new approach based on BN's tries to alleviate this state space explosion.

2.2 Bayesian Networks

Bayesian Networks model situations in which causality plays a role but where our understanding of what is actually going on is incomplete, so we need to describe things probabilistically [5].
Bayesian Networks are in fact Directed Acyclic Graphs (DAG). The nodes represent random variables. The edges represent relationships (causality) between variables. If there is an arc from node 'A' to another node 'B', then values of variable 'B' depend directly on values of 'A'. Because of the directionality we call this node 'A' the child of 'B' and 'B' the parent of 'A'. Nodes without parents are root nodes, nodes without children are called leafs. [1]

Some used definitions in the examples are described here first. A BN is a compact representation of a joint probability distribution. The joint probability distribution of two random variables is the likelihood of observing all combinations of the two variables [11]. The values or probabilities for each possible scenario are displayed in Conditional Probability Tables (CPTs) or diagrams (CPD). In Figure 2.14 on page 22 a Bayesian network with CPTs is given.

In the next section we further explain BN through the use of an example. In the example we thoroughly examining the different aspects and calculations typically found in a Bayesian Network.

2.2.1 Bayesian Network example Wetgrass

In Figure 2.14 on page 22 there are 4 'random variables'. These are *Cloudy*, *Sprinkler*, *Rain* and *WetGrass*. (C,S,R,W). The *Cloudy* variable is unaffected by other variable. No arrows are pointing to this node indicating no dependencies. The chance that it is cloudy is 50% true (0.5 on a scale from 0 to 1). The sprinkler is depended on the fact whether it is cloudy or not. If it is cloudy, the chance that the sprinklers are on is quite low. *Cloudy*(C) is **True** and in the column of the probability that *Sprinklers* is **True** show the value 0.1. This means that there is a 10% chance that the sprinklers will be on when it is cloudy. The same goes for rain. The chance that it rains when it is cloudy is greater than when it is not cloudy. The Variable *Wetgrass* is depended on two variables. So a larger CPT is needed to represent all the possible combinations. The chance that the grass is not wet (W=F) when both variables of *Sprinkler* and *Rain* are **True** is quite low, only 0.01.

[1] The edges are usually drawn from bottom to top

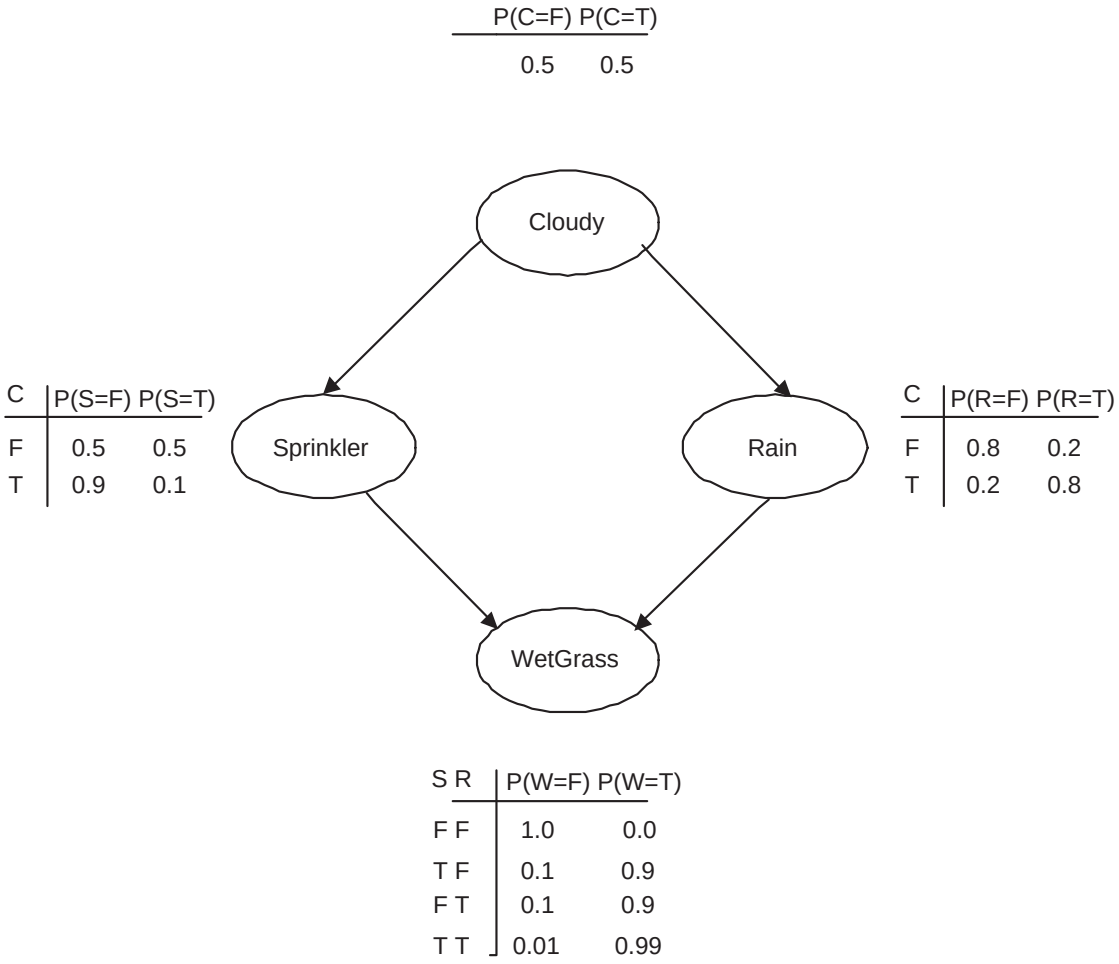


Figure 2.14: Example Discrete Bayesian Network
[3]

During this example we will be using the Bayes Law formula:

$$\mathcal{P}(A, B) = \mathcal{P}(A|B)\mathcal{P}(B) = \mathcal{P}(B|A)\mathcal{P}(A) \tag{2.1}$$

Which generalizes to:

$$\mathcal{P}(A_1, ..., A_n) = \mathcal{P}(A_1)\mathcal{P}(A_2|A_1)...\mathcal{P}(A_n|A_1, ..., A_{n-1}) \tag{2.2}$$

With these formulas and without knowing any of the variables we write down the joint probability of the example. The thing we want to know is the probability that the grass is wet. First the joint probability of all variables. $\mathcal{P}(C, S, R, W) = \mathcal{P}(W, R, S, C)$
$= \mathcal{P}(W|R, S, C)\mathcal{P}(R, S, C)$
$= \mathcal{P}(W|R, S, C)\mathcal{P}(R|S, C)\mathcal{P}(S, C)$
$= \mathcal{P}(W|R, S, C)\mathcal{P}(R|S, C)\mathcal{P}(S|C)\mathcal{P}(C)$
From the Figure we see that **S** and **R** are not directly dependent, the same for **C** and **W**. Thus we get the formula:

$$\mathcal{P}(C, S, R, W) = \mathcal{P}(C)\mathcal{P}(S|C)\mathcal{P}(R|C)\mathcal{P}(W|S, R)$$

The probabilities needed are shown in the tables of Figure 2.14 on page 22. From the joint probability equation we can compute the probability of being in any state of the any node with any given observations (i. e. evidence).

What is the probability then that the grass is wet? $\mathcal{P}(W_T)$ is the probability of *Cloudy* being either **True** or **False**, combined with *Sprinkler* being **True** or **False** and *Rain* **True** or **False**. 3 variables with 2 values, $2^3 = 8$ probabilities.

$$\mathcal{P}(W_T) = \sum_{c\in\{T,F\}} \sum_{s\in\{T,F\}} \sum_{r\in\{T,F\}} \mathcal{P}(C_c, S_s, R_r, W_T)$$

$= \mathcal{P}(C_T, S_T, R_T, W_T)$
$+ \mathcal{P}(C_F, S_T, R_T, W_T)$
$+ \mathcal{P}(C_T, S_F, R_T, W_T)$
$+ \mathcal{P}(C_F, S_F, R_T, W_T)$
$+ \mathcal{P}(C_T, S_T, R_F, W_T)$
$+ \mathcal{P}(C_F, S_T, R_F, W_T)$
$+ \mathcal{P}(C_T, S_F, R_F, W_T)$
$+ \mathcal{P}(C_F, S_F, R_F, W_T)$

This transformed to values we actually have in the CPTs.
$= \mathcal{P}(C_T\mathcal{P}(S_T|C_T)\mathcal{P}(R_T|C_T)\mathcal{P}(W_T|S_T, R_T)$
$+ \mathcal{P}(C_F\mathcal{P}(S_T|C_F)\mathcal{P}(R_T|C_F)\mathcal{P}(W_T|S_T, R_T)$
$+ \mathcal{P}(C_T\mathcal{P}(S_F|C_T)\mathcal{P}(R_T|C_T)\mathcal{P}(W_T|S_F, R_T)$
$+ \mathcal{P}(C_F\mathcal{P}(S_F|C_F)\mathcal{P}(R_T|C_F)\mathcal{P}(W_T|S_F, R_T)$
$+ \mathcal{P}(C_T\mathcal{P}(S_T|C_T)\mathcal{P}(R_F|C_T)\mathcal{P}(W_T|S_T, R_F)$
$+ \mathcal{P}(C_F\mathcal{P}(S_T|C_F)\mathcal{P}(R_F|C_F)\mathcal{P}(W_T|S_T, R_F)$
$+ \mathcal{P}(C_T\mathcal{P}(S_F|C_T)\mathcal{P}(R_F|C_T)\mathcal{P}(W_T|S_F, R_F)$
$+ \mathcal{P}(C_F\mathcal{P}(S_F|C_F)\mathcal{P}(R_F|C_F)\mathcal{P}(W_T|S_F, R_F)$

Now we fill in the numbers.

$$\begin{aligned}
&= 0.5 \cdot 0.1 \cdot 0.8 \cdot 0.99 \\
&+0.5 \cdot 0.5 \cdot 0.2 \cdot 0.99 \\
&+0.5 \cdot 0.9 \cdot 0.8 \cdot 0.9 \\
&+0.5 \cdot 0.5 \cdot 0.2 \cdot 0.9 \\
&+0.5 \cdot 0.1 \cdot 0.2 \cdot 0.9 \\
&+0.5 \cdot 0.5 \cdot 0.8 \cdot 0.9 \\
&+0.5 \cdot 0.9 \cdot 0.2 \cdot 0 \\
&+0.5 \cdot 0.5 \cdot 0.8 \cdot 0 \\
&= 0.6471 = 64.7\%
\end{aligned} \tag{2.3}$$

To read more information about this example see [15].

The probability that the grass is wet without any evidence is 64.7% as calculated in 2.3 The same calculations can be done for the probability that the sprinkler was on. This is the probability that *WetGrass* is **True** and *Sprinkler* is **True**.
$\mathcal{P}(W_T, S_T) = \sum_{c\in T,F} \sum_{r\in T,F} \mathcal{P}(C_c, S_T, R_r, W_T)$
$= \mathcal{P}(C_T, S_T, R_T, W_T)$
$+ \mathcal{P}(C_F, S_T, R_T, W_T)$
$+ \mathcal{P}(C_T, S_T, R_F, W_T)$
$+ \mathcal{P}(C_F, S_T, R_F, W_T)$

This transformed to values we actually have in the CPTs.

$= \mathcal{P}(C_T)\mathcal{P}(S_T|C_T)\mathcal{P}(R_T|C_T)\mathcal{P}(W_T|S_T, R_T)$
$+ \mathcal{P}(C_F)\mathcal{P}(S_T|C_F)\mathcal{P}(R_T|C_F)\mathcal{P}(W_T|S_T, R_T)$
$+ \mathcal{P}(C_T)\mathcal{P}(S_T|C_T)\mathcal{P}(R_F|C_T)\mathcal{P}(W_T|S_T, R_F)$
$+ \mathcal{P}(C_F)\mathcal{P}(S_T|C_F)\mathcal{P}(R_F|C_F)\mathcal{P}(W_T|S_T, R_F)$

$$\begin{aligned} &= 0.5 \cdot 0.1 \cdot 0.8 \cdot 0.99 \\ &+0.5 \cdot 0.5 \cdot 0.2 \cdot 0.99 \\ &+0.5 \cdot 0.1 \cdot 0.2 \cdot 0.9 \\ &+0.5 \cdot 0.5 \cdot 0.8 \cdot 0.9 \\ &= 0.2781 \end{aligned} \tag{2.4}$$

This is the probability that both the sprinkler is on and the grass is wet.

However, we wanted to know the probability of the sprinkler to be on with only the wet grass as evidence. To use this result we need Bayes's Law which is:

$$\mathcal{P}(H|E) = \frac{\mathcal{P}(H) \cdot \mathcal{P}(E|H)}{P(E)}$$

From this follows that

$$\mathcal{P}(H|E) = \frac{\mathcal{P}(H, E)}{P(E)}$$

We want to know is the probability of *Sprinkler* being **True** given the evidence that *WetGrass* is **True**.

$$\mathcal{P}(S_T|W_T) = \frac{\mathcal{P}(S_T, W_T)}{\mathcal{P}(W_T)} = \frac{\sum_{c\in\{T,F\}} \sum_{r\in\{T,F\}} \mathcal{P}(C_c, S_T, R_r, W_T)}{\mathcal{P}(W_T)}$$

$$\mathcal{P}(S_T|W_T) = \frac{\mathcal{P}(S_T, W_T)}{\mathcal{P}(W_T)} = \frac{\text{answer from 2.4}}{\text{answer from 2.3}} = \frac{0.2781}{0.6471} = 0.4298 \tag{2.5}$$

The probability that the Sprinkler was on, when you know that the grass is wet is 43%.

2.2.2 Discrete Time Bayesian Network

The previous sections did not discuss any notion of time. The concept of time is incorporated in a version of Bayesian Networks called the Discrete Time Bayesian Network (DTBN). Recall that a node represents the failure of a basic component (or gate) to occur. We now add the concept of the component failing in timeinterval x.
We define a mission time T and divide the time from 0 to T into n intervals where each interval represents a certain system state.
This n is called *time granularity*. This is illustrated in Figure 2.15.

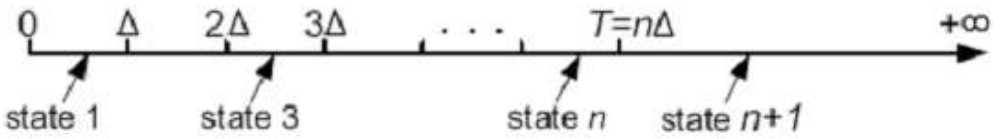


Figure 2.15: Correlation between states and intervals
[4]

The first n states represent the failure of a specific component in one of the n first time intervals (i. e. , during the mission time); and the last state n+1 represents the failure of this component after mission time [4].

The fact that a component is in State n means that the basic component (or gate) has failed in that time period. In Figure 2.15 for a component to be in state 3 means that the component has failed in time interval $[2\Delta, 3\Delta]$. n times this Δ is the misiontime T, in formula $\Delta = \frac{T}{n}$, see Figure 2.15.

Static gates have no notion of order, you can divide the mission time, but the order in which subcomponents fail does not matter. The static gate will fail if the proper amount of subcomponents has failed within mission time. This attribute takes care that the solution is still exact even when not dividing the mission time (e. g. $'n' = 1, T = 1\Delta$). This creates only two states in a DTBN. Time periods $[0.T]$ and (T, ∞).

- State 1 means the component fails during mission time interval $[0, T]$.
- State 2 means the component fails in interval (T, ∞)

For example the static component *AND* (Figure 2.3 on page 15 in Chapter 2.1.1), with inputs A and B, will only fail when both subcomponents fail during mission time (meaning that both inputs will be in state 1).

A dynamic component is more elaborate because the order of failure is important. For instance the *PAND* (Chapter 2.1.2 on page 17), with inputs A and B, will only fail if A fails before B and within mission time. The mission time must be divided in more than one interval. This means that n must be greater than 1.
This is illustrated in table 2.1.
Here a *PAND*'s CPT is given. inputs A and B and a n of 3 make up for four states per component.

A		State 1 $[0, \Delta]$				State 2 $(\Delta, 2\Delta]$				State 3 $(2\Delta, T]$				State 4 (T, ∞)			
B		State 1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
PAND	State 1	1															
	State 2		1				1										
	State 3			1				1				1					
	State 4				1	1			1	1	1		1	1	1	1	1

Table 2.1: *PAND* probability table

There are 4 states. State 1,2 and 3 divide the mission time in three. State 1 is $[0, \Delta]$, State 2 is $(\Delta, 2\Delta]$. State 3 is $(2\Delta, T]$, State 4 is outside mission time (T, ∞).
Because A must fail before B fails in order for the *PAND* to fail there are several possibilities where the PAND will not fail. Because , for example, if A is in state 3 (this meant that A failed in the time period of $(2\Delta, T]$) the *PAND* will now only fail when B fails after A (e. g. state 4, and not states 1 and 2). The *PAND* will always fail at the same time as B (only when A has failed as well). Thus when A fails in state 3 and B fails in the same time interval, the *PAND* will fail in this same interval.
All possibilities where the *PAND* will fail are written with a '1' in the table. All other combinations will not cause the *PAND* to fail. Therefore in the table all empty squares are actually filled with zeros ('0'). The light Gray coloured states are those states where both A and B fail in the same time interval. The convention is to let the *PAND* fire as well, although theoretically B could have fired before A. State 4 is a special state. Because the probabilities always add up to 1 there are a number of ones that are somewhat counter-intuitive. When the *PAND* doesn't fire within mission time it will fire *at infinitum* (i. e. state 4, which included the infinity).
Instead of describing every single state using a large table, we can use a simpler way to describe which instances the gate fires. Through the use of a Probability Mass Function (PMF). The CPT of the *PAND* in table 2.1 can be displayed as a conditional (PMF).
This function gives the probability that a discrete random variable is exactly equal to some value. A probability mass function differs from a probability density function in that the values of the latter, defined only for continuous random variables, are not probabilities; rather, its integral over

a set of possible values of the random variable is a probability [17]. For example if X is a discrete random variable, taking values on some countable sample space $S \subseteq R$. Then the probability mass function $f_x(x)$ for X is given by:

$$f_X(x) = \begin{cases} \mathcal{P}(X = x), & x \in S, \\ 0, & x \in \mathbb{R} \backslash S. \end{cases}$$

in 2.6 the CPT of the *PAND* is given in the form of a PMF.

$$f_{X|A,B}(x|a,b) = \mathcal{P}(X = x|A = a, B = b) = \begin{cases} 1 & if\ a \leq b, x = b, \quad (1) \\ 1 & if\ a > b, x = n+1, \quad (2) \\ 0 & otherwise, \quad (3) \end{cases} \tag{2.6}$$

This formula represents the same result as in the table. The *PAND* will only fire when A has failed in a earlier (or equal) time interval as B and where the current time interval is the same as the interval where B failed. This is in words what the line (1) means.
Line (2) means that when A fails after B, the *PAND* will only fire in time interval $n+1$. Which is the last interval from T to ∞.
The dark Gray coloured cells in the table means that there the rule on line (2) has been applied. The third line (3) indicates that all other scenarios will be 0 (i. e. the *PAND* will not fire in those possibilities).

A *PAND* will always immediately fire when B fires with the only condition that A has fired previously. There are no delays that would cause the *PAND* to fire in a later time interval.

A CSP doe not behave similarly. The CSP only fails when the primary fails followed by a failure of the spare. See Chapter 2.1.2.

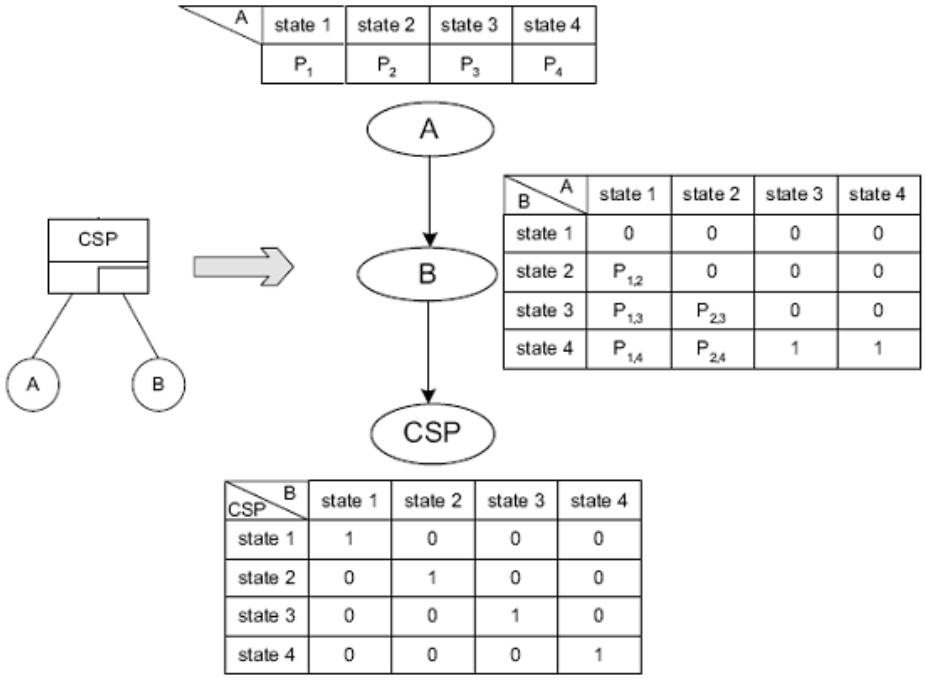


Figure 2.16: CSP and equivalent BN

In Figure 2.16 the CSP is given. In the CPT of node B there are 5 places in the table where the probability isn't 1 or 0. Also in the CPT of state A the values are not 1 or 0.

But how do we calculate these probabilities?

The possibility that the system fails on a timescale from 0 through infinity is always 1 (100%). You know the system will eventually fail. When the input B has failed in state 1, the values for the CSP in all four states will be equal to 1. This is the case because the CSP fails at the same time as input B fails.
If primary input A fails in state 1, spare B can fail in state 2, three or four. However because the system eventually fails, those three numbers (indicated by $P_{1,2}$,$P_{1,3}$,$P_{1,4}$) must equal 1. The probabilities in Figure 2.16 are derived from a failure distribution F. The probability of an event X to occur at time x (x is the statenumber) is the probability of this event to occur in time period $(x-1)\Delta$ to $x\Delta$.
If during missiontime we subtract the failure distribution of period $(x-1)\Delta$ from period $x\Delta$ then we end up with the failure distribution of the period in which event X occurs. The failure distribution of the infinite period (e. g. timeperiod n+1 or outside mission time) is 1. For the probability in this timeinterval we substract the failure distribution of mission time period from 1.
In a formula:

$$\begin{cases} \mathcal{P}(X = x) = \mathcal{P}((x-1)\Delta < X \leq x\Delta) = F(x\Delta) - F((x-1)\Delta) & for\ 1 \leq x \leq n \\ \mathcal{P}(X = n+1) = \mathcal{P}(T < X \leq \infty) = 1 - F(T) & otherwise \end{cases} \tag{2.7}$$

If we now want to calculate the probability of A being in state 2, we start with filling in 2.7

$$\mathcal{P}(A = 2) = \mathcal{P}(\Delta < A \leq 2\Delta) = F(2\Delta) - F(\Delta) \tag{2.8}$$

Now we need the failure distribution function. We assume for now that the subcomponents of the CSP are exponentially distributed. Failure of the components is distributed over the mission time as in Figure 2.17 on page 28.

The equation now takes in the following form:

$$(1 - e^{-\lambda \cdot 2 \cdot \Delta}) - (1 - e^{-\lambda \cdot \Delta})$$

(Remember that $\Delta = \frac{T}{n}$)
The λ represents the component failure rate. Now to fill in equation 2.8. We take missiontime 1 second $\Delta = \frac{T}{n} = \frac{1}{3}$, we set the lambda of the inputs to 0.01.

$$(1 - e^{-0.01 \cdot 2 \cdot \frac{1}{3}}) - (1 - e^{-0.01 \cdot \frac{1}{3}}) \tag{2.9}$$

0.006644-0.003327=0.003317. $P_2 = 0.003317$ The probability of input A failing in state 2 is 0.003317.

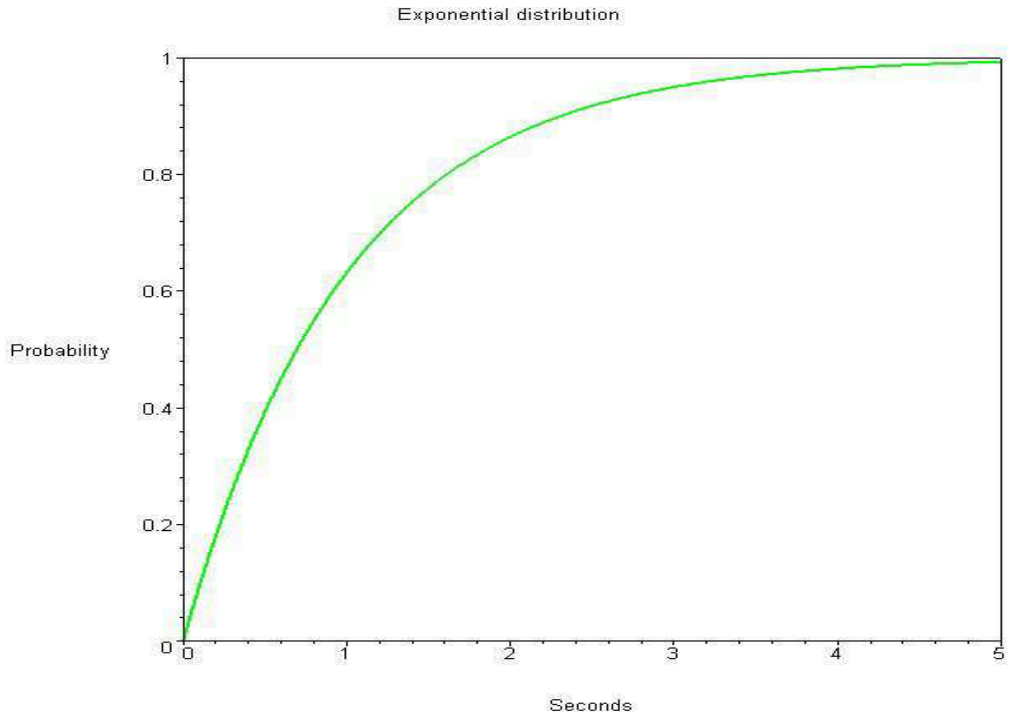


Figure 2.17: Exponential distribution

Chapter 3

Design and Implementation

The tool IDIC allows the user to draw DFTs via a GUI and to analyse these DFTs via a conversion to BN. IDIC consists of four main parts.

1. GUI
2. Compiler
3. (BN to SMILE) translator
4. BN analyser

These main parts are visualised in Figure 3.1 by the boxes.
The GUI is the central part of IDIC. Here the FT is drawn and the conversion to BN is started and the results of the analysis are viewed here. The next part of IDIC is the compiler. The compiler takes a textfile which is saved from the GUI and compiles this to a standard Bayesian Network. This network is the input for the next part; The BN to SMILE translator. The translator will transform the standard BN to a BN in SMILE format. This SMILE format is input to the BN analyser. The analyser will analyse this BN network and deliver probability data to the GUI. This probability data is the unreliability of the system.

The overview of the tool is depicted in Figure 3.1

Here the 4 main parts are visible in this overview. The subcomponents and intermediary deliverables are also in this schema. The next sections will be overviews of the individual parts visible in the figure. The sections are given also in the following list.

- Galileo 3.1
- DFT in Galileo format 3.2
- Compiler 3.3
  - ANTLR 3.3.1
  - Lexer 3.3.2
  - Parser 3.3.3
  - ANTRL tree 3.3.4
  - Treewalker 3.3.5
- Standard Bayesian Network 3.4
- Translator to SMILE format 3.5
- BN in SMILE format 3.6

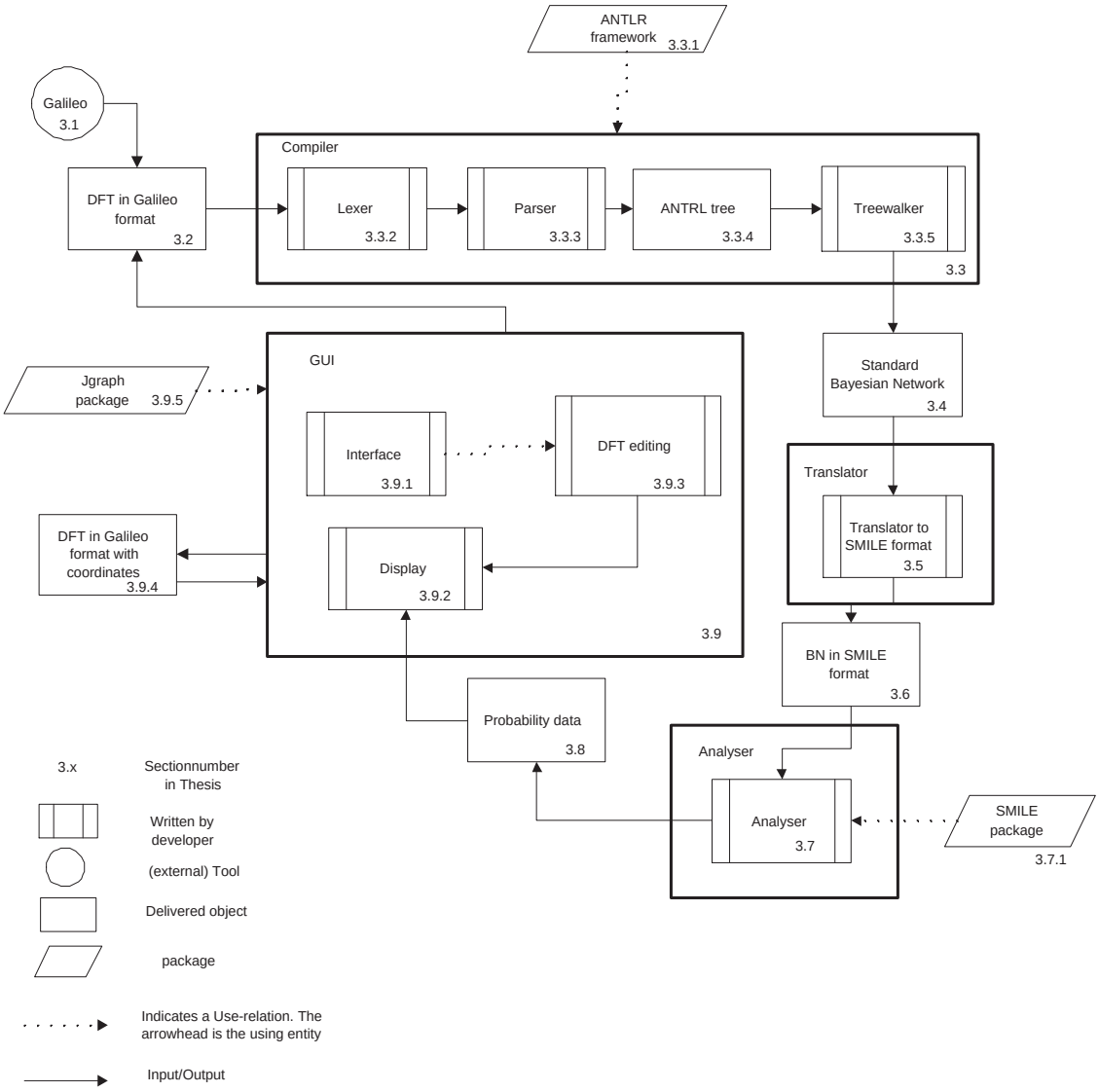


Figure 3.1: Overview

- BN Analyser 3.7
  - SMILE package 3.7.1
- Probability Data 3.8
- GUI 3.9
  - interface 3.9.1
  - Display 3.9.2
  - DFT editing 3.9.3
  - DFT in Galileo format with coordinates 3.9.4
  - JGRAPH 3.9.5

The implementation is written in JAVA and uses the programming environment Eclipse[7] in which the used packages were combined to form the final tool.

We will be using an example throughout this chapter. It will be a DFT consisting of an *AND* gate with two children. This is depicted in Figure 3.2. The left basic event has a discrete

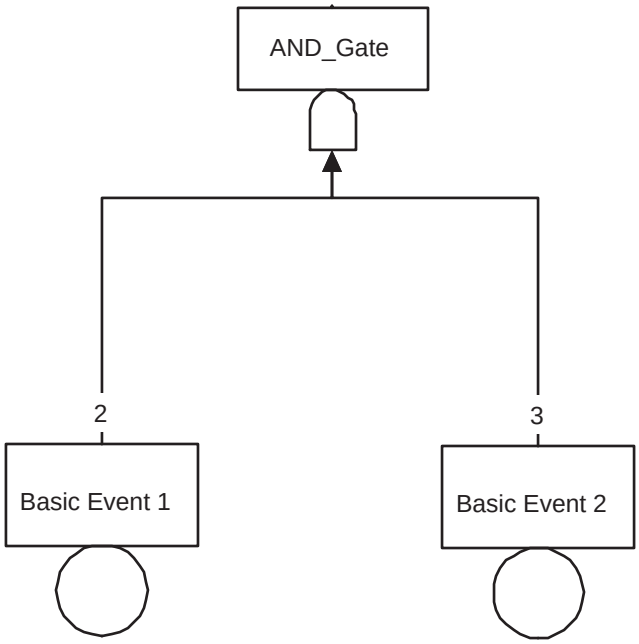


Figure 3.2: Running example, *AND* with 2 events

probability of 0.2 and a dormancy of 0.5, the right basic event has a discrete probability of 0.5 and a dormancy of 0.5. Because Galileo uses a standard dormancy of 0.5 and IDIC a standard dormancy of 0.0, this property must be set to get a consistent example through all the phases of the program and thus all the sections of this chapter.

3.1 Galileo

Galileo is a dynamic fault tree analysis tool used by engineers to model and analyse the reliability of a system. Fault trees are models of system failure, where particular combinations and orders can cause overall system failure [6]. One can either create the FT textually in Galileo with help of MS Word or visually with help of MS Visio. Galileo uses BDDs for static Fault Trees and MCs to solve Dynamic Fault Trees.

3.2 DFT in Galileo format

The start of the conversion process to a BN starts with the FT. To be precise with a textfile containing the FT. Therefore the FT needs be of a certain format and written to this textfile. The format chosen is Galileo. This format is easily read by human eye and is easy to produce. The creation of this file will be discussed in the GUI section 3.9. To visualize this format a simple FT is chosen. An *AND* gate called “AND_Gate” and two discrete basic event (“basic event 1” and “basic event 2”). One has probability of 0.2 and the other 0.5. Both gates will have a dormancy of 0.5(our running example Figure 3.2.

```
1  toplevel "AND_Gate";
2  "AND_Gate" and "Basic event 1" "Basic Event 2";
3  "Basic Event 2"
4   phase 1 prob=0.2 cov=1 res=0 repl=1 dorm=0.5;
5  "Basic event 1"
6   phase 1 prob=0.5 cov=1 res=0 repl=1 dorm=0.5;
7
8  page 1 "AND_Gate";
9  page 1 "Basic Event 2";
10 page 1 "Basic event 1";
```

The gate that is the top of the tree is called toplevel. This gate is mentioned separately in the textfile. The word toplevel followed by the name is listed first, line 1. Then all of the connections are listed. In this example there is only one and it is the gate called "AND_Gate". It is of type *AND* and has two children, line 2. Once all connections are listed, the basic events that are in the tree are listed with their properties. Some properties are used by the conversion, others are not. *Phase*, *cov*, *res* and *repl* are not used. *prob* and *dorm* in this case are. *prob* stands for (discrete) probability and *dorm* stands for dormancy. The next part of this format are page listings, lines 8 through 10, these are needed by Galileo to enable the file to be read by Galileo itself and is used by Galileo's layout algorithm. This Galileo formatted file is the input for the compiler.

3.3 Compiler

The compiler has as input the textfile with the DFT in Galileo format. It outputs a Bayesian network in a standardized format. The compiler needs a way to determine what nodes there are in the file, which basic events are present and which properties these have. Also the tree structure, that is the connections between the gates and events, needs to be represented. These are all needed to be able to output a BN.
For this task the ANTLR framework is used. This program provides a framework in which a lexer (to read the textfile), a parser (to represent the gates, basic events etc. that are in the tree) and a treewalker (to make the BN) can be easily created. These individual parts are now discussed in separate sections.

3.3.1 ANTLR framework

ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting. [13]

ANTLR provides support for tree construction. The input is a file, which is read. Predefined portions are identified as 'tokens'. This part, called a lexer, is used to construct the tokens. Predefined keywords are tokens. IDIC uses for instance *AND*, *prob* as keywords.
Comments in the style of programming with the \\ or *\,* can also be identified. These can be ignored or handled differently.

The second part, the parser, constructs the actual tree. Every possible occurrence of a token must be handled here. For example, a token indicating a name of a node can be followed by a type and one or more strings indicating its children. The constructed tree is therefore a node with its children as actual children. This token can also be followed by nothing (to be precise with a end-of-line character). This indicates that the token represents the name of a basic event and the next line will be the properties of this event.

The last part of the ANTLR package is the treewalker. The treewalker starts at the top of

the tree. At every node the treewalker looks and determines what action to take. IDIC needs to make a Bayesian Network, so every new gate it encounters will result in the creation of a new BN node.

3.3.2 Lexer

The lexer has as input a DFT formatted textfile. After the lexer it should output a large stream of tokens. One of the tokens is a STRING. Everything between a " and " including multiple words is called a String. This is needed because a name of a node is always between " and " and can consist of several words. Everytime the lexer encounters a " it will put everything that follows in one token, until it encounters the ".
The textfile of the example Figure 3.2 looks like this:

```
toplevel "AND_Gate";
"AND_Gate" and "Basic event 1" "Basic Event 2";
"Basic Event 2"
 phase 1 prob=0.2 cov=1 res=0 repl=1 dorm=0.5;
"Basic event 1"
 phase 1 prob=0.5 cov=1 res=0 repl=1 dorm=0.5;

page 1 "AND_Gate";
page 1 "Basic Event 2";
page 1 "Basic event 1";
```

In the end it will pass its result to the parser. Because the lexer reads the file and makes tokens out of the individual letters and characters the result will look like this for the first two lines:

```
toplevel , STRING, SEMICOLON, EndOfline ,
STRING, AND, STRING, STRING, SEMICOLON, EndOfLine
```

Ofcourse these tokens have a field where the actual value is stored, for the first STRING this will be "AND_Gate".
All the lexer does is grouping several characters together. It has predefined words, if recognised they will be identified as such. Example is the *AND*.
Every word between quotes is the name of a gate or basic event and therefore it is grouped into a token called STRING.
The special characters EoL (End of Line) and EoF (end of File) are also tokenized for use by the parser. This stream of tokens is then passed on to the parser.

3.3.3 Parser

The parser receives the stream of tokens from the lexer. The task for the parser is to group these tokens in a tree in order for the treewalker to use the tree for the creation of the BN. The tree of our running example Figure 3.2 is displayed in Figure 3.3 The structure of the original tree is visible in the tree. The topgate is called "And_Gate". Next line we see this And_Gate and it has two children, called "Basic Event 1" and "Basic Event 2". This is exactly as the original FT. Next line we have the name Basic Event 2, directly followed by its properties. We see 'prob' (which stands for probability) and the number 0.2 which is the discrete probability we had given to this event. The last three children of this property branch is dorm, '=' and 0.5. The dormancy is 0.5.

This tree is very simply made. Just read the stream of tokens you get from the lexer and determine how to build your tree. If the token is a STRING followed by an *AND* then you build the tree with the STRING as the parent. The two STRINGS that follow the *AND* token will become its children. If the token following STRING is a EoL character, then the next token will be the token PHASE. PHASE is always followed by a NUMBER, so NUMBER is its first child,

Figure 3.3: ANTLR Tree

the second child will become an other subtree. The root of this tree is either PROB, LAMBDA, or RATE. In all cases the value that follows will become the first child and the token COV the last child. All tokens that follow COV will become its children. The PAGE tokens that Galileo adds to its files will be added at the end of the tree. Presently they are not used, but a possible future layout algorithm could use this data. The output of the parser is now an ANTLR tree.

3.3.4 ANTLR tree

The tree displayed in Figure 3.3 is the tree delivered by the parser and input to the treewalker. This tree is always the same format. First the Toplevel with as child the name of this toplevel. In this case "AND_Gate". Followed by all the gates in the original FT. In the example it is only one gate; the *AND* gate called "AND_Gate". The first leaf is that of the name "AND_Gate" followed by a branch. The root of this branch is of type *AND* and has two children called "Basic Event 1" and "Basic Event 2". Each node is put in the tree as root followed by a branch. For gates this

means a node of the type of the gate followed by its children.
For a basic event this branch begins with a node called phase, followed by a number and then either *prob* for (discrete) probability, *lambda* for exponential and *rate* for Weibull probability. This is then followed by a *equals* node followed by the actual value. This goes one for all the properties until the last node denoting the value of the dormancy.
After the toplevel, all the gates, all the events only the page attributes are left at the bottom of the tree.

This ANTLR tree is now passed on to the treewalker.

3.3.5 Treewalker

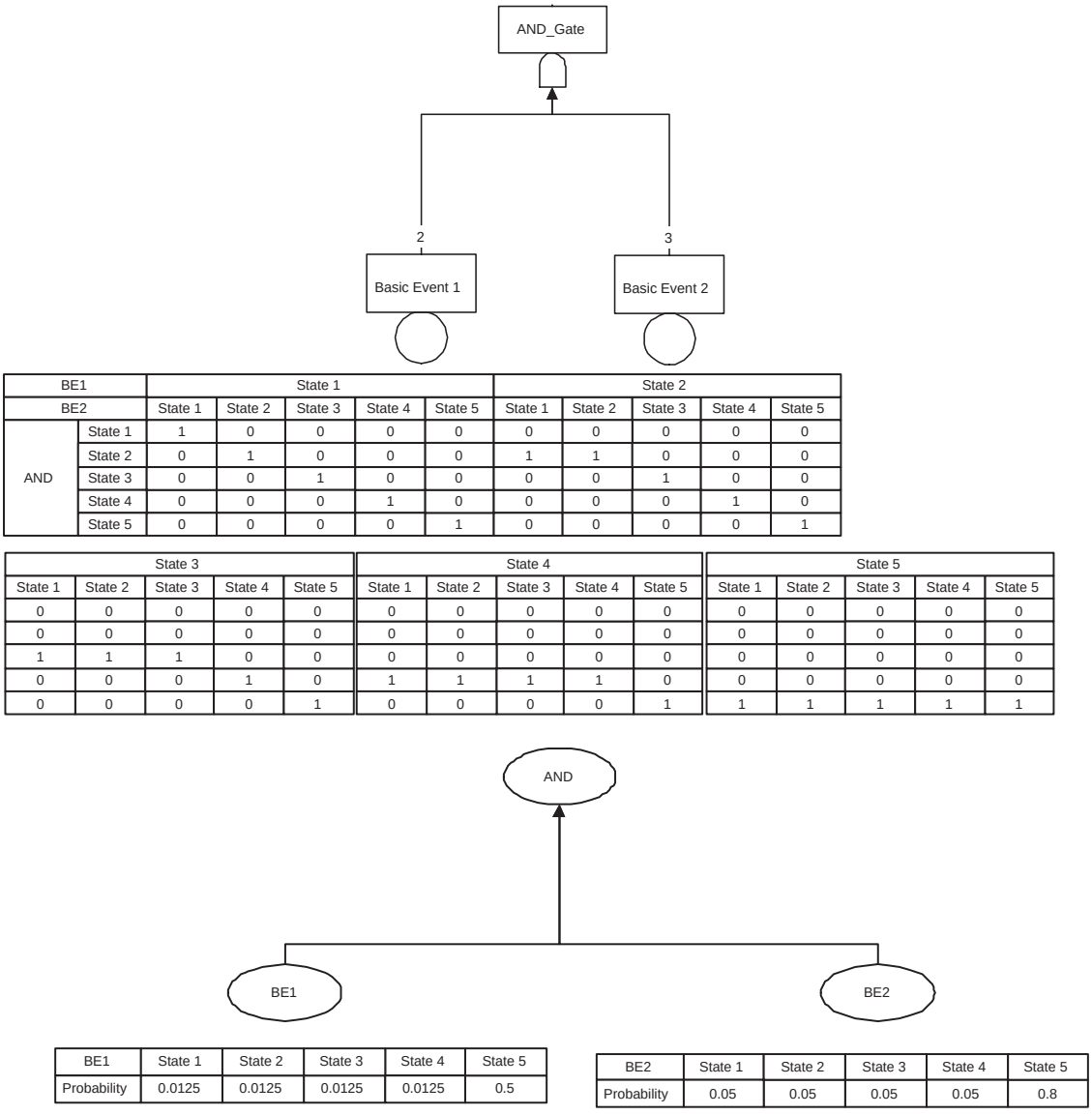
The treewalker will receive the ANTLR tree from the parser. Its job will be to create a Bayesian Network from it. With a treewalker the process starts at the top of the tree and works its way down through every branch and leaf. It is up to the treewalker what actions will occur when visiting the branches and leaves of this tree.
The toplevelname is saved for futher use. We are only interested in the gates themselves and the connection between them. The FT in Figure 3.2 consisted of an *AND* gate and two basic events. This example is displayed in Figure 3.4 with its BN equivalent.This is what must be created by the treewalker.
For construction of the BN there are functions and classes written. One class actually creates the network. It adds nodes, edges and the CPTs and it stores them in a hashmap [16]. A second class controls the nodes themselves. It will add the children to this node and most important it will create the CPTs.
The *AND* looks simular in FT as it does in BN format. Simply add a node to the BN with type *AND*, create its children, add connections between parent and children and finally add a CPT to the *AND* and its children.
The stored node will be stored under its name and have the type, the names of the two children and a CPT within.
If our *AND* gate would have had multiple children at this point, they would have been cascaded into multiple *AND* gates with maximum of two children.
The two basic events are created the same way. Because basic events are always children of a gate, they usually already exist and don't have children, so only the CPT needs to be added. A basic event has only one line in its CPT. The values are the discrete probability divided by the number of states during missiontime. The value substracted from 1 is the probability for outside missiontime (n+1).

```
1 for (int i=0; i<statenumber-1;i++) {
2   aBasicEventdiscDef[i]= prob/(statenumber-1) ;
3     interresult=interresult+aBasicEventdiscDef[i];
4 }
5   aBasicEventdiscDef[statenumber-1]=(1.0-interresult);
```

A simple for loop fills a array with this value, line 2 and will calculate the remaining probability for time n+1, line 5. This is done by subtracting all the values from within missiontime, line 3, from 1. After all the individual gates and basic events are handled the BN generation is finished.

3.4 Standard Bayesian Network

The standard BN is simply a list of nodes. Every node has a field for the name of the node, the type of the gate, an array containing the CPT, the number of states of which the BN is build up from and two fields for each of its two children.
The *AND* gate from the example will be stored like this:
The first item is the name of the node, the second item is the CPT of the BN with four states



BE1		State 1					State 2				
BE2		State 1	State 2	State 3	State 4	State 5	State 1	State 2	State 3	State 4	State 5
AND	State 1	1	0	0	0	0	0	0	0	0	0
	State 2	0	1	0	0	0	1	1	0	0	0
	State 3	0	0	1	0	0	0	0	1	0	0
	State 4	0	0	0	1	0	0	0	0	1	0
	State 5	0	0	0	0	1	0	0	0	0	1

State 3					State 4					State 5				
State 1	State 2	State 3	State 4	State 5	State 1	State 2	State 3	State 4	State 5	State 1	State 2	State 3	State 4	State 5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1	1	1	1	1

BE1	State 1	State 2	State 3	State 4	State 5
Probability	0.0125	0.0125	0.0125	0.0125	0.5

BE2	State 1	State 2	State 3	State 4	State 5
Probability	0.05	0.05	0.05	0.05	0.8

Figure 3.4: The example *AND* gate and its corresponding BN

Field	contents
Name	AND_Gate
CPT	1,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,1,0,0,0, 1,1,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,1,0,0, 0,0,1,0,0, 1,1,1,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,1,0, 0,0,0,1,0, 0,0,0,1,0, 1,1,1,1,0, 0,0,0,0,0, 0,0,0,0,1, 0,0,0,0,1, 0,0,0,0,1, 0,0,0,0,1, 1,1,1,1,1
statenumber	4
child 1	Basic Event 1
child 2	Basic Event 2

Table 3.1: AND_Gate stored

during mission time and a fifth outside mission time. The number of states during mission time is stored in the next item.
The last two items are its children or "none" when the node has no children, usually the basic events.
The two basic events are:

	Basic Event 1	Basic Event 2
Field	contents	
Name	Basic Event 1	Basic Event 2
CPT	0.125, 0.125, 0.125, 0.125, 0.5	0.05, 0.05, 0.05, 0.05, 0.8
statenumber	4	4
child 1	none	none
child 2	none	none

Table 3.2: Basic events stored

3.5 Translator to SMILE format

The input for this translator is the BN from section 3.4. The output will be a BN in a format understood by the package that will analyse the BN. This package is called SMILE, see section 3.7.1. This package contains 3 functions needed to create the BN in its own format. These functions are *addNode*, *addArc* and *setNodeDefinition*, respectively to add a node, add an arc and add a CPT. The *addNode* function requires a name and a number of states to be able to add a node to the new Smile formatted network. Both are fields in the node stored in the BN network. So we just need a list of all the nodes, go over this list and just add the nodes you encounter. This is line 5 in below code fragment.
Each Node has its own children as fields. When the node itself is made, each of its children is also created. This is done because an arc can only be created between two nodes who already exist in the network. Once a child is created, the function *addArc* is called. This function requires nothing more than the names of the two nodes between the arc needs to be drawn, lines 9 and 14.
The CPT of a node can only be added if the node itself exists and if there is a arc between the node and its two children (if present). This is because the SMILE package will have a internal representation of the CPT. This representation is initially the same size as the number of states as indicated upon creation. When a child is added, the CPT needs to be bigger to account for the states of the child as well. So only when both childs have been added to the network will the gate have enough room to hold the entire CPT.
This is the last step of the translator, add the CPT. This function *setNodeDefinition* requires the name of the node to which to add the CPT and a double array containing all the values. This

happens to be the exact format as the CPT has in the BN network. This is done on line 16.

```
1  private void adding(BNNetwork vnet, String topLevelName, Node n) {
2  ...
3      name=n.getName();
4      nrOfStates=n.getnrOfStates();
5      addNode (name, nrOfStates);
6      child1=n.getChild1();
7      if (child1!="None") {
8        addNode(child1 ,nrOfStates);
9        addArc(child1 ,name);
10     }
11     child2=n.getChild2();
12     if (child2!="None") {
13       addNode(child2 ,nrOfStates);
14       addArc(child2 ,name);
15     }
16     net.setNodeDefinition(name, n.getCPT());
17 ...
```

The arguments for this function are the original BN network, the name of the toplevel (needed later for the analyses) and the current node.
The resulting BN is now translated into the SMILE format.

3.6 BN in SMILE format

The BN in SMILE format is not stored as a list of nodes, but instead it is stored as a network. This network is an internal representation of SMILE, but the network is still the same BN as in Figure 3.4. This network is then passed to the analyser.

3.7 BN Analyser

The analyser has as input the BN in SMILE format. It analyses the network and creates the probability data the user wants. The first step is actually analyse the network. SMILE will analyse the network and store the results for all the nodes internally. The analysis is done using the function *updateBeliefs()*. Now the results are prepared in the network with the function *getOutcomeIds(name)* The name argument is the name of the node from which you want the results. Currently it is always the topnode. A array is used to retrieve the data from the network using the formula *getNodeValue(name)*. The code below is this first part of the analyser.

```
1  public String prob(String name, Integer n, Network net) {
2      result="";
3      net.updateBeliefs();
4      net.getOutcomeIds(name);
5      double[] aValues = net.getNodeValue(name);
```

The second part is to build a certain format in which to represent the results. Using a loop all the values are taken from the array of values and added to a resultstring along with a few index-markers, line 4. The unreliability of the system is added to this string, line 6. The format of this probability data taken from the running example is given in section 3.8.

```
1       result = result + ("Toplevelname: " + name + " \n");
2       for (int index =0; index<(n); index++) {
3         double prob = aValues[index];
4         result= result + ("Index"+index + " Prob: "+ prob+" \n");
5       }
6       result=result+(1.0-aValues[n-1]);
7       return result;
8   }
```

The resulting data is passed to the GUI. An example of this data is in section 3.8. The functions used are from the SMILE package.

3.7.1 SMILE package

SMILE (Structural Modelling, Inference, and Learning Engine) is a fully portable library of C++ classes implementing graphical decision-theoretic methods, such as Bayesian networks and influence diagrams, directly amenable to inclusion in intelligent systems.[8]
The SMILE package is used to calculate the probabilities. Its input is a Bayesian network. There are many different variations possible with this package. For now, this program only utilizes the probability of the top node (usually the full, or composed system) within a Bayesian Network.

The SMILE package requires a network or a file as input, as well as the name of the node of which you want the results. SMILE requires a specific format with Doublearrays for the CPT values.
The tool saves the resulting Bayesian network as a GeNIe-formatted file before passing it to the SMILE package for analysis.
GeNIe is SMILE's windows interface and is a versatile and user-friendly development environment for graphical decision-theoretic models[8]

3.8 Probability data

The probability data is the output from the analyser. It is a String which can be written to standard output, printer, file or a part of a GUI. Here the data is inserted into a specific field of the GUI.
The running example Figure 3.2 is executed and the results are given in the fragment below.

```
New conversion:
Toplevelname: node0
Index0 Prob: 0.00625
Index1 Prob: 0.01875
Index2 Prob: 0.03125
Index3 Prob: 0.04375
Index4 Prob: 0.9
0.1
```

The data is set up as follows. The first line is to indicate a new conversion has started, this is done because the field in the GUI can hold several conversions after each other.
The toplevelname is given, because this is the node of which the results are requested. The following lines are the actual results. For each of the states (0 until n+1) the result is displayed. For state 0, the chance that the system will fail is 0.00625. For state n+1 (outside missiontime) it is 0.9.
The chance that the system will fail outside missiontime was 0.9, therefore, the chance that the system would fail within missiontime is 0.1. The last number is not 1 minus the 0.9, but the

individual results from all the states within missiontime added up. which is 0.1. This entire string is directly inserted in a textfield of the GUI.

3.9 GUI

The Graphical User Interface (GUI) allows the user to draw a Fault Tree and customize it. The main part of the GUI is a central drawing area. The nodes are inserted into this area through the use of buttons marked with the type of node to add.
For the basic events, a drop down choice box is present. After selecting the desired type, the button can be pressed.
Also present are a few fields to the right of the drawing area. They present info about the current selected node, like name, type and if applicable values like lambda, alpha or rate. As can be seen in Figure 3.5 Basic Event 1 is currently selected. These values can also be edited and changed.

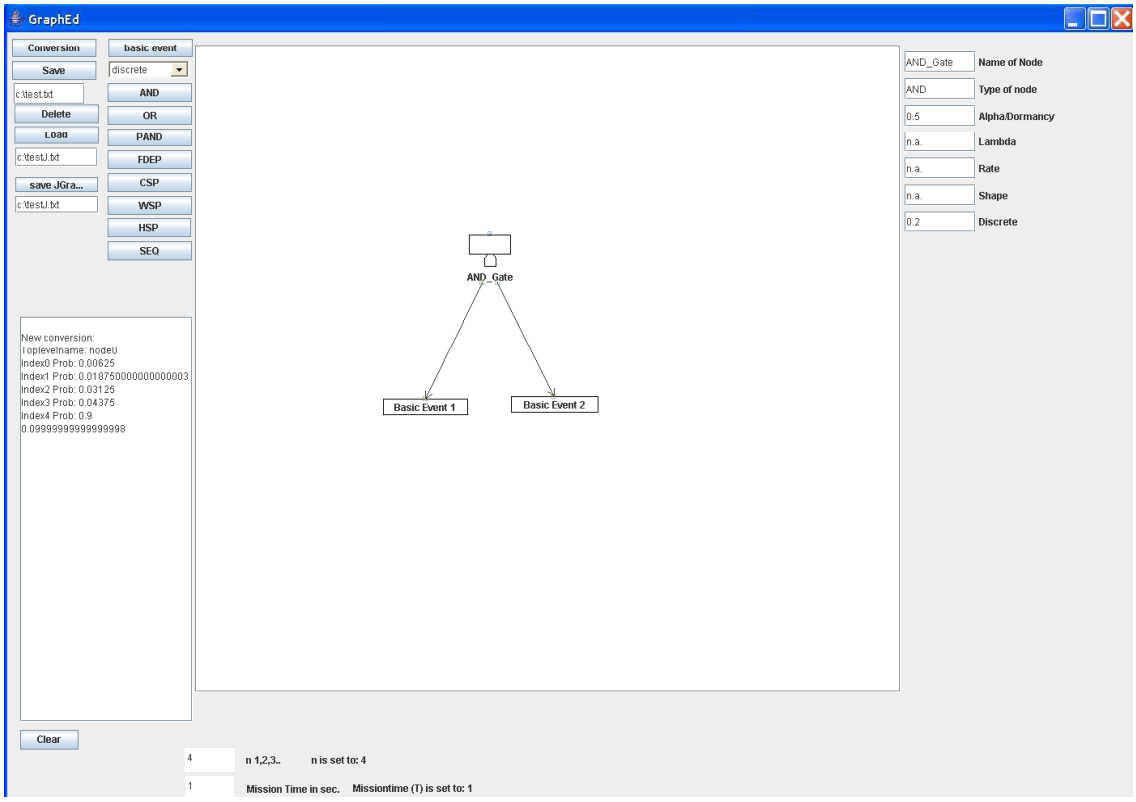


Figure 3.5: Screenshot

The remaining buttons deal with saving the JGraph as Galileo file (button “save”) and saving as derived ‘Jgraph’ format with coordinates (button “Save JGraph”) in order to load (“Load”) and ofcourse to start the conversion (button “Conversion”).
For conversion two fields are available where the n, and T (which is the mission time as discussed in the background section) can be entered. These can be found at the bottom of the screen.
The result is displayed in a scrollable window on the left.

The GUI is therefore able to take two inputs. Either the user draws his own tree from scratch, or he loads a previously drawn tree. The output is always the unreliability at every interval (n=0 through n+1) and the unreliablity of the system as a whole during missiontime. The three dif-

ferent parts of the GUI are discussed next. Interface, Display and DFT editing. The saving and loading of the DFT is discussed in section 3.9.4 followed by a section describing the package used in this part of IDIC; JGraph, section 3.9.5.

3.9.1 Interface

Interface is all the parts of the GUI where the user can communicate with the tool. This is therefore all the buttons and edit fields. There are two columns of buttons on the left side of the screen. The rightmost column of buttons are the buttons which call upon the functions to create a new gate (or basic event) in the central drawing area. Pressing the button will result in a new gate of the type that was written on the button to appear in the left top corner of the drawing area. With the mouse the gates can be moved to other locations.
The pulldown box directly below the 'basic event' button contains three types of basic events. The *discrete*, *exponential* and *weibull* type. Selecting a type and then clicking on the 'basic event' button will create a basic event in the left top corner of the type selected in the drop down box. The type of the event is not visible, instead the type of event can be determined upon selecting the basic event. At the right side of the screen the contents of the fields will change to represent the type of the basic event. When the type is *discrete*, the fields for *lambda*, *rate* and *shape* will change to n.a. (Not Applicable) and the field for *discrete* changes to the value previously entered or 0.0 as initial value.
To change the value, enter a new value and press enter. This will change the value in the selected event or gate. This is also the way to change the name of a gate or event.
Below the drawing area are the two 'global' fields. Here the n and *MissionTime* are entered. The n is a value of $\mathbb{N}$. The larger the n the more accurate the results will be. The *MissionTime* is entered in seconds and is also $\in$ of $\mathbb{N}$. Any deviation of this format including extra tabs will be rejected as valid formats. The leftmost column of the screen are the rest of the buttons and fields. The top button will start the conversion if both the n and *Missiontime* are entered conform specification. If the values are correct the conversion will start. The next button is called "save". This button will save the drawn tree in Galileo format. This format cannot be loaded in the tool, but this file can be loaded in Galileo. The editfield below the button lets the user choose where to store the file. Standard this is a file called 'test.txt' in the root of the 'c' drive.
The button delete lets the user delete any part of the drawing currently selected. Either a gate, arc or a combination of both. The load button can load any previously saved drawing as long as it is in specified format. This format is give in section 3.9.4. Again the editbox below the button determines the location and name of the file to be loaded. If the file cannot be found at these coordinates the file cannot be loaded. The standard name and location to be loaded is "c:\testJ.txt". The button below will save the drawing with coordinates to the same standard name and location.

3.9.2 Display

The display is all the fields which the user cannot edit or control. There are only two of those fields. The central drawing area and the scrollable box on the left side of the screen. This leftside box is for displaying the results of a conversion. The format of this is given in section 3.8. The central drawing area is where the gates, basic events and lines are drawn. How this is done can be read in section 3.9.3 The basic events are simple rectangles. The lines are drawn from top to bottom. The toplevel gate has no incoming edges and the leaves will have no outgoing edges.
The gates are standard. The pictures representing the gates are all given in the background section (section 2).

3.9.3 DFT editing

There are three subsections in this section. Nodes, edges and deleting/updating. In each subsection a short description of the steps taken for each task is given.

nodes

The nodes are created by creating a Cell. Because I want to store several properties I need to create my own format. This is called 'MyGraphCell'. This cell is then inserted to the graph. JGraph makes sure this created cell is inserted in the graph and displayed on screen. But what about this 'MyGraphCell'. This is a format based on JGraph's own *DefaultGraphCell*. It has the following fields:

```
1  public class MyGraphCell extends DefaultGraphCell {
2          private String name;
3          private String gateType;
4          private String alpha;
5          private String lambda;
6          private String rate;
7          private String shape;
8          private String discrete;
9          private String basicEventType;
10         private Boolean primary;
11         private int x=20;
12         private int y=20;
13         private int w=45;
14         private int h=20;
15         private DefaultPort port1;
16         private DefaultPort port2;
17         private DefaultPort port3;
```

The strings are all attributes of a gate. The gateType is the type of gate e. g. *AND*, *PAND* etc. or the type basicEvent. In which case the applicable attributes of a basic event are filled. discrete, lambda or rate and shape and for easy access the field basicEventType.
The integers x and y are the location of the gate or event. Standard in the leftmost corner (20,20). The dimensions (Width and Height) are 45 by 20, see the values in the list above.
Each gate will have three ports. These ports are the hooks to which a line can be attached. Because there is a distinction in only the children (for example, primary and spare for a spare gate), there is one port on top of the node and two at the bottom.
The way the gate looks e. g. the displayed picture is determined by the field of gateType.

edges

The edges are drawn from port to port. Each time the left mousebutton is clicked, the program checks for a port at this location. this is the start port. If the mouse is moved after this a line is drawn on the screen. This is done by the JGraph package. When the mousebutton is now released and there is an other port at this location a line is drawn between these two points. The parents of these ports are then notified of this. This means that the source gate will now have a child and the target will have a parent.

deleting/updating

The delete is straightforward. Just tell the graph that the current selected node(s) and or edges need to be deleted. However because the ports are added to the standard GraphCell these need to be deleted separately. Simply delete the ports and then proceed with the gates themselves.

The updating is mostly changing values in the stored nodes. Because everytime a node is selected it will (re)read the values stored and update the field on the screen. The names of the nodes is different because they need to be changed on screen and in the internal graphstructure as well. Fortunately the graph has his own update function. Simply change the name in the stored node and update the name in the graphstructure, this way JGraph will automaticly change the name as it appears on screen. After all this you will want to save your drawn tree for future use.The next section is about saving and loading and the format used for this.

3.9.4 DFT in Galileo format with coordinates

This section will explain the DFT format used for saving and loading of the drawn trees. This format is identical to the Galileo format with the addition of coordinates. The reason for using the Galileo format and adding coordinates is because the Galileo format is easily read and the savefunction for this format is already written. The drawn tree is saved to the Galileo format to initiate the conversion. To add coordinates is a small addition. The format of the runningexample is:

```
toplevel "AND_Gate";

"AND_Gate" and "Basic Event 1" "Basic Event 2" 353.0 202.0;

"Basic Event 1" 290.0 358.0
phase 1 prob=0.5 cov=1 res=0 repl=1 dorm=0.5;

"Basic Event 2" 419.0 359.0
phase 1 prob=0.2 cov=1 res=0 repl=1 dorm=0.5;
```

The format is the same as in section 3.2. However now after each gate the coordinates of the gate are written. First the name (AND_Gate), type (*AND*) and children (Basic Event 1 and Basic event 2) followed by x-coordinate (353.0) and y-coordinate (202.0). The gate definition is completed with the semicolon (;). The basic events have no type, so when a name is followed by a number this is always the x- and y- coordinates of this basic event.

Saving

The input for saving is the global graph which is a JGraph component. No individual nodes or edges. These must be extracted from this graph in order to be able to save the graph in Galileo format with additional coordinates. The output will be a file at the location determined by the field below the button “save JGraph”. The format of this file is Galileo with added coordinates. Because the JGraph package is used this format is also called JGraph format, hence the name on the button. To save the graph in a file, we need to have the nodes from the graph. Now we need to construct the file. The file consists of three parts. The toplevel, the gates with type and children and finally the basic events. In order to find the toplevel we use the following pseudo code algorithm.

```
Make a list of all nodes;
Make a copy of this list;
Create empty list of processed nodes;
for all nodes do
    Get neighbours;
    if neighbours==0 then
        Node is leaf;
    else
        If neighbour is not processed then process it and add to processed list.;
    // This basically ensures that every node which is a child will be written to the
    processed list.
Compare the list of all nodes with list of processed nodes. Overwrite the positions of the
nodes that are already processed. What remains is a list of unprocessed nodes and empty
strings. Print Toplevel with the unprocessed nodes.
```

The result of this conversion is that the nodes who have children (and are therefore gates) are already handled. These nodes should be saved in a string in the format 'Name type child1 child2 x-coordinate y-coordinate;' All this information is stored in the nodes themselves.
Because a check is made for leafs, these can be stored as well.

All that has been written to the file so far has been toplevel. The next part of the file are the gates. We had already formatted these lines, so simply write the resultstring to the file ensures the gates.

The leafs are a bit more work. Because the type of event determines the exact format to write. So after printing the name, coordinates and the nextline symbol a check is made for the type of basic event which is also part of the node. Depending on the type, the correct values are printed. The result is the file given in the beginning of section 3.9.4

Loading

Loading requires a location and name of a previously saved (or manually created) file. This file has to be in Galileo format with added coordinates. The result will be the original drawn graph in the central drawing area, just at it appears in Figure 3.5. The first thing this function does is removing all the nodes, edges and ports from the drawing e. g. making the screen empty. Because loading requires a empty graph to which the loaded nodes, edges and properties are added. Because the build of the file is known and there are only a few different variations ANTLR is not used here. Instead a large case statement is used. The file is read and based on the current token one of the cases is taken. If the token is the word toplevel, then we know the next token will be a string with the name of this toplevel. If the token is a *String* (characters enclosed by quotes) followed by a *Word* (several characters without enclosing quotes) then we know this is a gate with its type. So we now do the same as we would have pressed the corresponding button on the GUI interface. We create a new node with the type that is the read *word*. The next token is the first child. This child is created and added as child to the gate. The next token is the second child. Again the child is created and added as child to the gate. The next two tokens are x- and y-coordinates. Simply updating the node involves removing and recreating this node in order to place this node on the new coordinates on screen.
Once all the gates are processed, the basic events come next. Again the node is created on the coordinates read directly after the name of the event. The properties are all added as the appropriate token is read. After reading the last token all the nodes are now created. However the observant reader should realize by now that there are no arcs drawn. This is because the source and target cells must be created before an arc can be added. However because the basic event are recreated and replaced upon reading of the coordinates for the basic events the arcs cannot be added when reading the gates. The connections are therefore stored and read at the end.
The arcs are drawn from port to port. The ports with their internal location relative of their parentport are read from source and target node and the connect function of JGraph is called.

Now all the nodes are drawn and the connecting arcs are present as well. The properties are

also stored and the network is now ready for use. Next section will give some information about the JGraph package used.

3.9.5 JGRAPH

The JGraph package is a graphical package and enables the user to create nodes and draw edges between them. The package also allows for moving nodes and edges, delete them and change appearance.

JGraph is a mature, feature-rich open source graph visualization library written in Java. JGraph is written to be a fully Swing compatible component, both visually and in its design architecture.[2] The GUI is build using this package. There is a global graph defined to which nodes and edges can be added, changed or deleted. The package takes care for updating the visual container which has been initialized with this graph. With help of JGraph, mouse events are captured and lines can be drawn from node to node. The saving and loading is also done with help from JGraph. When saving, the program requests all the nodes from the graph. It will save the nodes and all its children in the memory. It identifies the basicevents and save them too. Until all nodes are processed until just the topnode is left. This topnode can now be saved in a file. This is because the Galileo format requires the topnode first, followed by the gates with their children and concluded by the basicevents with their properties.

3.10 Testing

There are a few points where the implementation needs to be tested. These main focus points are:

- User interface
- conversion

For each of these points a description of the subcomponents and the checked functionality is given.

3.10.1 User interface

The user interface consists of:

A central drawing area
The central Drawing area is the most important aspect of the program. Without this you can't draw (or load) a Fault Tree. Without a Fault Tree, there is no conversion. However, without additional functionality there isn't much this area can do. Without some way to add nodes to this area nothing can be drawn. As soon as there are nodes, edges can be drawn. Simply by clicking on specific points of a node (called ports) the user starts to draw an edge. Releasing the mousebutton on a (target) port will connect both ports with an edge. Testing this is straightforward. Either by displaying some text, or checking the connection in the total graph or the saved textfile is sufficient.

Several buttons
There are a number of buttons in the program. Checking these can, in some cases, be straight forward. The 'node' buttons, e. g. the *AND*, *OR*, *basic event*, *PAND*, etc. are simply checked by observing the appearance of the correct node in the drawing area.
The Save/load buttons do more, and therefore require more work. Not only do they need to display the entire graph (respectively in a text file or on screen) but they need to show them correctly.

Not all possible configurations of gates and events can be checked. Only the most basic networks e. g. one gate with two children and the case studies have been thoroughly tested.

some node properties
The properties of the current selected node needs to be received from the current selected node. Simply clicking on different nodes and observing the change in the properties is sufficient. These values are taken from the file on loading, but the values can also be changed by the user. Allowing the user to change the properties and saving these changed values (if correct values e. g. Type must be one of the allowed ones, values must be in the correct format) is checked by observing the change after editing.

result area
The result area must be scrollable (if multiple results).

global properties
Inserting global properties like the filename to save to, the n and missiontime are checked by reading and printing these values.

3.10.2 Conversion

The analysis of the conversion is the most difficult part. Not only the basic building blocks need to be correct, but also the program in itself. What better way to compare the results than actually comparing the probability result from this program to Galileo. This has been done for all the basic networks, these are just a single gate with two children.
Two case studies are also fully checked.

The next chapter shows these two case studies.

Chapter 4

Case Studies

There are 2 case studies which have been used for this thesis. Both shall be introduced and the results are given and discussed.

The first case study is the Hypothetical Cascaded PAND System (HCPS). This system is shown as DFT in Figure 4.1

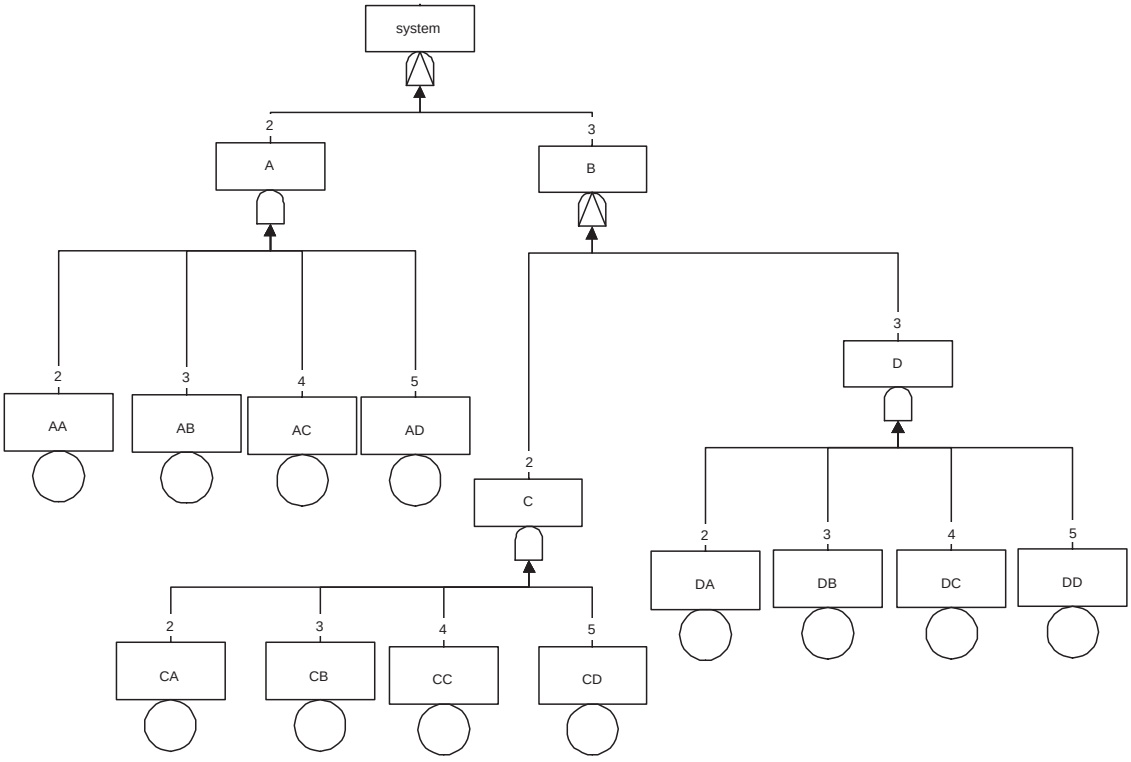


Figure 4.1: Hypothetical Cascaded PAND System

Because the topnode is a Dynamic gate, the whole system has to be converted to MC in Galileo. This means that the (fast) BDD technique can't be used and the time needed for the analysis is increased.
The probability of the top node ("system") is displayed in table 4.1 for several values for 'n'.

n	probability
4	0.00242830
10	0.00176829
15	0.00162748
30	0.001490108
80	0.001406222
85	0.00140329
87	0.00140221

Table 4.1: Probability for HCPS system

From this table it can be seen that the larger you choose the 'n', the more accurate the answer is becoming. Note that solving this, including the conversion to BN, takes place in 10 seconds for the largest 'n'
The Galileo file saved can also be solved by Galileo itself. This results in:

```
Unreliability ===> 0.00135668
Elapsed time: 0 hours, 8 minutes, 4 seconds.
```

A lower and more accurate result, but very slow. The BN that has been saved in GeNIe format provides the same answers as in table 4.1. The IDIC result approaches the exact number from Galileo, however it never reaches it. The 'n' of 87 is the highest result. Any higher will cause a memory overload and no result is reached. We assume that for higher 'n' the result will be even closer to the exact result of Galileo.
Now to group these results in table 4.2.
Coral is a tool which solves FT via MC. The results from Coral are taken from [10]. We see that

Tool used	Result	n used	Time needed
Galileo	0.00135668	none	8m 4s
IDIC	0.00140221	87	0m 10s
Coral	0.00135668	none	1m 7s

Table 4.2: Results from various tools for HCPS system

both MC-based tools have the same answer and that IDIC is very close. This is due to the fact that IDIC works with intervals. The higher the n, the more accurate the results will be. The n of 87 as seen in the table is the highest reachable with the current implementation and machine used.

The second case study is the Multi-processor Distributed Computing System (MDCS). This system is shown as DFT in Figure 4.2

This system includes some *WSP* gates with spares and several exponential basic events.
The results are given in table 4.3 with the same values for 'n'.

n	probability
4	2.0002545175756836E-9
10	2.0002547396202885E-9
15	2.000254628597986E-9
30	2.0002547396202885E-9
80	2.000254850642591E-9
85	2.000254850642591E-9
87	2.000254850642591E-9

Table 4.3: Probability for MDCS system

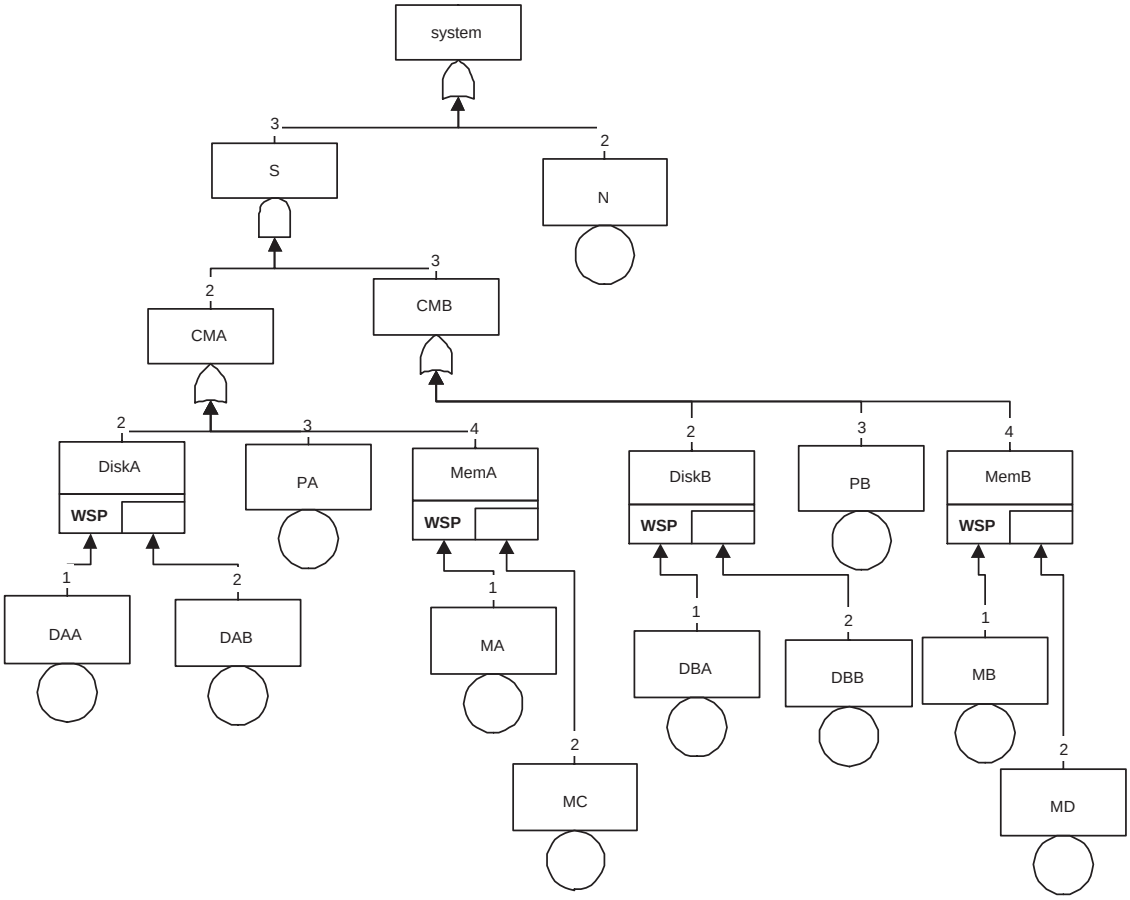


Figure 4.2: Multi-processor Distributed Computing System

Again this system is also checked by Galileo. This is where Galileo is slightly faster. Where Galileo has the result directly, IDIC needs 5 seconds for the upper values of 'n'. The result however appears less accurate than IDIC because Galileo cuts the result of after the fifth decimal.

```
Unreliability ===> 2.00025e-009
Elapsed time: 0 hours, 0 minutes, 1 seconds.
```

Now to group the results in table 4.4.
In this example IDIC is slightly slower. This is because the higher parts of the tree can and have

Tool used	Result	n used	Time needed
Galileo	2.00025e-009	none	0m 1s
IDIC	2.000254850642591E-9	87	0m 5s
Coral	2.00025e-009	none	not known

Table 4.4: Results from various tools for MDCS system

been solved by using BDD techniques in Galileo. In this case, the n of 87 is accurate enough to get the same result as Galileo and Coral.

Chapter 5

Conclusion

We discussed some background to the definitions used in this thesis. We discussed Fault Trees and Bayesian Networks. Some insight was given in Markov Chains and we saw the benefits of Bayesian Networks in probability analysis. We introduced a tool where the user could draw his fault trees and with the press of a button get the unreliability of the FT back. The tool, called IDIC, used Bayesian Networks for the analysis and is independent of other programmes (Galileo is dependent on MS Word and MS Visio).
We also saw some case studies in which the results of IDIC were compared to other programmes. In this chapter a summary of the work is given. This is followed by the results from the tool and we draw the conclusion if it 'was worth it'. The work and benefits are put into their perspective. This chapter is followed by a section on future work. What can be done to enhance the tool and to ensure future use.

5.1 Summary

Fault Trees are widely used for reliability analysis. Static FTs who have no temporal and/or functional failure dependencies are solved by using BDDs. Dynamic FT are usually solved by MCs. Unfortunately a MC will have a potential effect called state space explosion. This means that the number of states necessary to solve the MC (e. g. calculate the unreliability of the composed system) can become extremely large. Bayesian Networks is a different method to calculate unreliability. This method requires less states than a MC and is an attractive choice for this. But since most systems are drawn as FT opposed to BN it would be nice if the user can still use FTs but calculate the unreliability via BNs. IDIC provides this functionality. The user can draw his FT in the GUI of the tool and by pressing a button he lets the program calculate the unreliability of his system by translating the FT to a BN and analysing it.
The tool consists of a GUI in which several FT constructs can be drawn. These are:

- static gate
  - AND
  - OR
- dynamic gate
  - PAND
  - FDEP
  - CSP
  - HSP
  - WSP

- basic events
  - Discrete distributed BE
  - Exponential distributed BE
  - Weibull distributed BE

For the basic events the variables “discrete” (for discrete BEs), “lambda” (for exponential BEs), “rate” and “shape” (for Weibull BEs) and “alpha” (for all types of BE and gates) can be set. The variables of “n” (The missiontime is divided into n sections) and missiontime (time considered the lifetime of the system or the time considered as the mission duration) can also be set prior to the conversion.
IDIC will read the drawn FT, convert it to a BN and analyses it. The results, which is the unreliability per state, is displayed in the GUI.
IDIC has been tested via two case studies. The results of these are discussed in the next section.

5.2 Results

IDIC is intended to work with FTs and use BNs to analyse the system, modelled in the FT structure. To say this method is indeed an improvement a comparison is made with Galileo and Coral. Galileo uses MCs to analyse its FTs and IDIC used BNs to do the same. In table 5.1 the results, for the first case study (HCPS), are given. The tools used and their results are given in respectively columns 1 and 2. In column 3 the n is given. The higher the n the more accurate the results will be. The time needed to calculate the results is given in column 4.
We see from the table that IDIC is not as accurate as the other two tools. This is due to the

Tool used	Result	n used	Time needed
Galileo	0.00135668	none	8m 4s
IDIC	0.00140221	87	0m 10s
Coral	0.00135668	none	1m 7s

Table 5.1: Results from various tools for HCPS system

fact that BNs become more accurate when the n is increased. The n of 87 in the table is the maximum reachable. A higher n resulted in memory overflows, however the results is fairly close to the results obtained by the other tools. The time however is faster. 67 seconds for Coral and 484 seconds for Galileo to 10 seconds for IDIC. This could be profitable for larger systems.
The second case study (MDCS) and its results are given in table 5.2

Tool used	Result	n used	Time needed
Galileo	2.00025e-009	none	0m 0s
IDIC	2.000254850642591E-9	87	0m 5s
Coral	2.00025e-009	none	not known

Table 5.2: Results from various tools for MDCS system

This case the results are identical, the time needed however is not. This however can be improved upon by selecting better algorithms and/or data structures and improving memory usage.

Construction of IDIC was meant to be modular and adaptable. This has been done. If the used Galileo format was found to be inadequate, it can be replaced. The same goes for the package used to implement the GUI, or analyse the BN.
From both case studies it is shown that this method, using BN to solve unreliability questions in FT models, is in these cases faster and results in nearly the same results as MC-based programs.
Because this is still a prototype numerous improvements and/or adaptations can be done. In the next section a number of these improvements are discussed.

5.3 Future Work

IDIC is not finished. The GUI is functional, but not faultproof and there are several improvements still to be implemented.

A few of these improvements are listed below.

- Errorhandling. Most of the errors that can exist are not yet distributed back to the GUI. The user will not know what is wrong.
- Erroneous user input. Although most edit fields will not accept anything other that numbers, some fields will present a error when a TAB token is detected or when letters are used instead of numbers. However in the case of exponential numbers with a E in the number this is desired.
- User friendlyness. The TAB key can be used to 'tab' to other field. The editfields should update when the focus leaves the field. Currently this is only done upon the pressing of the Enterkey.
- Adaptation. Provide a browse screen where the user can browse to the field where he/she wants to save the files. Use a profile file for this. Also make the screensize adaptable for larger (or smaller) screens.
- extend. Provide evidence, use different algorithms for solving, give results for every node, not just the topnode.
- intelligently. Let the tool use the E for very large (or very small) numbers, make use of either plus or minus tokens, make Missiontime not just whole numbers but also reals.
- Surveyable. Make a menu structure, make use of contextmenus upon the press of a (mouse)button.
- Results. Results can be altered to suite the need of the user, they can be displayed graphically in a chart, or table.
- add functionality. Several constructs of spare gates and (shared) spares is not detected yet. Such as multiple spares shared by mulitple gates.
- upgrade. A FT cannot have some type of constructs, such as having a gate as spare. BNs can deal with these.

Chapter 6

Abbreviations

3P2M	3 Processors 2 Memory units
BDD	Binary Decision Diagram
BE	Basic Event
BN	Bayesian Network
CPD	Conditional Probability Diagram
CSP	Cold Spare
CPT	Conditional Probability Table
DAG	Directed Acyclic Graph
DBN	Dynamic Bayesian Network
DFT	Dynamic Fault Tree
DTBN	Discrete Time Bayesian Network
FDEP	Functional DEPendency
EoF	End of File
EoL	End of Line
FT	Fault Tree
GUI	Graphical User Interface
HECS	Hypothetical Example Computer System
HCPS	Hypothetical Cascaded PAND System
HSP	Hot Spare
IDIC	Infinite Diversity in Infinite Combinations
MC	Markov Chain
MIU	Memory Interface Unit
n.a.	Not Applicable
PAND	Priority AND
PMF	Probability Mass Function
SEQ	SEQuence enforcing
SFT	Static Fault Tree
WSP	Warm SPare

Appendix A

Manual

This Appendix is a manual on how to use this tool. It is divided into several paragraphs and contains all the different tasks that can be executed.

Creating nodes: To create a gate node, click on the button displaying the name of the gate you want to create. To create a basic event, select the appropriate distribution from the pull-down box below the 'basic event' button. For exponential distribution, select *exponential*. For discrete distribution select *discrete* and for Weibull distribution, select *Weibull*.

Creating Edges: Each gate has 3 points where a line can be attached. 1 centre top, and the other two at 1/3 and 2/3 of the bottom line. These points are called ports. To draw a line, hold your mouse above one of the ports, click and hold the mousebutton. If the mouse is moved a line will appear. When the mousebutton is released above an other port, it will draw the line.

Deleting nodes or edges: To delete a node or edge, select the desired edges and nodes. This can be done by either by selecting them with the mouse (hold shift to select multiple items) or to click with the mouse on a empty place on the canvas and holding the button. Moving the mouse will result in a selection box.
After selecting the nodes and or edges to delete, press the button 'delete'.

Modifying attributes: To modify an attribute of a node, click on the node. On the right side of the screen the properties of the node will appear. To modify them, click in the field, change the value and press **enter**. Pressing enter is the signal for the program to store the value in the graph. The fields n and *MissionTime* can be altered at any moment. Pressing enter is not required, because these values are checked and processed when the user presses the button 'conversion'.

Changing location of saving and loading: On the left of the screen are three boxes containing a filename. Standard this is "c:\test(J).txt". To change the location, edit the field to contain the desired location and name for the new file. The file will be automatically overwritten. To load a file, the desired location and name of the file to be loaded must be entered. None existing files will not be loaded. The topfilename is the location where the program saves the Galileo formatted file used in the conversion. The other two are the locations for saving and loading the specific format for IDIC.

Bibliography

[1] M Chechik A. Gurfinkel. Extending extended vacuity. Technical report, Department of Computer Science, University of Toronto, 2004.

[2] Gaudenz Alder. Design and implementation of the jgraph swing component. Technical report, KHLim, February 2002.

[3] L. Bosia and P. von Rohr, editors. *A Class for discrete Bayesian Networks in Darwin*, volume ETH Zurich, 2005.

[4] H. Boudali. *A Temporal Bayesian Network Reliability Modelling and Analysis Framework*. PhD thesis, University of Virginia, 2005.

[5] Eugene Charniak. Bayesian networks without tears. *The American Association for Artificial Intelligence*, 1:15, Winter 1991.

[6] Kevin J Sullivan David Coppit, editor. *Galileo: A tool built from mass-market applications*, Limerick Ireland, June 2000. 22nd Iternational Conference on Software Engineering. http://www.cs.virginia.edu/ ftree/2003-redesign/pages/Research/abstracts.html.

[7] Robert McGovern David Gallardo, Ed Burnette. *Eclipse in Action - A guide for Java developers*. Manning Publications Co., 2003.

[8] Marek J. Druzdzel, editor. *GeNIe: A development environment for graphical decision-analytic models*, volume Annual Symposium of the American Medical Informatics Association (AMIA-1999),, Washington D.C., November 6-10 1999.

[9] J.B. Dugan. *Reliability Analysis of Computer-Based Systems using Dynamic Fault Trees (Tutorial)*. Department of Electrical, Computer & Systems Engineering from the university of Virginia, June 2001.

[10] P. Crouzen H. Boudali and M. Stoelinga. A compositional semantics for dynamic fault trees in terms of interactive markov chains. In *LNCS 4762*, 2007.

[11] P.E. Mandar Khanal. probability concepts. April, 2007 http://coen.boisestate.edu/mkhanal/probabil.htm.

[12] A. Villafiorita M. Bozzano. Esacs: References on dynamic fault trees. April, 2007 http://www.cert.fr/esacs/doc/dynamicFT.html.

[13] Terence Parr. Antrl website.

[14] J.D. Andrews R.M. Sinnamon, editor. *Fault Tree analysis and Binary Decision*, volume RMAS symposium, Las Vegas USA, Jan. 1996.

[15] T. Seppnen. Exercise 2 a2 and q2.pdf. website from Finish University of Oulu, Department of Electrical and Information engineering http://www.ee.oulu.fi/research/tklab/courses/521497S/, March 2006.

[16] Sun Microsystems. *Java API*.

[17] K.S. Vastola. Essentials of probability. Online guide into probability from the Departement of Electrical, Computer and Systems Engineering Rensselaer Polytechnic Institute Troy NY http://networks.ecse.rpi.edu/~vastola/pslinks/perf/node12.html, 1996.

[18] Relex website articles. Fault tree construction: Events and gates part iii: Dynamic gates. http://www.relex.com/resources/art/art_faulttree3.asp, September 2006.