Anja Karl B.Sc.

Safety Methods applied to Functional and Technical Architectures

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology
and
Research Center Virtual Vehicle

Supervisor

Wotawa, Franz, Univ.-Prof. Dipl.-Ing. Dr.techn.
Institute of Softwaretechnology (IST)
Schwarzl, Christian, Dipl.-Ing. Dr.techn. Bakk.techn.
Virtual Vehicle

Graz, TODO: Date

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Date/Datum	Signature/Unterschrift

Abstract

Society relies more and more on technical systems. With the trust placed in these comes a high need for safe systems. For safety critical systems it is necessary that they are particularly reliable. This thesis describes an algorithm calculating the reliability of systems consisting of connected components. The functionality of most of these components depends on the output of other components, which introduces fault propagation. To avoid such problems, the system may also contain components for redundancy. Evaluation of the reliability of the system can be used for identifying weak points in the system and allows for improving the system by adding further redundancy components. The system's information flow may also contain loops and cycles. Therefore it is not possible to use the common Fault Tree Analysis (FTA) method. Instead, this thesis uses a Bayesian Network approach. It describes the full process starting at a UML model of the system, creates an intermediate Dependency Graph and proposes an efficient way of applying cycle unrolling to such systems to receive a valid Bayesian Network. The approach preserves the successor and descendant relationships in the graph in such a way, that the failure probabilities are taken into account only once. Therefore it is necessary to use conditional probability for evaluating the reliability of each component, for which we introduce an algorithm specific to Connected Components Systems containing redundancy components. The thesis will be supported by examples for the complete implementation of the whole process and concluded with an evaluation of real systems using this implementation.

Contents

Contents iii

List of Figures vi

List of Listings vii

List of Examples ix

Credits xi

1 Introduction 1
- 1.1 What is Reliability and why is it important? . . . 1
- 1.2 How can we measure Reliability? . . . 1
- 1.3 How can we evaluate the reliability of a system? . . . 1
- 1.4 Difference between Faults and Failures . . . 2
- 1.5 Distinction between qualitative and quantitative evaluation of a model . . . 2
- 1.6 What kind of systems are we handling? . . . 2
- 1.7 The problem this thesis addresses . . . 3
- 1.8 How this thesis provides a solution . . . 3
- 1.9 Capabilities and Limitations of the solution . . . 4
- 1.10 Potential applications of the results . . . 4
- 1.11 The Complete Process Described in this Thesis . . . 4
- 1.12 Structure of this thesis . . . 5

2 Related Work 7
- 2.1 Failure Mode and Effects Analysis . . . 7
- 2.2 Type-Hierarchy of Reliability Models . . . 7
- 2.3 Power-Hierarchy of Reliability Methods . . . 8
- 2.4 Reliability Block Diagram . . . 8
- 2.5 Fault Tree Analysis . . . 9
  - 2.5.1 FTA: Basics . . . 9
  - 2.5.2 Method for handling systems with acyclic loops . . . 10
  - 2.5.3 Handling Systems containing cycles . . . 10
  - 2.5.4 Evaluation of possible causes for a failure using Fault Tree Analysis . . . 10
  - 2.5.5 Dynamic Fault Tree . . . 11
  - 2.5.6 Priority Dynamic Fault Trees with Repeated Events . . . 11
- 2.6 The Petri Net Approach . . . 12
- 2.7 Reliability Analysis using Bayesian Networks . . . 13
- 2.8 Loop Unrolling in compiler construction . . . 13
- 2.9 Loop Unrolling in Model Checking . . . 13
- 2.10 Adaption Loop Unrolling to Cycle Unrolling . . . 13

3 The Input to our Analysis: The System and its Components **15**
3.1 Definition of a Connected Components System . . . 15
3.2 Component's Composition . . . 16
3.3 Component Types . . . 17

4 The Dependency Graph: Structure and Creation **21**
4.1 Introduction . . . 21
4.2 Definition of the Dependency Graph . . . 21
4.3 Nodes in the Dependency Graph . . . 22
4.3.1 The Forward Nodes . . . 22
4.3.2 Preprocessing Nodes . . . 22
4.3.3 Failure Nodes . . . 22
4.3.4 Output Nodes . . . 22
4.4 Relations of Nodes in the Dependency Graph . . . 23
4.5 State of Nodes . . . 23
4.6 Transforming a Connected Components System's Component into a Graph . . . 23
4.7 Creating the Graph from a full Connected Components System . . . 24

5 Transformation into a Bayesian Network: Applying Cycle Unrolling **27**
5.1 Introduction . . . 27
5.2 Definition of a Bayesian Network . . . 27
5.3 Definition of Graph Structures . . . 27
5.4 Tarjan's strongly connected component algorithm . . . 29
5.5 Unrolling Cycles . . . 32
5.5.1 Proof for limiting the number of unrolling iterations . . . 32
5.6 Determine the best way to cut as many cycles as possible . . . 33

6 Calculating the Reliability of Nodes in the Dependency Graph **41**
6.1 Introduction . . . 41
6.1.1 Preliminaries . . . 41
6.1.2 Definition of Failure Events . . . 43
6.1.3 Calculating Probabilities of Independent Failure Events . . . 44
6.1.4 Calculating Failure Probabilities with Random Variables . . . 46
6.1.5 Introducing Time . . . 46
6.1.6 Failure Rate Distributions . . . 47
6.1.6.1 Constant Failure Rate and Exponential Distribution . . . 47
6.1.6.2 Weibull Distribution . . . 49
6.1.6.3 Further Distributions . . . 49
6.1.7 Dealing with Conditional Dependence between Random Variables . . . 51
6.1.8 Dominators and Shared Predecessors . . . 51
6.1.8.1 Rules of d-Separation . . . 52
6.1.8.2 Calculation of Failure Probability with Dependent Random Variables . . . 54
6.1.9 Calculating the Reliabilities in Complex Dependency Graphs . . . 55
6.1.9.1 Reliability at a Failure Node . . . 55
6.1.9.2 Reliabilities at a Forward Node . . . 55
6.1.9.3 Reliability at a Node with two or more Parents . . . 56
6.1.9.4 Reliability at an Output Node . . . 56
6.1.9.5 Reliability at AN Preprocessing Nodes . . . 57
6.1.9.6 Reliability at VOTER Preprocessing Nodes . . . 57
6.1.10 Algorithm for Reliability Calculation . . . 58

7 Conclusions **63**

List of Figures

2.1 Reliability Block Diagram 8

2.2 This Figure shows a simplification of the system input for the Fault Tree in Figure 2.3. There exist multiple acyclic loops inside the system. There are for example three paths from node "2to the next And Gate, passing through node "3", "4", or "5". 9

2.3 The resulting Fault Tree for the system in Figure 2.2. The different parts between the loops are combined via an Or Gate, as are the two And Gates combining the loop paths. Source: https://en.wikipedia.org/wiki/Fault_tree_analysis License: Public Domain 10

2.4 A system containing a feedback - cycle between subsystem A and subsystem B. 11

2.5 Petri Net 12

2.6 Fault Tree corresponding to Petri Net in 2.5 12

3.1 Nested Connected Components Systems 18

3.2 Composition of a generic Component 19

3.3 Connected Components System with Different Types of Components 20

4.1 The Dependency Graph of a Single OR Component 24

4.2 The Dependency Graph of a Single VOTER Component 24

4.3 Exmaple of a Basic Dependency Graph 26

5.1 An Example of a Bayesian Network. Both Factors `Driver Drunken` and `Malfunctioning Car` may cause a `Car Accident`. 28

5.2 Strongly Connect Component Example: Input 30

5.3 Strongly Connect Component Example: First Strongly Connected Component 30

5.4 Strongly Connect Component Example: Next Strongly Connected Component 30

5.5 Replacing a Cycle by a Strongly Connected Component 31

5.6 Abstraction of Strongly Connected Components used for cycle unrolling proof 33

5.7 Proof that two unrolling iterations are enough 34

5.8 Strongly Connected Component after applying Cycle Unrolling 35

5.9 Dependency Graph with Unrolled Cycle 36

5.10 Strongly Connected Component containing more than one cycle 37

5.11 If we cut the wrong edge of a cycle, we might duplicate further cycles contained in the same Strongly Connected Component. Here the cycle with dashed edges is copied because we chose the wrong edge (between $cyc2.out$ and $cyc1.in2$). 38

5.12 The edges in towards the node *cyc*1.*out* are the better choice, because all cycles can be solved in one step. We can determine this node by looking at the number of edges inside the Strongly Connected Components ending in this node. The node with the most of these edges is the best choice. 39

6.1 Sample Space Ω with Events 45
6.2 Reliability of Exponential Distribution 48
6.3 Reliability of Weibull Distribution 49
6.4 Failure Rate of Weibull Distribution 50
6.5 Dependency Graph containing Loops 53
6.6 d-Separation Rule 1 54
6.7 d-Separation Rule 2 54
6.8 d-Separation Rule 3 54

List of Listings

5.1 Tarjan's strongly connected component algorithm 29

List of Examples

3.1.1 Connected Component System with two Components 16
3.1.2 Two nested Connected Components Systems 17
3.3.1 Connected Components System with Different Types of Components 20

4.7.1 Generation a Dependency Graph from a Connected Components System 25

5.2.1 Exmaple of a Bayesian Network 28
5.4.1 Strongly Connected Component Algorithm 30
5.4.2 Cycles and Strongly Connected Components 31
5.5.1 Unrolling applied to a small Dependency Graph 33
5.6.1 Strongly Connected Component containing more than one cycle 34

6.1.1 Sample Space Ω of the System in Example 4.7.1 44
6.1.2 Failure Event in the System in Example 4.7.1 44
6.1.3 Union and Intersection of Failure Events 44
6.1.4 Dependency Graph containing a Loop 52
6.1.5 d-Separation rules applied to Example 6.1.4 55
6.1.6 Reliabilities in Example 6.1.4 57
6.1.7 Repetition of Closest Branching Predecessors in the Dependency Graph of Figure 6.5 . . 58
6.1.8 Relative Reliability in the Dependency Graph of Figure 6.5 - I 59
6.1.9 Relative Reliability in the Dependency Graph of Figure 6.5 - II 60
6.1.10 Relative Reliability in the Dependency Graph of Figure 6.5 - III 61
6.1.11 Absolute Reliability in the Dependency Graph of Figure 6.5 - I 62

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews, 2012].

Chapter 1

Introduction

1.1 What is Reliability and why is it important?

In these days, we depend a lot on machines. For some of them it is extremely important to operate reliably. This can be because they could put us into danger, if not working properly, like for example car [Grabert and Luy, 2012] or a medical system. On the other hand there are systems, which are not safety critical, but still need to be highly functional since they are very difficult to maintain, like offshore wind turbines[Arabian-Hoseynabadi, Oraee and Tavner, 2010]. Safety methods are also very important for nuclear [Commission, 1998] and aeronautic [Vesely et al., 2002] technologies. Reliability defines how likely a system is to be able to perform its functions.

1.2 How can we measure Reliability?

Høyland and Rausand [2004] define measurements of Reliability Engineering including Reliability, Availability, Mean Time To Failure and Maintainability. While Reliability is the ability to provide a required function over a time period t considering environmental and operational conditions, Availability also takes possible maintenance of the system into account. The Mean Time To Failure defines the mean time until a failure is happening in the system. Maintainability defines, how much effort it is to maintain the system, meaning to either retain or restore its functionality. In this thesis Maintainability is neglected, in which case Reliability is equivalent to Mean Time To Failure.

Reliability can be therefore defined as the probability that a system's function is provided after a time period t, $R(t) = \Pr(\text{item is functioning after time period } t)$

In a car, the reliability of for example the airbag would be the probability, that the airbag is working properly at time t, given all environmental conditions like abrasion of the cables and deviation of sensor accuracy. A malfunctioning airbag could be both the absence of a airbag activation as also an inadvertent activation of it. Availability would also considers maintenance, like for example recalibration of the sensors. The Mean Time To Failure would describe the average time passing until the airbag starts to malfunction. This time span gives an indication on how often the system should be maintained or replaced.

1.3 How can we evaluate the reliability of a system?

The process of evaluating the reliability of a system is a field of Reliability Engineering. For safety critical systems it is important to know, how likely it is, that some of its components are not available any more after a certain time. Once such weak points in the system are identified, their function's reliability can be improved for example by introducing additional redundancy components. Redundancy

components are additional components providing the same functionality as the existing components allowing to maintain the intended behaviour if the existing component fails. The output of the original and the redundant component is evaluated by another component, which decides which observed output is correct. Another application of reliability engineering is the evaluation of the impact of a failure. This means to calculate all influenced components of a failure. Similarly it is also possible to calculate the most likely causes of a failure. This thesis describes algorithms for the reliability calculation.

1.4 Difference between Faults and Failures

IEEE Standard Glossary of Software Engineering Terminology [1990] defines a *fault* as a defect in a hardware device or component and a *failure* as the inability to perform its required functions. The *error* describes the deviation between the desired value and the output of a *faulty* component. Contrarily, Høyland and Rausand [2004] defines so called *failure events*. This event occurs as soon as a so far properly working component enters a *fault state*, meaning a defect occurs and the the component stops to provide its required functions in a correct way.

1.5 Distinction between qualitative and quantitative evaluation of a model

According to Blischke and Murthy [2000] there exist two types of reliability analysis: The qualitative and the quantitative analysis. The first is used to find possible failure modes and or causes, while the later describes methods of measuring metrics like the Reliability, Availability and Mean Time To Failure. A typical example for a qualitative method is the Failure Mode and Effects Analysis, while a Fault Tree Analysis can be used for both applications. This thesis concentrates on a quantitative evaluation, while the described structures also provide potential for a qualitative analysis.

1.6 What kind of systems are we handling?

The systems this thesis covers are so called Connected Components Systems. Those are systems which consist of multiple hardware components, which are sending information over directed connections. Each component performs calculation on their received input and forwards the output to the next components. These calculations might fail, because the component might be in a fault state. A cause for a failure can be for example a hardware failure - which is very likely to happen after a certain time span. If a component receives incorrect information - because of a fault in one of its ancestors - its calculations will always fail.

A Connected Components System can be hierarchically structured, meaning it may have multiple nested Connected Components Systems. These nested Connected Component Systems are abstracted into one so called Super Component, in order to split the model into its main parts and provide a method for structuring the system.

Like in the paper by A. Bobbio et al. [2001], the systems contain gate components. The covered gate types are the And Gate, the Or Gate and the Voter Gate. These Gates are taken from the Fault Tree Analysis concept.

- An AND Gate's input is considered correct, if at least one input is valid.
- An OR Gate's input is considered correct, if all input connections send valid information
- A n-VOTER Gate's input is considered correct, if at least n input connections send valid information.

Every gates' calculations might fail, independent of their type. Additionally to the type a component provides information about its failure probabilities at time t.

An example for a safety critical Connected Component System would be the airbag system in a car. Airbags provide a protection against injuries in case of an accident but could cause severe threat if triggered while driving. According to Aljazzar et al. [2009] an airbag system consists of multiple crash sensors, which forward their measured data to two redundant microcontrollers. Both microcontrollers evaluate the data independently and decide, whether the airbags should be triggered or not. Only if both agree on triggering, the airbag is activated. The sensors would therefore be OR gate components, the microcontrollers can be modelled by OR gate components or voter components, depending on how they analyze their input data and the output of the microcontrollers is then evaluated by an AND gate component.

1.7 The problem this thesis addresses

Since reliability engineering is a very important topic, a lot of approaches already exist. The most common of them are the Fault Tree Analysis and the Failure Mode and Effects Analysis.

This thesis is providing an approach for a quantitative reliability evaluation of a system. Therefore Failure Mode and Effects Analysis - as a qualitative method - cannot be applied [Chiozza and Ponzetti, 2009]. The Fault Tree Analysis [Kapur and Pecht, 2014] provides methods for both quantitative and qualitative evaluation and is therefore more suitable. The reason why it still cannot be applied is, that it is operating on a tree like structure. Although it is capable of handling common cause failures, it comes with the cost of diverging from a model of the information flow between components and the need to restructure the tree. As a consequence, the model is hard to interpret. This drawback is the reason, why researchers came up with more expressive solutions including the Bayesian Network approach [Khakzad, Khan and Amyotte, 2011].

This thesis is aiming to provide a solution for even more arbitrary system structures. In these systems it is not only possible to model common cause failures, but also feed back cycles in the information flow. These cycles can be neither handled by a Fault Tree, nor by a Bayesian Network, which is defined as an directed acyclic graph [Koski and Noble, 2009a].

1.8 How this thesis provides a solution

This thesis is going to handle the cycles by applying Cycle Unrolling, an algorithm originating from compiler construction. In compiler construction it is used to evaluate program code loops. The main idea is to process or evaluate a fixed number of cycle loops, before continuing with the further program. If a cycle is unrolled infinitely often, it will converge to the real solution. In the case handled case it is possible to fix the number of iterations to two to reach the exact values for the reliability.

In a graph based application, for every processed cycle, the nodes are duplicated and the recursive edges directed to the duplicated nodes. In the end, the recursive edges are dropped to cut the cycle. The main problem is not only to determine the correct number of processing cycles to keep, but also to decide on an edge to cut the cycle. This is even more important, if multiple cycles sharing some nodes exist. In the worst case, this implies a exponential growth of the graph. If there are cycles sharing some nodes, the number of duplicated nodes is depending on the edge selection algorithm. By cutting the cycles in front of the node with the most incoming edges from inside the cycles, one can resolve at least two nested cycles at the same time.

As soon as there are no cycles left, the resulting the reliability of each node can be evaluated using the created Bayesian Network. This can be done using conditional probability. For identifying the node, on which the probability is conditioning, the so called rules of d-separation are used to identify dependence

between the nodes. Afterwards, we provide an algorithm to calculate the reliability of every component in three steps. The first step is to calculate the relative reliability, which corresponds to the probability, that the component is functional at time t, given some specific components are working. In step two, we calculate the reliability given the same components are malfunctioning. In the end the product rule provides a method for deriving the absolute reliability of each component.

There are still drawbacks in this solution. For calculating the relative reliability, we need to iterate over all possible states of the condition components. Therefore the algorithm still has an exponential run time in the worst case.

1.9 Capabilities and Limitations of the solution

Using the Bayesian Network and Cycle Unrolling, it is possible to handle complex systems containing loops and feedback cycles. For the approach this thesis uses the reliability is calculated for a given time snapshot, disregarding the time it takes for one information to reach the next node. Therefore at each time snapshot, a component can either be working or malfunctioning, but this state cannot change within the calculation.

The application developed in the course of this thesis supports two different reliability probability distributions. These so called failure rate shapes, namely the Exponential Distribution and the Weibull Distribution are described in detail in this thesis. Every component has a failure rate shape information and the values of its parameters for each calculation it performs.

Calculating the reliability in dependence of time, it is not possible to consider reparation of components, since the exact time of failure is unknown. Therefore after a failure, a component will always remain in the fault state.

To simplify the calculation, this thesis neglects recovery from faults and masking failures. It will never be the case, that a second failure will correct another failure. This means only AND and VOTER gate components can interrupt fault propagation.

1.10 Potential applications of the results

The calculated reliability can be used to identify weak points in the system and add additional redundancy components if necessary. Further applications would be to combine this method of calculating the reliability with a method for calculating a possible cause for a fault in a production system. For this, there are methods like the cut set algorithm. This algorithm calculates all possible failures resulting in a fault in the component in question. Combining this knowledge with the calculated reliability, it is possible to determine the most likely cause for the fault.

Another potential application of the dependency graph would be to calculate the impact of a failure in one component. This can be done by a straightforward algorithm calculating every non-recursive path starting at the node with the failure.

1.11 The Complete Process Described in this Thesis

The process covered by this thesis starts from a System defined as a UML model, and transforms it into an intermediate graphical representation, called the Dependency Graph. This graph may still contain cycles. The next step is to transform it into an acyclic version, using cycle unrolling. The resulting graph is a valid Bayesian Network. The Rules of d-Separation define, which nodes are conditional dependent in the graph. With the knowledge of dependencies between nodes this thesis proposes a way of calculating the reliability of each node. The first step is to identify the closest potential common cause nodes and calculate the conditional reliability given all possible states of these nodes. The real reliability is given by

the product of the conditional probability and the real probability of the conditioning nodes This thesis provides an algorithm for identifying those nodes and calculating the real reliability of each node, given the Bayesian Network. This approach is more efficient than a state space search on all possible failure cause nodes, assuming there a less loops in the system.

1.12 Structure of this thesis

Chapter 2 describes related work on the field of Reliability Engineering. In the beginning it summarizes Failure Mode and Effects Analysis. Afterwards it continues with a type hierarchy of reliability models according to Reibman and Veeraraghavan [1991], followed by the more detailed power hierarchy of different models introduced by Malhotra and Trivedi [1994]. These reliability models are described in the order of their expressiveness. Since the Bayesian Network approach is the most significant for this thesis and not covered by in the power hierarchy, the next chapter is describing already existing solutions using this approach and cites papers comparing their expressiveness to other models.

The description of the related work in the field of reliability engineering is followed by two sections on work related to cycle unrolling. Both applications - namely Compiler Construction and Bounded Model Checking - inspired the idea of applying cycle unrolling in this thesis.

The last section connects the related work with the new concepts of this thesis.

In the Chapter 3 we describe the so called *Connected Components System* describing the system handled by the algorithms. The chapter defines the different types of components and the computation units of each component.

The first step of the process is to construct an intermediate graph called the *Dependency Graph*. The Chapter 4 explains the different nodes in it and includes an algorithm for creating the graph from a Connected Components System as described in chapter 3.

Next, we need to transform the *Dependency Graph* into a *Bayesian Network*. First this chapter provides the definitions of a *Byesian Network* including a general example. This transformation consists of two steps: The first step is to identify so called Strongly Connected Components using and algorithm invented by Tarjan [1972]. Afterwards the cycles are processed using cycle unrolling, a method derived from loop unrolling known from compiler construction. In the end of the chapter the result is a valid Bayesian Network. Additionally, this chapter provides a proof of the validity of unrolling cycles only two times.

Chapter 6 goes into detail on how to calculate the reliability. First the preliminaries of probability theory used are explained. Afterwards, we start with simple examples of reliability rules derived from Fault Tree Analysis. The next sections are about the different Failure Rate Distributions. Since the previously explained rules are only suited for networks without Common Cause Failures, we need to provide rules, which can also handle conditional dependent nodes. We explain the rules of d-Separation and how they can be used for identifying dependent nodes in the graph. We introduce some functions returning a special type of nodes. If we provide the state of these nodes, the parents of the input node are always conditional independent. Finally, this chapter provides formulae to calculate the reliability of each node and provides an efficient algorithms, as well as examples of it applied to an example Dependency Graph.

The last chapter is still work in progress.

Chapter 2

Related Work

2.1 Failure Mode and Effects Analysis

One common approach in Reliability Engineering is the Failure Mode and Effects Analysis (FMEA). Arabian-Hoseynabadi, Oraee and Tavner [2010] describe it as powerful tool to identify possible Root Causes and Failure Modes. The causes of are failure are called Root Causes, while Failure Modes are the different ways in which a component might fail. Every Root Cause has an assigned Severity, Occurrence and Detection. The Severity measures the risk of the failure to cause harm, while the Occurrence defines the frequency of a root cause. The Detection is a value, which defines how likely a Root Cause is detected, before a failure is occurring.

Snooke and Price [2012] explain, how FMEA can be done in an on-board aeronautic system. They describe, how Model Based generation of FMEA is performed. For this every potential failure is simulated in order to determine abstract high-level consequences.

2.2 Type-Hierarchy of Reliability Models

There exist different types of Reliability Models [Reibman and Veeraraghavan, 1991]:

- Parts-Count Models:

  These are the least expressive models. They assume, if any of the components in the system fail, the whole system cannot provide its functionality anymore. This type of models is not suited for systems which contain redundancy components.

- Combinatorial Models:

  These models are an extension of the Parts-Count Model, which are able to handle fault-tolerant systems. Fault Trees and the Reliability Block Diagrams are examples of this type.

- State-space models:

  State-Space Models, such as Markov Chains, evaluate the state-space of a system. Every combination of components working or malfunctioning is considered a state of the Markov Chain. These models support the calculation of the mean time to failure (MTTF) and the availability of the model, taking maintainability into account.

2.3 Power-Hierarchy of Reliability Methods

Malhotra and Trivedi [1994] compared some of the different methods for reliability analysis. They discussed four methods, namely Reliability Block Diagrams, Fault Trees Without Repeated Events, Fault Trees with Repeated Events and Reliability Graphs. According to them, there exists a hierarchy between those models, where Fault Trees with Repeated Events are the most powerful technique, followed by the Reliability Graph. Third are two equally expressive techniques, namely the Fault Trees Without Repeated Events and Reliability Block Diagrams. They described even more expressive options as Continuous Time Markov Chains, Generalized Stochastic Petri Nets, Markov Reward Models and Stochastic Reward Nets. Between those, the Continuous Time Markov Chains and Generalized Petri Nets are equivalent, since they can be converted from one to another without loss of capability. Equally, Markov Reward Models and Stochastic Reward Nets are convertible.

2.4 Reliability Block Diagram

The Reliability Block Diagram describes the function of a system [Rausand and Høyland, 2003]. The model represents the logical connections between components involved in a function of a system. If a system has multiple functions, each function has its own Reliability Block Diagram.

Every component in a Reliability Block Diagram is indicated by a Block. If there exists a connection between a system input a and its output b, the specific function of the system is provided. This does not necessarily mean that all functions of the components connecting a and b are achieved. Instead it is sufficient if some specific failure modes do not occur. An example of a Reliability Block Diagram can be seen in Figure 2.1.

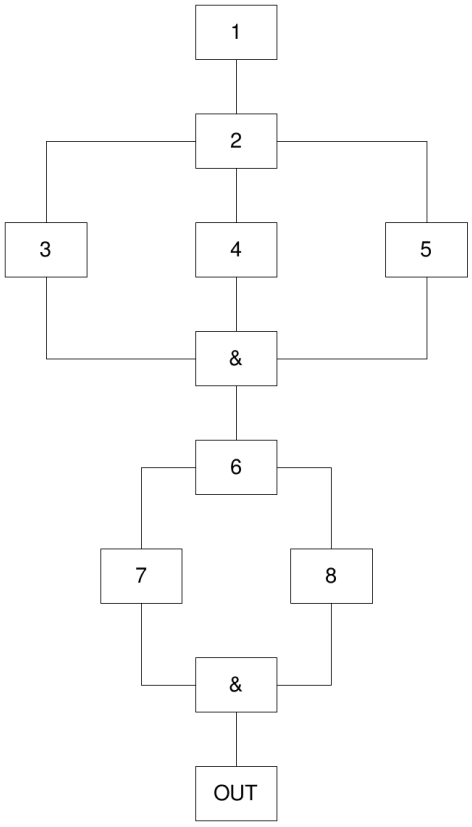


Figure 2.1: An example for a Reliability Block Diagram

2.5 Fault Tree Analysis

2.5.1 FTA: Basics

According to Commission [1998], the fault tree is a graphical model, which views the interrelationships, leading to one specific undesired event. They state that it is not a model of all possible system failures or all possible causes for a specific failure. There exist many different nodes, which mainly depend on the application. The main nodes, which inspired the Dependency Graph of this thesis are:

- Basic Event:

  This event symbolizes the introduction of a new fault.

- Intermediate Event:

  The Intermediate Event, which occurs, if one or more antecedent causes are triggered.

- And Gate:

  The output of an And Gate is faulty, if all inputs are faulty.

- Or Gate:

  The Or Gate outputs faulty information, if any of its inputs is faulty.

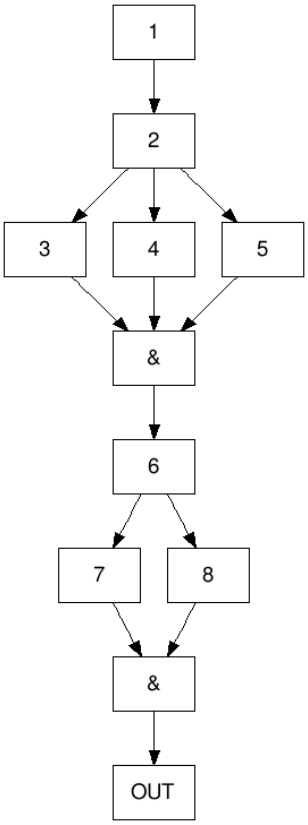


Figure 2.2: This Figure shows a simplification of the system input for the Fault Tree in Figure 2.3. There exist multiple acyclic loops inside the system. There are for example three paths from node "2" to the next And Gate, passing through node "3", "4", or "5".

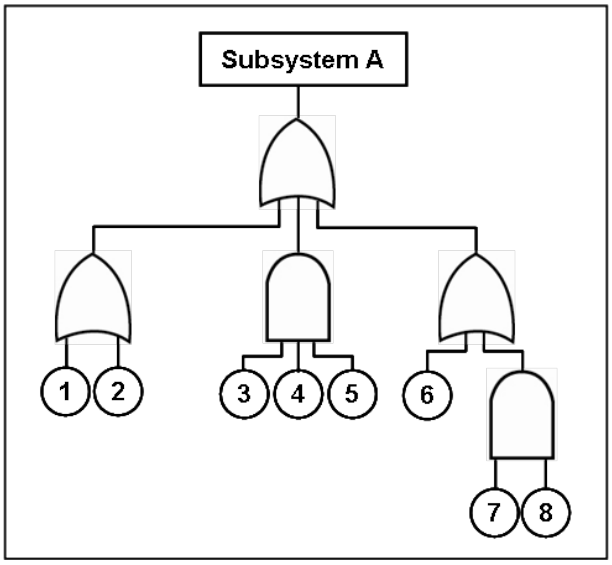


Figure 2.3: The resulting Fault Tree for the system in Figure 2.2. The different parts between the loops are combined via an Or Gate, as are the two And Gates combining the loop paths.
Source: https://en.wikipedia.org/wiki/Fault_tree_analysis
License: Public Domain

2.5.2 Method for handling systems with acyclic loops

An example of a fault tree is depicted in Figure 2.3.

This example also provides a solution for Systems, which contain acyclic loops. A system containing such loops has multiple paths between two nodes. This is a problem, because the fault tree as a tree graph cannot contain such acyclic loops.

This can be handled by splitting the tree into the part before the loop, and the loop itself. These parts are afterwards combined using an Or Gate Node [Commission, 1998]. Figure 2.2 shows the input of a system containing multiple paths between some nodes. This system results in the Fault Tree shown in Figure 2.3.

2.5.3 Handling Systems containing cycles

Vesely et al. [2002] describe a way to cut cycles for the fault tree approach. Since Fault Trees are focusing on one specific fault, they suggest to split a system as shown in in Figure 2.4 into two possible causes for a failure. If the fault in question is in component B, we would combine the two causes "Failure in component B" and "Failure in signal towards B".

2.5.4 Evaluation of possible causes for a failure using Fault Tree Analysis

A commonly used method for finding all possible causes for a failure is called the Minimal Cut Set Algorithm. A Cut Set is a set of all needed primary events, such that the top event will occur. A Cut Set is called minimal, if there exist no other Cut Set for the considered Fault Tree, which is a subset of it. [Kapur and Pecht, 2014]

For example the Minimal Cut Sets of the Fault Tree in Figure 2.3 would be:

$$\{1\}, \{2\}, \{3, 4, 5\}, \{6\}, \{7, 8\} \tag{2.1}$$

This algorithm can be used for evaluating all possible causes for a fault. If combined with the probability for each failure, one could calculate the most likely causes for a fault. The probability for each failure is equivalent to $1 - Rel(Component)$.

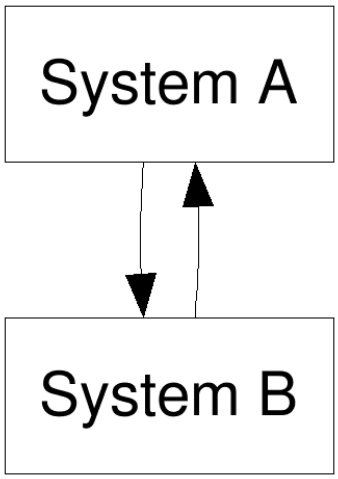


Figure 2.4: A system containing a feedback - cycle between subsystem A and subsystem B.

2.5.5 Dynamic Fault Tree

To be able to handle more complex Networks and sequence dependency, Dugan, Bavuso and Boyd [1992] describe in their paper a so called Dynamic Fault Tree. They introduced special gates, namely the Functional Dependency Gate, the Cold Spare Gate, the Priority And Gate and the Sequence Enforcing Gate.

- The Functional Dependency Gate has a trigger input, as well as some dependent input, in addition to one or multiple non dependent inputs. This trigger can be used to disable the dependent inputs. It can be used to turn off faulty inputs.
- Cold Spare Gates have one primary active unit and one or multiple alternate units. As soon as the active unit's input is triggered, the gate waits until all alternate units are triggered as well. The output of the gate is true, as soon as all inputs have been triggered.
- The Priority And Gate specifies that not only all input events need to be triggered, but they also need to be triggered in the correct order.
- Like the Priority And Gate, the Sequence Enforcing Gate takes multiple inputs, and requires them to occur in a specific order. Contrary to the Priority And Gate, this Gate enforces the events to occur in this specific sequence, all other possible sequences are never considered in the evaluation of the model.

Usually, those Dynamic Fault Trees are analyzed after an automatic conversion to Markov models [Amari, Dill and Howald, 2003]. Amari, Dill and Howald [2003] propose an efficient algorithm for performing an analysis on Dynamic Fault Trees directly, but this method is limited to a qualitative evaluation [Merle et al., 2010].

Andrea Bobbio et al. [2008] show a way how every Dynamic Fault Tree can be modeled by a Bayesian Network as well.

2.5.6 Priority Dynamic Fault Trees with Repeated Events

A restricted version of the Dynamic Fault Tree, the so called Priority Dynamic Fault Tree was described in the paper by Merle et al. [2010]. They limited the dynamic components of the Dynamic Fault Tree and only considered the Priority And Gate and the Functional Dependency Gate. By calculating the so called Sequence Cut Set and transforming it into a canonical form, they are able to calculate the Probability of the Top Event for any failure time distribution of the Basic Events.

2.6 The Petri Net Approach

Petri Nets are defined as a directed Graph consisting of Places P, Transitsions T, directed Edges E and Markings of the graph M [Hura and Atwood, 1988]. Every Transition has an input and an output function, which defines the set of input - or equally output - places. The input places define the conditions before a transition event happens, while the output places describe the conditions that are met afterwards. The markings of the graph represents (data) tokens at a place. They can be used to perform a state-space search for reliability analysis. Figure 2.5 shows an example Petri Net. The transitions in this Figure are for example $t1$ and $t2$, while places are prefixed with a p like $p1$ and $p2$.

This Petri Net is derived from the Fault Tree in Figure 2.6. Every AND Gate was transformed to one transition with multiple input places, while the OR Gates correspond to one transitions for each input place.

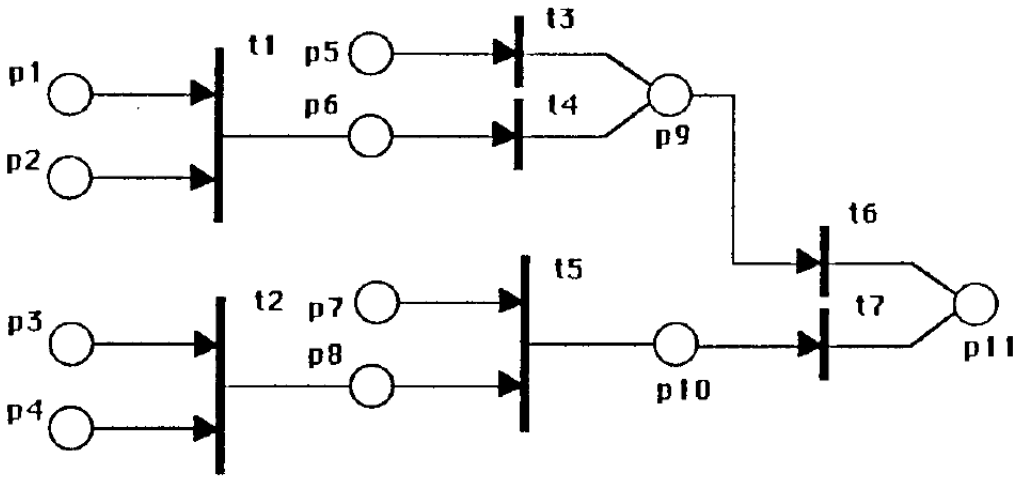


Figure 2.5: This Figure shows an example of a Petri Net. Places are indicated by circles, while the block t symbolizes a transition. This Petri Net corresponds to the Fault Tree shown in Figure 2.6
Source: Paper by Hura and Atwood [1988] ©1988 IEEE

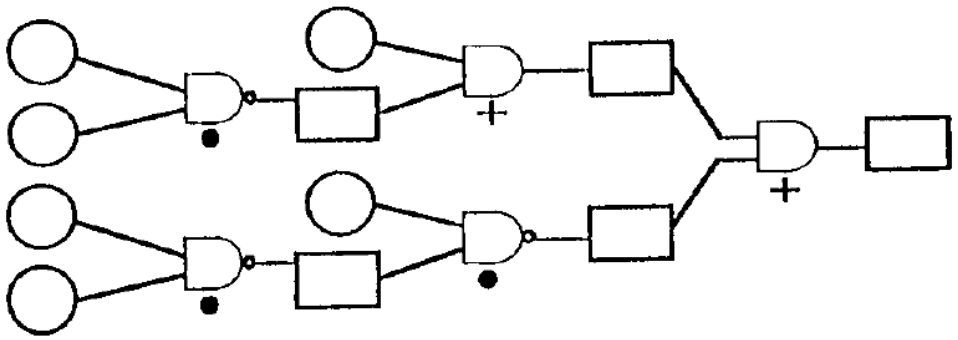


Figure 2.6: This Fault Tree is equivalent to the Petri Net as shown in Figure 2.5. AND Gates are symbolized by the dot below, while OR Gates are indicated by an plus symbol.
Source: Paper by Hura and Atwood [1988] ©1988 IEEE

2.7 Reliability Analysis using Bayesian Networks

A. Bobbio et al. [2001] described in their paper the advantages of using the Bayesian Network approach. They state that Bayesian Networks are more powerful than Fault Trees, since they can handle statistical dependence between inputs. They provided an algorithm for converting a Fault Tree containing And, Or and Voter Gates into a Bayesian Network. Additionally, Bayesian Networks are able to handle uncertainty in the network. This means the gates do not need to be deterministic. They just need to provide a probability for their behavior.

An additional advantage is the simplification of evaluating common cause failures. Common Cause Failures are failures, which are combined in a gate node, with a cause in one commonly shared node. This is equal to the previously described acyclic loop, as shown in Figure 2.2. For example a common cause failure in the first and gate could have a failure in component 2 as root cause.

Khakzad, Khan and Amyotte [2011] support these statements with a case study to compare Fault Tree Analysis and the Bayesian Network approach. They concluded their work stating that Bayesian Networks are more suitable for complex networks and networks containing uncertainty. Bayesian Networks would also be capable of modelling systems with multi-state failures.

A further Paper on this topic was published by Langseth and Portinale [2007], where they explain the main methods necessary for calculating the probability in such a Bayesian Network.

2.8 Loop Unrolling in compiler construction

The concept to unroll the cycle was inspired by the so called Loop Unrolling used in compiler construction. In compiler construction the loops in question are program code parts, which are executed multiple times, like for example a while-loop, or a for-loop. The number of iterations the loop is processed can be either fixed or a variable. These loops can be represented by cycles in an information flow graph. This graph is quite similar to the Dependency Graph of this thesis, which also represents an information flow - not between code statements, but between hardware components.

Cooper and Torczon [2004] describe Loop Unrolling as an old technique, where the body of a loop is replicated multiple times and the logic controlling how often the loop is process is adapted. This concept minimizes the number of comparisons and branches necessary for processing the loop. The loop can be processed more efficiently, since the computer may execute multiple statements in parallel.

2.9 Loop Unrolling in Model Checking

Another application of loop unrolling is Bounded Model Checking for Software [Biere et al., 2009]. The goal of bounded model checking is to verify the correctness of a model, in this case a software project. The principle of bounded model checking to form a propositional formula of the program. A SAT Solver can use this formula to find counter examples violating the correctness properties of the model. If a program contains infinite loops or loops bounded by the user input Bounded Model Checking only takes a limited number of cycle iterations into account. The more efficient way is to unwind each loop separately instead of unwinding the whole program. Again, the loop body is replicated k times, and the while condition is transformed into an if statement.

2.10 Adaption Loop Unrolling to Cycle Unrolling

Software loops are control flow cycles. The cycles we are handling are corresponding to never terminating loops. Therefore we cannot unwind them as often as they are processed. Nevertheless it is possible given some assumptions to show that the reliability of each node in the cycle does not change after

processing the loop two times. In the provided proof, this thesis considers only fixed snapshots of the system, where every component may be either working or not working at a fixed point in time. This feature is used in this thesis to create an acyclic Bayesian Network from a cyclic system by unwinding the cycles two times.

Chapter 3

The Input to our Analysis: The System and its Components

3.1 Definition of a Connected Components System

In this work, we are evaluating so called *Connected Components Systems*.

Definition 3.1.1 (Connected Components System)**.** A *Connected Components System* is a tuple $< C, I, O, \lambda >$ of a set of *components* C, a set of *input ports* I , a set of *output ports* O and a set of *directed connectors* $\lambda \subseteq IxO$.

A port belonging to an embedded component is denoted with the component name followed by a dot and the port ID like $obj.in1$.

The set of ports must fulfill the following conditions, where I_C are all input ports belonging to the component C and O_C are all its output ports:

- $I_c \subset I \qquad \forall\, c \in C$
- $O_c \subset O \qquad \forall\, c \in C$
- $I_{c1} \cap I_{c2} = \emptyset \qquad \forall\, c1, c2 \in C : c1 \neq c2$
- $O_{c1} \cap O_{c2} = \emptyset \qquad \forall\, c1, c2 \in C : c1 \neq c22$
- $I = \bigcup_{c \in C} I_c \cup I_{system}$
- $O = \bigcup_{c \in C} O_c \cup O_{system}$

Definition 3.1.2. A *component* is a part of the system which provides some functions. It may hold a infinite positive number of input and output ports and has a unique identifier.

While each *component* represents a single element in the system, it is possible to arrange components belonging together in groups.

Definition 3.1.3 (Super Component)**.** A *Super Component* is simplified representation of an underlying group of components.

A Super Component provides input and output ports, over which its inner components receive or send information from or to components outside the group. An underlying Connected Components System specifies, how these input and output ports and the contained components are connected.

A Super Component may contain another Super Component, such that it is possible to structure the model in a hierarchical way. This feature supports the user because he can understand the system more intuitively and multiple knowledge engineers can work on creating one system simultaneously.

Definition 3.1.4 (Ports)**.** A *Port* is an interface between a component and its environment.

We distinguish between *Input Ports* for receiving information and *Output Ports* for sending information.

Definition 3.1.5 (Connectors)**.** The Ports of two components communicate over a directed *Connector*.

Every Connector must diverge from either a component's output port or an input port of the Connected Components System itself. Likewise its target must be either a component's input port or an output port a the Connected Components System.

Definition 3.1.6 (Functions of the system)**.** A system's function is to output specific values of any arbitrary type.

These values are calculated by the components of the system. Consequently a system's failure is always caused by a failure in a component.

Definition 3.1.7 (System Failures)**.** A *System Failure* happens if any of the output values of the system deviate from their intended value by an error exceeding a fixed threshold.

We do neither consider the severity of a failure in different output values nor their variance.

Definition 3.1.8 (Functions of the component)**.** Every component besides super components performs calculations on its input. These calculations produce new values, where each different output is forwarded over a separate output port.

These calculations of a component can fail independently. The user has to define a probability of a components function breaking down within a certain time span. He can provide this information by annotating every output port with a probability distribution and its parameters. If he does not provide any, we assume an ideal component.

Definition 3.1.9 (Fault Propagation)**.** Ever component's calculation is based on its input values. If any is incorrect, the component cannot provide its functions anymore. If no redundant component corrects this error, it will spread through the model. We call this phenomenon *Fault Propagation*.

This thesis limits itself to faults, which never mask nor repair themselves nor are repairable.

Example 3.1.1: Connected Component System with two components

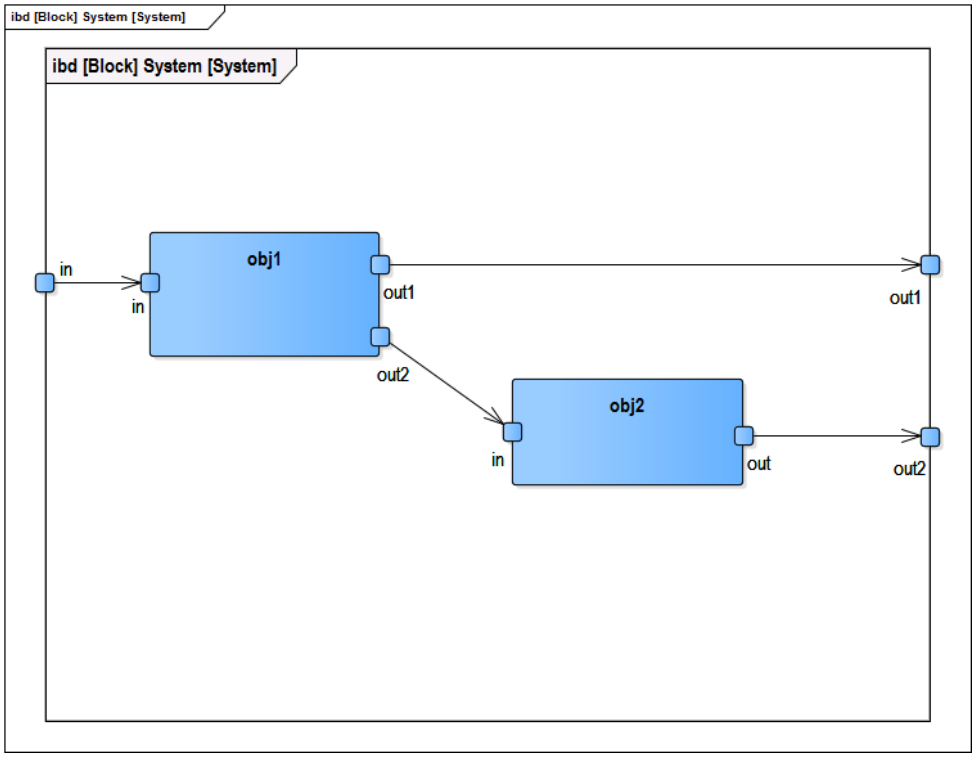
An example for a connected component system is a tuple $< C, I, O, \lambda >$, with

$$
\begin{aligned}
C &= \{obj1, obj2\},\\
I &= \{in, obj1.in, obj2.in\},\\
O &= \{out1, out2, obj1.out1, obj1.out2, obj2.out\}\\
\lambda &= \{[in, obj1.in], [obj1.out1, out1], [obj1.out2, obj2.in], [obj2.out, out2]\}
\end{aligned}
$$

3.2 Component's Composition

Figure 3.2 shows the composition of a generic component. It consists of n input ports $i_1, ..., i_n$, a preprocessing function $f(.)$ and m output calculations $g_1(.), ..., g_m(.)$. The output calculation might fail due to environmental influences or hardware faults, such that the output of the calculation $g_k(.)$ can derive from the ideal value by an error e_k. If this error exceeds a certain threshold, the calculation's output is incorrect. We say a failure occurred in the calculation.

The task of the preprocessing function is to detect and discard invalid input values. The capabilities of this function depend on the type of the component. Some are able to handle an arbitrary number of incorrect inputs, as long as one is valid, others require a specific number of inputs to be correct. Components without a type do not perform any preprocessings.



Example 3.1.2: Two nested Connected Components Systems

Figure 3.1 shows a connected component system containing a nested Connected Components System. They are defined as:

$$
\begin{aligned}
System :< & \{superComponent\}, \\
& \{in1, in2, superComponent.in1, superComponent.in2\}, \\
& \{out, superComponent.out\}, \\
& \{[in1, superComponent.in1], [in2, superComponent.in2], [superComponent.out, out]\} >
\end{aligned}
$$

$$
\begin{aligned}
superComponent :< & \{basic1, basic2\}, \\
& \{in1, in2, basic1.in, basic2.in1, basic2.in2\}, \\
& \{out, basic1.out, basic2.out\}, \\
& \{[in1, basic1.in], [in2, basic2.in2], [basic1.out, basic2.in1], [basic2.out, out]\} >
\end{aligned}
$$

3.3 Component Types

To avoid incorrect information spreading over the system a system can contain redundant components. These component or groups of components calculate the same value. If any of these values deviate from their intended value, special components can preprocess their input and discard invalid values. This thesis supports the following types of components:

- *OR Components*
- *AND Components*
- *VOTER Components*

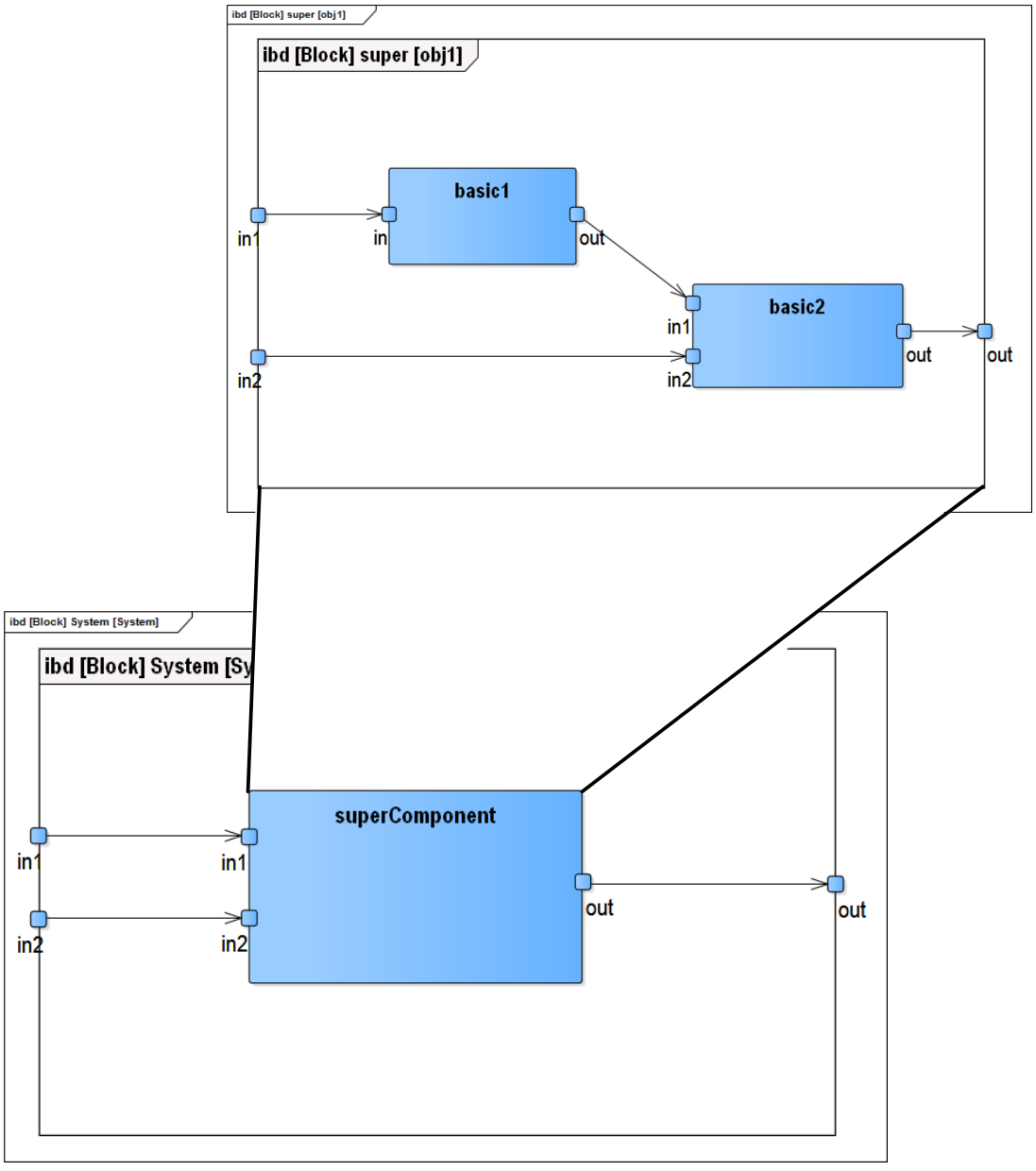


Figure 3.1: UML Diagram of two nested Connected Components Systems. The formal definition of these systems is included in Example 3.1.2

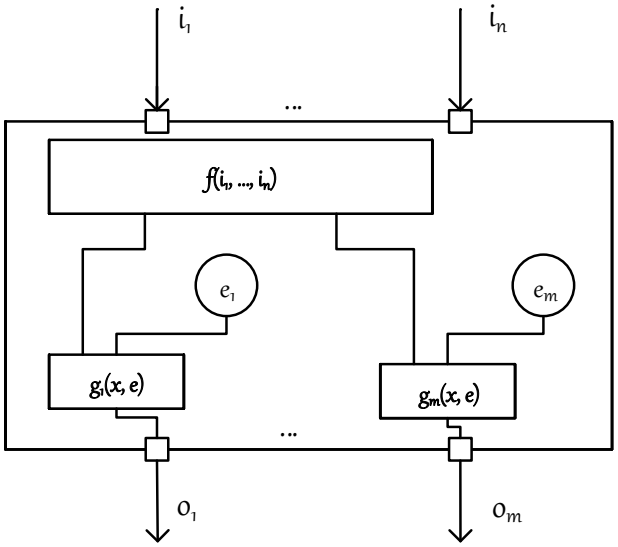


Figure 3.2: This Figure shows the composition of every component. Components receive input values over an arbitrary number of input ports $i_1, ..., i_n$. Those inputs are preprocessed by a special function $f(i_1, ..., i_n)$. This function depends on the type of the component. The output of the function is then forwarded to the main calculation unit. Every calculation g_k is influenced by an error e_k, which can make the calculation fail. Afterwards the component sends the output of each calculation unit - which can be either valid or invalid - over a separate output port to its children.

OR components are the most basic type of components. They do not perform any additional calculation of the received information. If any of its input ports delivers incorrect information, it will always produce incorrect outputs. It forms a logical OR between faults in its input values.

Contrary an AND Gate component forms an AND relationship between its inputs. If and only if all input values of the component are incorrect, it will propagate a fault. This means as long as any input is correct, the components can perform its calculations correctly.

Last, a VOTER component has a so called vote count. It defines how many inputs need to be correct such that the component performs its calculations on the correct value. The vote count needs to be a natural number greater equal zero and smaller or equal the number of inputs to the component.

Regardless of the input preprocessing type, any of these component's function can fail.

Definition 3.3.1 (Typed Connected Components System)**.** A *Typed Connected Components System* is a specialization of a Connected Components System, which allows to specify additional types of components. The additional parameter T in the tuple $< C, I, O, \lambda, T >$ defines the type of each component.

T may be incomplete or not specified. We assume that any component c without a type $t_c \in T$ is a OR component.

Example 3.3.1: Connected Components System with different types of basic components

Figure 3.3 shows a system containing an AND Component and a 2- VOTER component. The system is defined as:

$$
\begin{aligned}
&System: \\
&\quad <C = \{and, vote\}, \\
&\quad I = \{in1, in2, in3, in4, and.in1, and.in2, vote.in1, vote.in2, vote.in3\}, \\
&\quad O = \{and.out, vote.out, out\}, \\
&\quad \lambda = \{[in1, and.in1], [in2, and.in2], [in3, vote.in2], [in4, vote.in3], [and.out, vote.in1], [vote.out, out]\} \\
&\quad T = \{T_{and} = AND, T_{vote} = 2 - VOTER\} >
\end{aligned}
$$

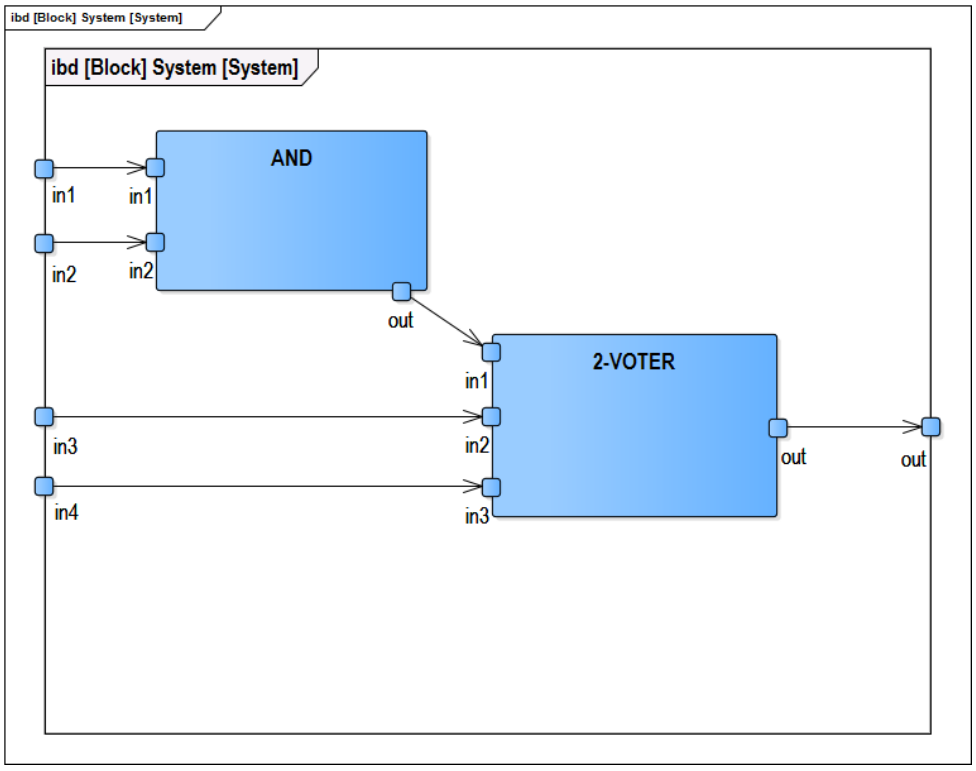


Figure 3.3: UML Diagram of a Connected Components System with and AND and a 2-Voter component 3.3.1.

Chapter 4

The Dependency Graph: Structure and Creation

4.1 Introduction

The algorithm we are using to evaluate the reliability operates on a Bayesian Network. This chapter describes how to transform a Connected Components System as defined in Chapter 3 into an intermediate representation called the Dependency Graph. This intermediate representation is already a directed graph, but if the system contains feedback loops, it will not be acyclic. Different from the Bayesian Network, we do not only consider one type of nodes. Depending on their probability distribution, we classify the nodes into different types. Every type of node has its own visual representation. We call this type of graph Dependency Graph.

4.2 Definition of the Dependency Graph

Definition 4.2.1 (Dependency Graph)**.** A *Dependency Graph* is a Tuple $< N, E >$, where N is a set of nodes and E a set of directed edges between those nodes.

As in a Bayesian Network, a Node can take two states: $True$ and $False$. These states are not known at the time of the evaluation. Instead the goal is to calculate the probability for each node to be in either state $True$ or $False$.

Definition 4.2.2 (Node)**.** Every *Node* $n \in N$ is an object with the meta data *name*, *type*, *type annotation*. As in a Bayesian Network, every node has to provide a *probability distribution*.

The name of a Node is its unique identifier, while its type must be one of the following values:

- Failure
- Forward
- Preprocessing
- Output

If the type of a node is *Preprocessor* it needs an additional type annotation. This annotation represents the vote count of the corresponding component.

The *probability distribution* specifies the probability of a node being in state $True$ given its parent's states. If a node's type is *Failure*, it never has parents. Instead it's probability distribution takes the time as parameter.

It is possible to perform a reliability analysis using only the structure of the graph and the probability distributions of each node. The meta data is only used for the graphical representation and limits the capabilities of the applied technique to the application described in this thesis.

4.3 Nodes in the Dependency Graph

The nodes of the Dependency Graph are derived from the structure of the components. Every part of the component is represented by a separate node. Consquently there are four main types of nodes:

- Forward Nodes (created from ports)
- Preprocessing Nodes (created from the preprocessing calculation)
- Failure Nodes (created from the error influence on the calculation)
- Output Nodes (created from the output calculation)

4.3.1 The Forward Nodes

Every *Forward Node* represents a port in the Connected Components System. They may only have one parent and their state is always equal to their parent's state. The visual representation of this kind of nodes is a ellipse with a dotted border containing the ID of the port.

4.3.2 Preprocessing Nodes

If a component is of the type *AND* or *VOTER*, it performs a preprocessing on its input values. This preprocessing is capable of eliminating invalid values. In the graph, a trapezium symbolizes a *Preprocessing Node*. The annotatations `AND` and `k:n` specify, which type of proprocessing the component performs. In this case `AND` stands for an and preprocessing type, and `k:n` for a k-out-of-n voter type. Additionally, the node usually displays the name of the component. Every *Preprocessing Node* has at least one parent and its state depends on both the parents' states and the type of the preprocessing function. This is defined via

4.3.3 Failure Nodes

As explained in Chapter 3, ever main calculation inside a component can fail due to environmental influences or hardware failures. This is indicated by an error, which falsifies the calculation results. In the graph we display this influence by an *Failure Node*. A Failure is never depending on any other component or component part. Therefore a *Failure Node* has no parents. Their state depend solely on the time span the system has been operating so far. The visual representation for this node is a box containing a lightning bolt.

4.3.4 Output Nodes

Finally, an *Output Node* represents the final calculation of the data send over one output port. This Nodes state is both influenced by the output of the *Preprocessing Node* and the failure influence from the *Failure Node*. If any of these values is invalid, the calculation will fail. Therefore the *Output Node* is a logical OR of all parent states.

4.4 Relations of Nodes in the Dependency Graph

In order to describe the operations used on this graph, it is necessary to define the relation of nodes in the Dependency Graph.

Definition 4.4.1 (Parent Nodes)**.** The *Parent* of a node $n \in N$ are all nodes $p \in P(n) \subset N$, if there exists a directed edge $e \in E$, from p towards n.

Definition 4.4.2 (Child Nodes)**.** The *Child* of a node $n \in N$ are all nodes $c \in C(n) \subset N$, for which there exists a directed edge $e \in E$, from n towards c.

Definition 4.4.3 (Predecessors)**.** The *Predecessors* of are node are all node, which are a parent of the node itself, or a parent of any other predecessor of the node.

Definition 4.4.4 (Successors)**.** The *Successor* of a Node are all Nodes, which are children of the node itself or a child of any other successor of the node.

4.5 State of Nodes

As described in the Introduction, each Node can either take the value $True$ or $False$. In the case of this application, the state of each node represents the corecctness of the underlying calculation. In detail this means:

- A *Forward Node* with the state $True$ signals, that the port it corresponds to sends invalid information.
- A *Preprocessor Node's* state is set to $True$, if and only if the preprocessing fails, due to too many incorect input values.
- A *Failure Node's* state is true, when hardware failures or environmental influences are enough to alter the main calculation, such that it outputs incorrect results.
- If either the output of the preprocessor - or the component's input if none - is faulty or the calculation is badly influenced by the environment or hardware faults, its output will always be incorrect. Therefore an *Output Node* takes the logical OR of all its parents' states to calculate its own one.

4.6 Transforming a Connected Components System's Component into a Graph

We transform a Connected Components System's component into a graph by replacing one part of the component after each other and connecting them in the correct order. Input ports' *Forward Nodes* are usually contained in the graph, because they give it a clearer structure, but output ports' *Forward Nodes* are usually neglected. After creating the *Forward Nodes*, we add an *Preprocessing Node* if the component has an preprocessing unit. Afterwards, we direct an edge from a *Failure Node* and the edge starting at the *Preprocessing Node* to the *Output Node*.

A generic composition of a component is shown in Figure 3.2 (Chapter 3). Figure 4.2 depicts the graph generated from the component. If a component does not contain a preprocessing unit - meaning it is neither a AND nor a VOTER component - there is no *Preprocessing Node* in the graph (Figure 4.1).

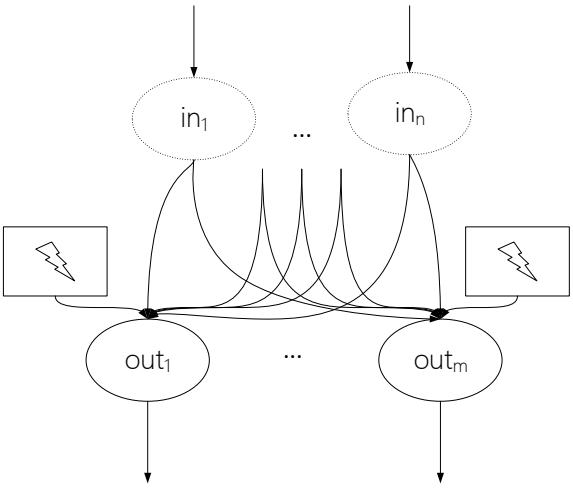


Figure 4.1: If transforming a single OR component into a Dependency Graph, it consists of nodes for the input ports (*Forward Nodes*), nodes for the failure influence (*Failure Nodes*) and nodes for the output calculation (*Output Nodes*).

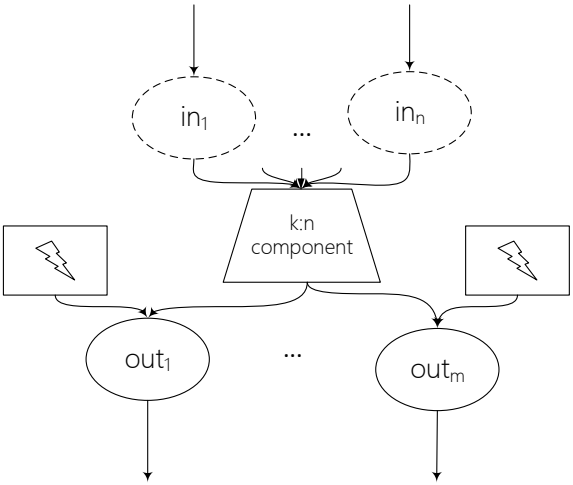


Figure 4.2: In contrast to the OR Component, a VOTER or AND Component has an additional *Preprocessing Node* between its input *Forward Nodes* and its *Output Nodes*. This node has an annotation for the user, which is either the string "AND", or of the Form "k:n", where k is the vote count of the component and n the number of input ports on the component.

4.7 Creating the Graph from a full Connected Components System

When we generate a Dependency Graph from a Connected Components System, we create a subgraph for every component and link them together according to the connections in the model. If there exists a super component in the model, its inner system is first generated and afterwards linked into the outer graph using *Forward Nodes* at the position of the ports.

Example 4.7.1: Generation a Dependency Graph from a Connected Components System

Chapter 3 provides some example Connected Components System. One of it - Example 3.1.1 - is defined as:

$$
\begin{aligned}
C &= \{obj1, obj2\},\\
I &= \{in, obj1.in, obj2.in\},\\
O &= \{out1, out2, obj1.out1, obj1.out2, obj2.out\}\\
\lambda &= \{[in, obj1.in], [obj1.out1, out1], [obj1.out2, obj2.in], [obj2.out, out2]\}
\end{aligned}
$$

The resulting Dependency Graph consists of the nodes

$$
\begin{aligned}
N = \{&System.in, Obj1.in, Obj1.out1, Obj1.out2, Obj2.in, Obj2.out, System.out1,\\
&System.out2, Failure(Obj1.out1), Failure(Obj1.out2), Failure(Obj2.out)\}
\end{aligned}
$$

The graph with nodes and edges is depicted in Figure 4.3.

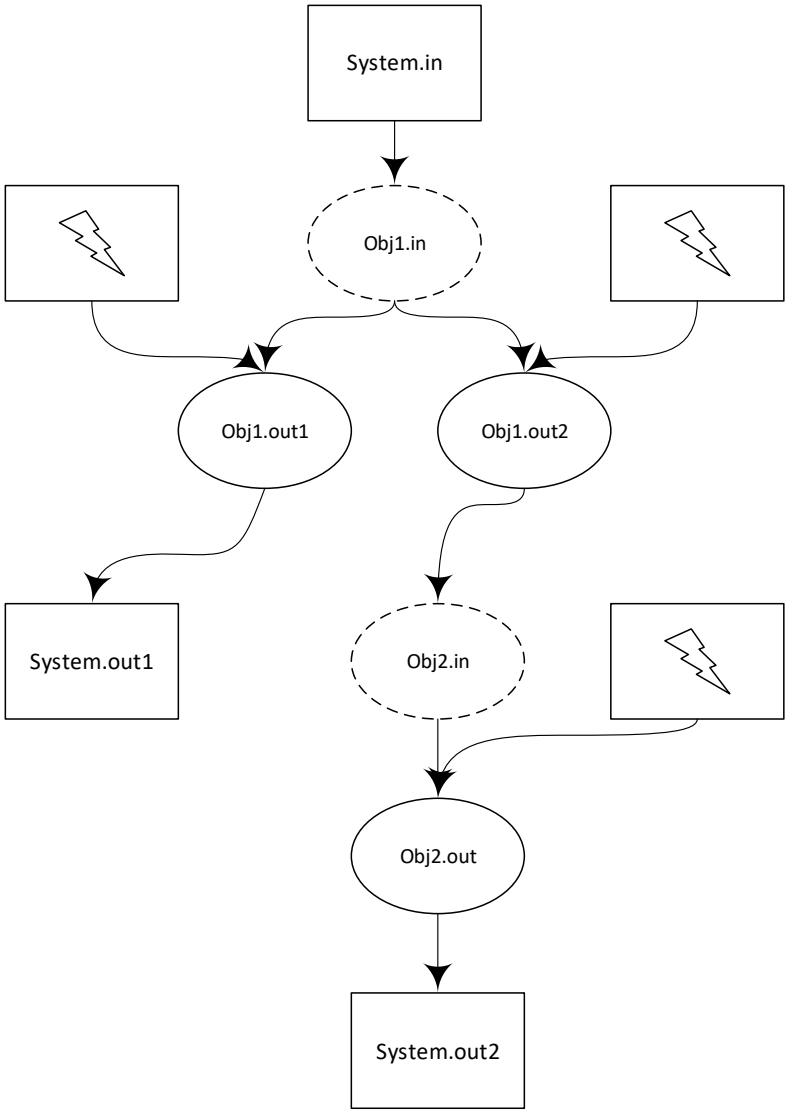


Figure 4.3: This Figure shows the Dependency Graph corresponding to the Example 4.7.1

Chapter 5

Transformation into a Bayesian Network: Applying Cycle Unrolling

5.1 Introduction

The previous chapter described how to generate a Dependency Graph from a Connected Components System. Unfortunately this graph does not fulfill all properties of a Bayesian Network. It still contains the feedback loops contained the Connected Components System. These cycles in the graph need to be resolved. Otherwise we cannot use the algorithms based on Bayesian Networks for evaluating the reliability. The approach we take for resolving the cycles is called Cycle Unrolling, inspired from the method Loop Unrolling known from Compiler Construction.

5.2 Definition of a Bayesian Network

The goal of this chapter is to transform a Dependency Graph into a valid Bayesian Network.

According to Khakzad, Khan and Amyotte [2011] a Bayesian Network is defined as follows:

Definition 5.2.1 (Bayesian Network)**.** A Bayesian Network is a directed acyclic graph. Each node in the graph represents a variable and arcs in the graph stand for direct casual relationships between the linked nodes.

Therefore a Bayesian Network is a Tuple <N,E,$\mathcal{P}$>, where N is a set of Nodes, E is a Set of directed Edges and $\mathcal{P}$ are the conditional probabilities of each Node taking the value 1 or 0 given the value of its parent nodes.

5.3 Definition of Graph Structures

The imported Dependency Graph is already a directed graph, but it may contain cycles.

Definition 5.3.1 (Paths)**.** A *Path* $P(n_1, n_2)$ from node n_1 towards node n_2 in a graph is a sequence of connected nodes, where the first is n_1 and the last is n_2. For a *directed Path* all edges between the nodes must direct from the prior node to the subsequent node.

Barber [2012, chapter 1, section 2.1] distinguishes in his book between Loops and Cycles. He defines them as:

Definition 5.3.2 (Loops)**.** A *Loop* in a graph is a undirected path, which starts and ends in the same node.

Example 5.2.1: Example of a simple Bayesian Network

An example Bayesian Network would be the relationship between a car accident and the possible factors `Drunken Driver` and `Malfunctioning Car`. Both factors can occur independent of each other, but both have an influence on the likelihood of a car accident. Therefore there leads a directed edge from both factors towards the `Car Accident` node (Figure 5.1).
Some fictitious values for $\mathcal{P}$ would be:

$$P(DD) = 0.85$$
$$P(MC) = 0.95$$
$$P(CA \mid DD = T, MC = T) = 0.75$$
$$P(CA \mid DD = F, MC = T) = 0.70$$
$$P(CA \mid DD = T, MC = F) = 0.65$$
$$P(CA \mid DD = F, MC = F) = 0.05$$

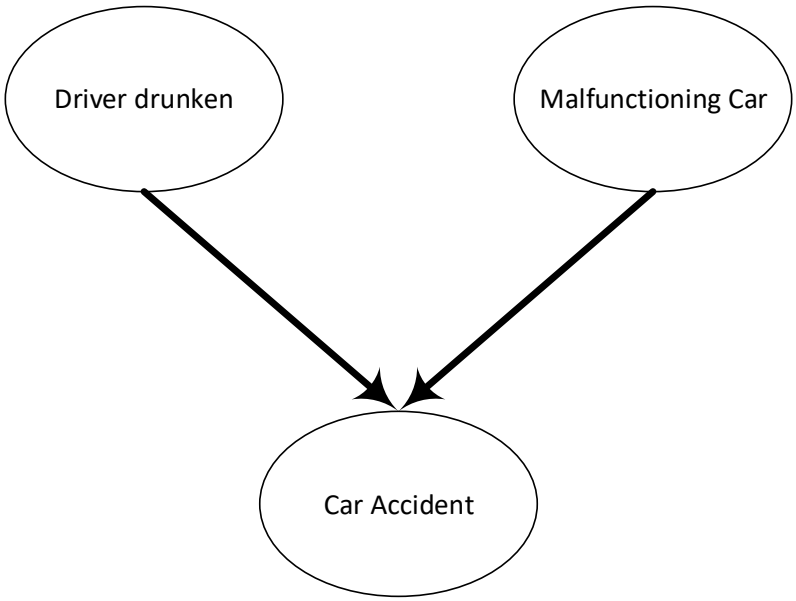


Figure 5.1: An Example of a Bayesian Network. Both Factors `Driver Drunken` and `Malfunctioning Car` may cause a `Car Accident`.

Definition 5.3.3 (Cycles)**.** A *Cycle* in a graph is a directed path from one node to itself. A graph without Cycles is called *acyclic*.

Later, we will use an algorithm invented by Tarjan [1972], which identifies Strongly Connected Components in a cyclic graph.

Definition 5.3.4 (Strongly Connected Graph)**.** A graph $G =< N, E >$ is called strongly connected, if for every two nodes $n_1, n_2 \in N$ there exists a directed path, that connects n_1 and n_2.

Definition 5.3.5 (Strongly Connected Components)**.** Strongly Connected Components are all maximal subgraphs $SCC < N, E > \subseteq G$ of a Graph $G < N, E >$, which are strongly connected.

Every cycle is therefore part of a strongly connected component.

5.4 Tarjan's strongly connected component algorithm

In order to perform probabilistic inference the Dependency Graph needs to be a Bayesian Network. Therefore it is necessary to transform it into an acyclic graph. This is done by unrolling all cycles in the graph. For this, we first need to identify all the cycles. An efficient algorithm for solving this problem is Tarjan's strongly connected component algorithm. It identifies all strongly connected components in the graph. Considering a strongly connected component consists of one or multiple cycles, we can use those to unroll the cycles later on.

```
1  algorithm tarjan is
2  input: graph G = (V, E)
3  output: set of strongly connected components (sets of vertices)
4
5  index := 0
6  S := empty
7  for each v in V do
8    if (v.index is undefined) then
9      strongconnect(v)
10   end if
11 end for
12
13 function strongconnect(v)
14   // Set the depth index for v to the smallest unused index
15   v.index := index
16   v.lowlink := index
17   index := index + 1
18   S.push(v)
19   v.onStack := true
20
21   // Consider successors of v
22   for each (v, w) in E do
23     if (w.index is undefined) then
24       // Successor w has not yet been visited; recurse on it
25       strongconnect(w)
26       v.lowlink  := min(v.lowlink, w.lowlink)
27     else if (w.onStack) then
28       // Successor w is in stack S and hence in the current SCC
29       v.lowlink  := min(v.lowlink, w.index)
30     end if
31   end for
32
33   // If v is a root node, pop the stack and generate an SCC
34   if (v.lowlink = v.index) then
35     start a new strongly connected component
36     repeat
37       w := S.pop()
38       w.onStack := false
39       add w to current strongly connected component
40     while (w != v)
41     output the current strongly connected component
42   end if
43 end function
```

Listing 5.1: Tarjan's strongly connected component algorithm, source:
https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

Example 5.4.1: Strongly Connected Component Algorithm shown on an example

This example will demonstrate how the algorithm works by a simple example. To simplify the demonstration, we operate on a basic graph without any type information of the nodes. Figure 5.2 shows the graph used as input. It contains the cycles $a-b-c-a$ and $c-d-c$. Harder to see is the third cycle $a-b-c-d-c-a$.

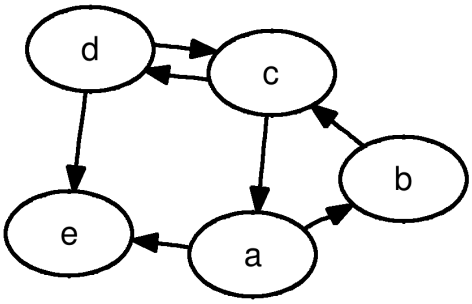


Figure 5.2: We will use this Graph for the demonstration of the SCC algorithm. It contains the cycles $a-b-c-a$, $c-d-c$ and $a-b-c-d-c-a$.

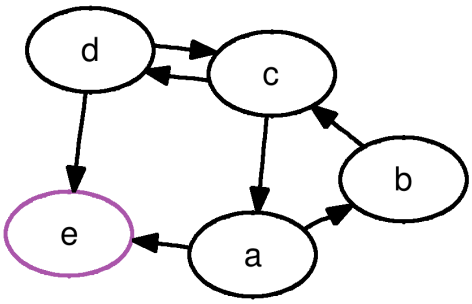


Figure 5.3: The first Strongly Connected Component that is returned is e. It has no child and does therefore not call the function recursively.

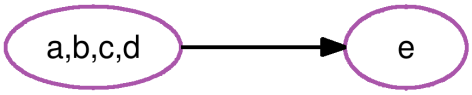
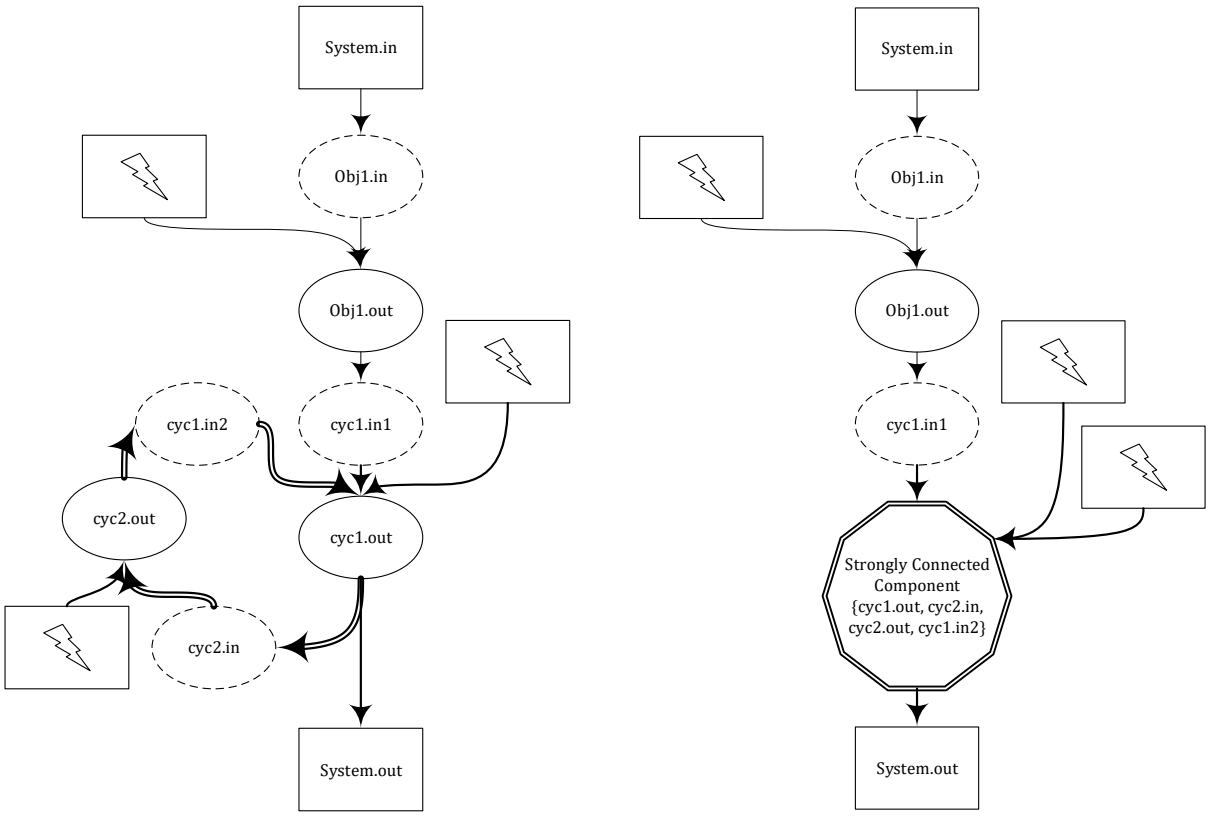


Figure 5.4: The next and last Strongly Connected Component, which the algorithm returns is a, b, c, d, since those four nodes are all contained in the same cycle. Now no further node is left to process and the program halts.

Example 5.4.2: Cycle and Strongly Connected Component

An example for a Dependency Graph of a system containing cycles is depicted in Figure 5.5a. Tarjan's Algorithm detects the nodes inside this cycle as part of a Strongly Connected Component (Figure 5.5b).



(a) This Dependency Graph contains a cycle marked with double line edged.

(b) Dependency Graph of Figure 5.5a, where the cycle is replaced by a Strongly Connected Component.

Figure 5.5: Replacing a Cycle by a Strongly Connected Component

5.5 Unrolling Cycles

Once we identified all Strongly Connected Components containing more than one node, we know all nodes contained in cycles. Next we want to unroll all cycles in a way that retains the child - parent relationships between nodes.

Inside a cycle all component's outputs depend on the output of all other components inside the cycle. We cannot calculate the reliability of the nodes using conditional probabilities, because those probabilities are defined in a infinite recursion.

To handle this problem, we unroll the cycle. The main principle of this approach is: All connectors inside the cycle are directed into the cycle. However, the last edge - which would complete the cycle - is redirected to a copy of the whole cycle. After the second cycle iteration, it is again redirected to the next copy. This continues as often as the data passes through the cycle - in our case infinitely often. This would guarantee, that the reliability converges towards the correct value.

Of course it is not possible to build a finite Bayesian Network we can work with by unrolling an infinite number of times. Still it is possible to get the exact values of the reliability of the nodes, under some limitations:

- Every calculation of the reliability for the whole system is only for one specific point in time.
- The time calculations and transmissions take are neglected.
- Since the reliability of the system is calculated for a specific time snapshot, hardware state and the environmental factors do not change. This implies that the state of every *Failure Node* is fixed for the whole calculation.

When we assume these limitations, we can show, that the reliability of each output of a node inside the Strongly Connected Component cannot change after the cycle has been processed two times. Therefore it is enough to unroll the cycle two times.

5.5.1 Proof for limiting the number of unrolling iterations

This proof considers any arbitrary node N inside a Strongly Connected Component. This node is receiving information from outside of the component, which can be either correct (1) or incorrect (0). The state of this input is contained in the boolean vector $\vec{IN}(N)$. The output of the node is a single value, which can similarly be either correct (1) or incorrect (0). Its state is denoted by $O(N)$. The output of the node can be both forwarded to the other nodes in the strongly connected component and to the outside. All further nodes of the Strongly Connected Components are hidden inside a node group S. This node group returns some data to the node N. However, in this case, the data is send over multiple input ports, which implies that it consists of multiple values, which can again take the state correct and incorrect - a boolean vector $\vec{O}(S)$. Since only the state of N is considered, all output towards other nodes than N may be neglected. Last, there is a binary input vector $\vec{IN}(S)$ representing information send from outside of the Strongly Connected Component to the node group S.

A visual representation of this abstraction is shown in Figure 5.6.

Every node besides Cause Event Nodes can be represented by a voter with a specific vote count. The proof uses this fact and shows for any voter node n, that its output's state does not change after two cycle iterations. The output of node N in cycle Iteration i is described by the series:

The resulting unrolled Strongly Connected Component of the abstracted model is shown in Figure 5.8

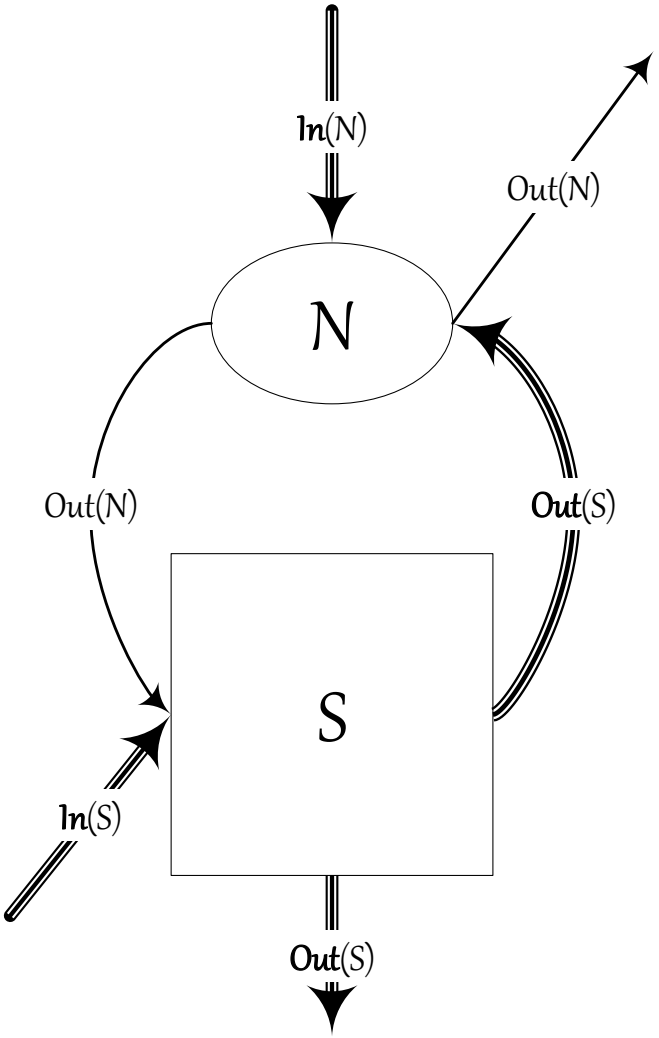


Figure 5.6: This Figure shows the abstraction of a Strongly Connected Component. The proof shows, that the reliability of any arbitrary node in the SCC remains constant after two cycle iterations. The input's state from outside of the SCC towards the considered Node is called $In(\vec{N})$, its boolean single output's state is named $Out(N)$. All other nodes of the SCC are abstracted into one node group. For the proof, this node group may have any structure and an arbitrary number of output towards N. Their state is called $Out(\vec{S})$. Any other output of the group of nodes towards outside of the Strongly Connected Component camn be neglected. Furthermore $In(\vec{S})$ symbolizes the state of the input to the group node S from outside of the SCC.

Example 5.5.1: Cycle Unrolling applied to a small Dependency Graph

The resulting graph after unrolling the cycle in Figure 5.5b is shown in Figure 5.9.

5.6 Determine the best way to cut as many cycles as possible

If the Strongly Connected Component contains more than one cycle the problem becomes harder. Now it is necessary to determine the node where the unrolling of the cycle should start. Example 5.6.1 shows

$$\vec{Out}(S)_0 = \vec{1}$$
$$Out(N)_0 = 1$$
$$\forall i > 0 :$$
$$Out(N)_i = |\vec{In}(N)|+|\vec{Out}(S)_{i-1}|>= votec(N)$$
$$Out_m(S)_i = |\vec{In}_m(S)|+Out(N)_{i-1} >= votec_m(S)$$

$$Out_m(S)_i = \begin{cases} 0 & \text{if} \quad |\vec{In}_m(S)|+1 < votec_m(S) \quad (= P_1(m)) \\ 1 & \text{if} \quad |\vec{In}_m(S)| \quad >= votec_m(S) \quad (= P_2(m)) \\ Out(N)_i & \text{if} \quad |\vec{In}_m(S)|+1 = votec_m(S) \quad (= P_3(m)) \end{cases}$$

$$Out(N)_i = |\vec{In}(N)|+ \sum Out_m(S)_{i-1} >= votec(N)$$
$$Out(N)_i = |\vec{In}(N)|+0|\{m : P_1(m)\}|+1|\{m : P_2(m)\}|+Out(N)_{i-1}|\{m : P_1(m)\}|>= votec(N)$$
$$Out(N)_i = Out(N)_{i-1}|\{m : P_1(m)\}|>= votec(N) - |\vec{In}(N)|-|\{m : P_2(m)\}|$$

$$Out(N)_i = Out(N)_{i-1} >= \frac{votec(N) - |\vec{In}(N)|-|\{m : P_2(m)\}|}{|\{m : P_1(m)\}|}$$
$$Out(N)_i = Out(N)_0 >= \frac{votec(N) - |\vec{In}(N)|-|\{m : P_2(m)\}|}{|\{m : P_1(m)\}|}$$
$$Out(N)_i = \frac{votec(N) - |\vec{In}(N)|-|\{m : P_2(m)\}|}{|\{m : P_1(m)\}|} <= 0$$

$$Out(N)_i = Out(N)_j \qquad \forall i, j > 0$$

Figure 5.7: Proof that two unrolling iterations are enough

Example 5.6.1: Strongly Connected Component containing more than one cycle
Figure 5.10 depicts a Dependency Graph, with one Strongly Connected Component (*cyc*1.*out* *cyc*2.*in*, *cyc*2.*out*, *cyc*1.*in*2, *cyc*3.*in*, *cyc*3.*out*, *cyc*1.*in*3). This Strongly Connected Component contains multiple cycles sharing one node, namely *cyc*1.*out*. When cutting one cycle for example in front of the node *cyc*1.*in*2, the graph becomes even more complex, because the second cycle also has to be duplicated (Figure 5.11). A better way is to cut the cycle in front of the node *cyc*1.*out*, where all cycles are unrolled at the same time (Figure 5.12.

a case where two different nodes lead to results of different complexity.

The best strategy for choosing edges to cut is to look at the node with the most edges inside the Strongly Connected Component ending in it. This edge is contained in the most cycles. Therefore all those cycles can be resolved at the same time, when cutting all those edges ending in this node.

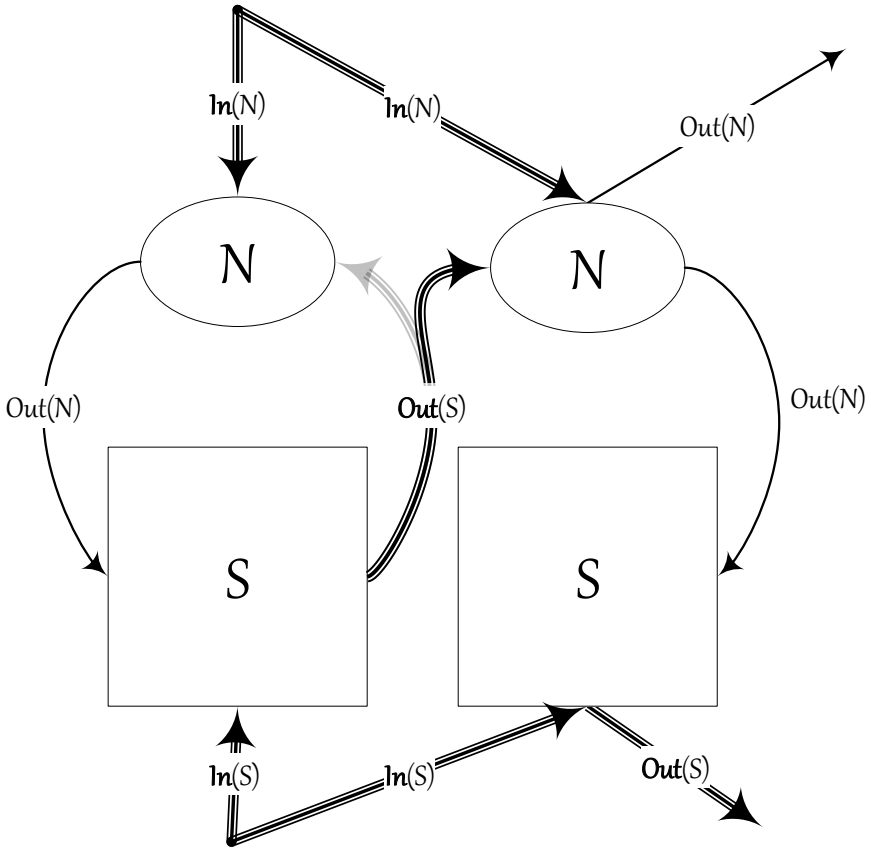


Figure 5.8: This Figure shows Dependency Graph resulting from an unrolled Strongly Connected Component. The output state of the nodes do not change after two cycle iterations. Therefore, we let the output of the nodes in the Strongly Connected Component start at the second iteration. The redirected edge $Out(S)$ is displayed in a light gray color, while the new edge is drawn as a black arrow.

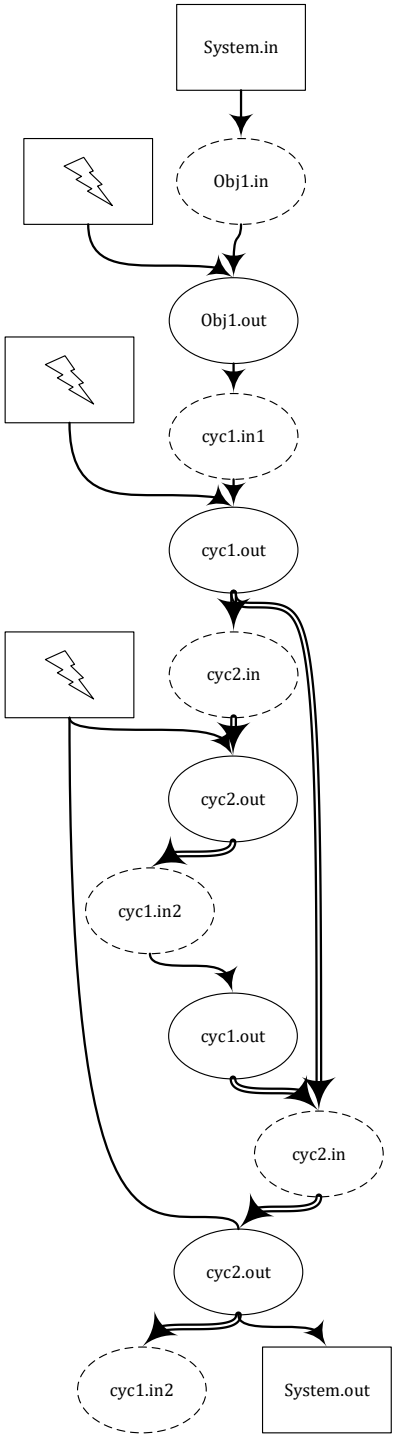


Figure 5.9: This Figure shows the Transformed Dependency Graph from Figure 5.5b.

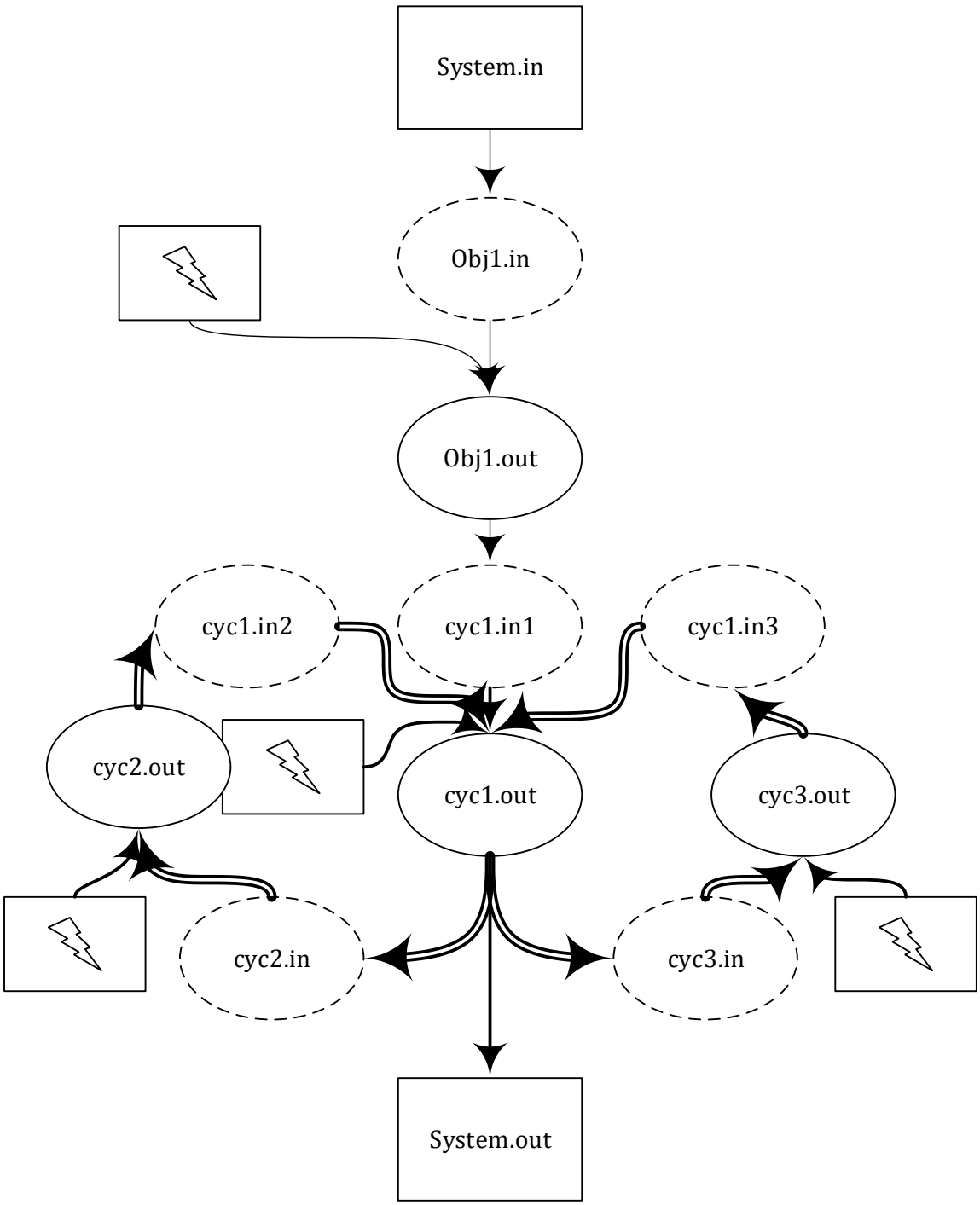


Figure 5.10: Strongly Connected Component containing more than one cycle

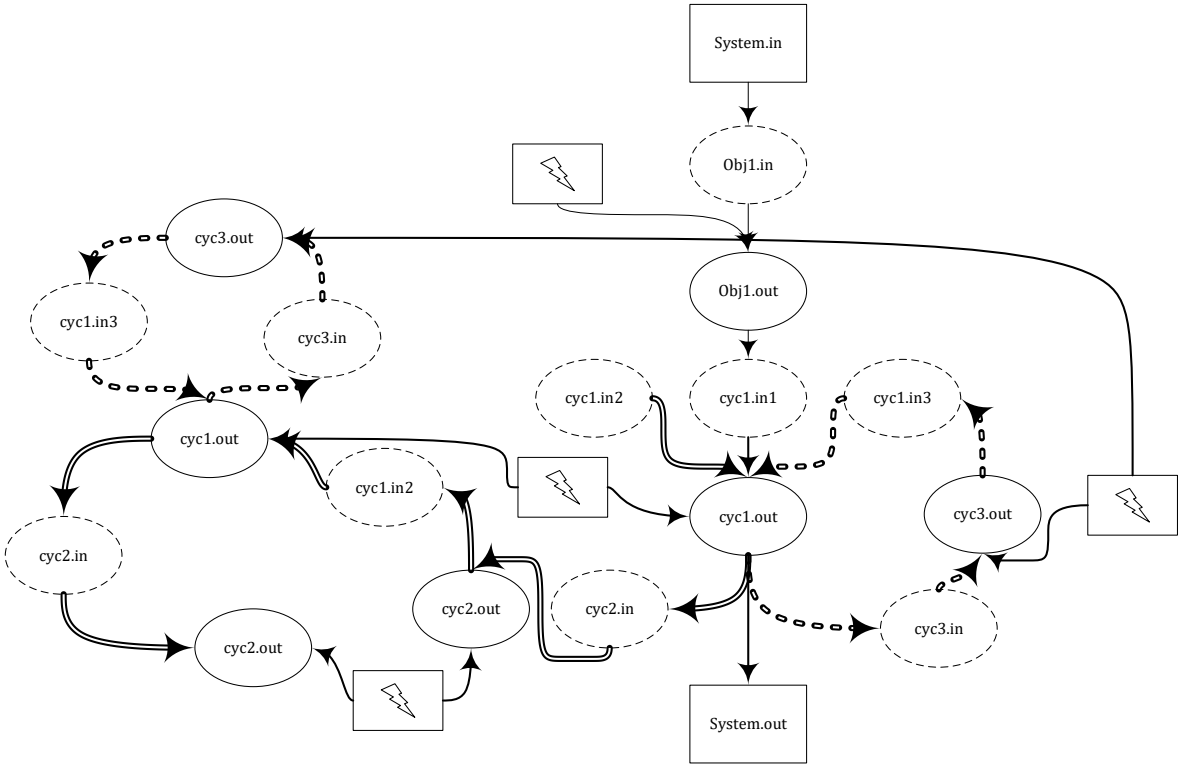


Figure 5.11: If we cut the wrong edge of a cycle, we might duplicate further cycles contained in the same Strongly Connected Component. Here the cycle with dashed edges is copied because we chose the wrong edge (between *cyc*2.*out* and *cyc*1.*in*2).

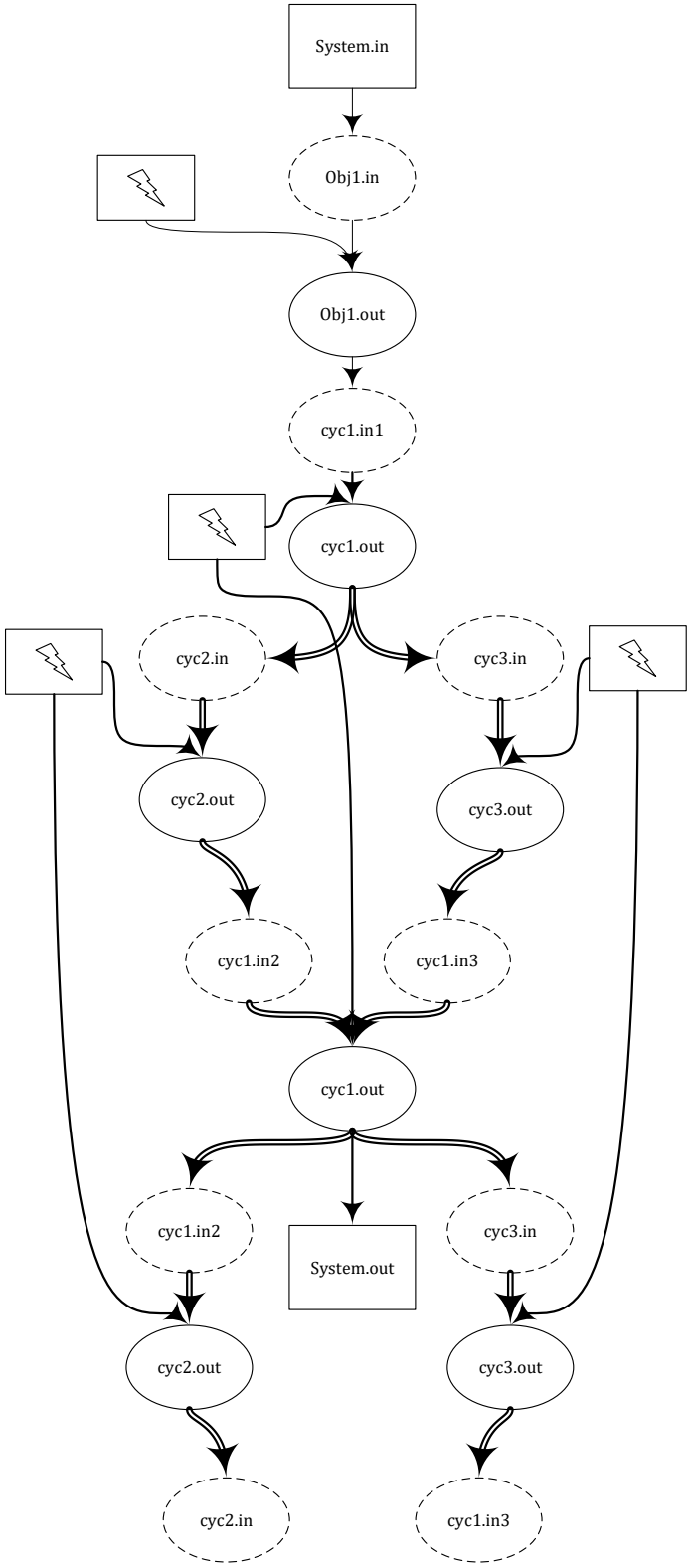


Figure 5.12: The edges in towards the node *cyc*1.*out* are the better choice, because all cycles can be solved in one step. We can determine this node by looking at the number of edges inside the Strongly Connected Components ending in this node. The node with the most of these edges is the best choice.

Chapter 6

Calculating the Reliability of Nodes in the Dependency Graph

6.1 Introduction

Each output port in the Connected Components System has an assigned failure rate shape (FRS) and its corresponding parameters. These values define the probability that the function belonging to this port produces incorrect output after a given time. Once a component has generated incorrect information, this information is forwarded to the next component and will introduce new failures in their functions. In order to avoid such malfunctions causing a complete system failure, the system may contain redundant components. Their output is afterwards evaluated by additional components, where some of their inputs might be faulty, while the output of the gate stays correct.

The goal of this chapter is to provide an algorithm for determining how well these redundancy components can eliminate failures. The measurement used for this is the so called *Reliability* of the system's components.

In this work we limit ourselves to components, which are neither repairable nor exchangeable, so that their functions will never recover. Additionally, we assume that components other than the redundancy components will never let failures disappear.

Aside from the Reliability we also want to calculate the mean time to failure (MTTF) for components introducing failures. In this work we will specialise on two very common failure distributions, namely the *Exponential* and the *Weibull Distribution*.

6.1.1 Preliminaries

We are going to perform some calculations of probabilities later on. Therefore, it is necessary to introduce the used notations and rules.

Notation

We are going to calculate probabilities of events, which we denote as $P[E]$ and probabilities of random variables. The later are expressed in two ways. The probability $P(X)$ denotes the probability distribution of a random variable X. In contrast to this, we use the notation $P_X(x) = P[X = x]$ as the probability of random variable X taking the value x. For the set of all possible assignments to a random variable X, we are going to write $\mathcal{X}$. For *Random Vectors*, we will use the notation $\vec{X}$.

Probability Distribution

The *Probability Distribution* of a random variable X is a function $P_X(x) : \mathcal{X} \mapsto [0, 1]$ with $0 \leq P_X(x) \leq 1$ and $\sum_{x \in \mathcal{X}} P_X(x) = 1$. [Stirzaker, 1999, page 194]

Joint Probability Distribution

For two given random variables X and Y, we calculate the *Joint Probability Distribution* as $P_{X,Y}(x, y) = P[X = x \cap Y = y]$.

All *Joint Probability Distribution* must fulfill the properties of probability distributions [Stirzaker, 1999, page 239]:

$$0 \leq P_{X,Y}(x, y) \leq 1$$

$$\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} P_{X,Y}(x, y) = 1$$

The Probability of a *Random Vector* $P_{\vec{X}}(\vec{x})$ is defined as the *Joint Probability* of all Random Variables inside the vector.

$$P_{\vec{X}}(\vec{x}) = P_{X_1,...X_n}(x_1, ..., x_n)$$

Conditional Probability

A so called *Conditional Probability* $P_{X|Y}(x|y) = P[X = x | Y = y]$ defines the probability, that a random variable X takes the value x, if the random variable Y is known to be y. We can compute it using the formula:

$$P_{X|Y}(x|y) = \begin{cases} \frac{P_{X,Y}(x,y)}{P_Y(y)} & \text{if } P_Y(y) > 0 \\ 0 & \text{else} \end{cases} \tag{6.1}$$

Stirzaker [1999, page 282] proofs that $P(X|Y)$ is also a probability distribution with $0 \leq P_{X|Y}(x|y) \leq 1$ and $\sum_{x \in \mathcal{X}} P_{X|Y}(x|y) = 1$.

Chain Rule

The *Chain Rule* provides a way to split a joint probability into a product of two or more probability distributions. The *Chain Rule* for two random variables is:

$$P(X, Y) = P(X) \cdot P(Y|X) = P(Y) \cdot P(X|Y) \tag{6.2}$$

For an arbitrary number of random variables $X_1, ..., X_n$ it is defined as:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i | X_{i-1}, ..., X_1) \tag{6.3}$$

Independence

We say two events A and B are *independent*, if $P[A \cap B] = P[A] \cdot P[B]$. Equally, two random variables X and Y are *independent*, if and only if the joint probability distribution of them can be expressed as $P(X, Y) = P(X) \cdot P(Y)$.

For multiple independent random variables $X_1, ...X_n$ the joint probability distribution can be expressed as the product of probability distributions:

$$P(X_1, ...X_n) = \prod_{i=1}^{n} P(X_i)$$

Conditional Independence

Two events A and B are called *conditional independent* given a vector of events $\vec{C}$, if $P[A, B \mid \vec{C}] = P[A \mid \vec{C}] \cdot P[B \mid \vec{C}]$. [Stirzaker, 1999, page 87]. Consequently, two random variables X and Y are also conditional independent given a vector of random variables $\vec{Z}$, if $P(X, Y \mid \vec{Z}) = P(X \mid \vec{Z}) \cdot P(Y \mid \vec{Z})$.

A resulting property of two *conditional independent* random variables X, Y, given a random variable vector $\vec{Z}$ is that $P(X \mid Y, \vec{Z}) = P(X \mid \vec{Z})$.

Marginal Probability Distribution

Given a joint probability distribution $P(X, Y)$, we can calculate the marginal probability $P_X(x)$ as in Equation 6.4.

$$P_X(x) = \sum_y P_{X,Y}(x, y) \tag{6.4}$$

Rule of Total Probability

From the rule for the marginal probability distribution combined with the chain rule, we can derive a new rule:

$$P_X(x) = \sum_y P_Y(y) \cdot P_{X \mid Y}(x \mid y) \tag{6.5}$$

Rohatgi and Saleh [2011] call this rule the *Rule of Total Probability*.

6.1.2 Definition of Failure Events

In his book, Stirzaker [1999, page 33] defines a so called *Sample Space* Ω.

Definition 6.1.1. A *Sample Space* Ω of a probabilistic experiment is the set of all possible outcomes.

For our purpose, Ω contains all possible states of each node $s_N \in \Omega_0 = \{\texttt{correct(=False)}, \texttt{incorrect(=True)}\}$ An outcome is the state of all nodes in the system. We abbreviate `incorrect` as i, and `correct` as c. An `incorrect` state of a node signals that there is a fault in the associated information, while the state `correct` indicates that the information at the node is valid.

$$\Omega = \underbrace{\Omega_0 \times ... \times \Omega_0}_{|\mathcal{N}| \text{ times}} = \Omega_0^{|\mathcal{N}|}$$

Example 6.1.1: Sample Space Ω of the System in Example 4.7.1

In the case of our Basic System Example 4.7.1, we have a Dependency Graph with $|\mathcal{N}|= 11$ nodes, as shown in Figure 4.3. Consequently, our *Sample Space* consists of $|\Omega_0|^{11}$ possible system states, all combinations for states for the 11 variables.

$$\begin{aligned}\Omega &= \{(ccccccccccc), (cccccccccci)..., (iiiiiiiiiii)\} \\ &= \{(\omega_1, \omega_2, ..., \omega_{11}) : \omega_i \in \Omega_0, i = 1..11\} \\ &= \Omega_0 \times \Omega_0 \times ... \times \Omega_0 = \Omega_0^{11}\end{aligned} \tag{6.6}$$

Example 6.1.2: Failure Event in the System in Example 4.7.1

For example the Failure Event E_k with the constraint *component.out_k introduces Failures* would contain all possible combinations of Ω_0 with the state of the data at node $k = component.out_k.FAULIURE_NODE$ fixed to `incorrect`.

$$\begin{aligned}E_1 &= \{(icccccccccc), (iicccccccccc), (icicccccccc)...\} \\ &= \{(\omega_1, \omega_2, ..., \omega_{11}) : \omega_l \in \Omega_0, l = 1..11 \wedge omega_k = 1\}\end{aligned} \tag{6.7}$$

Definition 6.1.2. An *Event* is a subset of Ω, for which a specific constraint holds.

Definition 6.1.3. A *Failure Event* of a node N is an *Event* $E_N \subset \Omega$ with the constraint, that the state of N is `incorrect`.

The *Union* of A and B is the set $(A \cup B)$, where all elements are part of the set A or part of the set B. Contrarily, the *Intersection* $(A \cap B)$ contains all elements, which are part of both sets.

Example 6.1.3: Union and Intersection of Failure Events

Considering the Failure Events E_1 and E_3 the union and intersection are:

$$\begin{aligned}E_1 &= \{(\omega_1, \omega_2, ..., \omega_{11}) : \omega_k \in \Omega_0, k = 1..11 \wedge \omega_i = 1\} \\ E_3 &= \{(\omega_1, \omega_2, ..., \omega_{11}) : \omega_k \in \Omega_0, k = 1..11 \wedge \omega_3 = i\} \\ E_1 \cup E_3 &= \{(icccccccccc), (ccicccccccc), (icicccccccc)...\} \\ &= \{(x_1, x_2, ..., x_{11}) : \omega_k \in \Omega_0, k = 1..11 \wedge (\omega_1 = i \vee \omega_3 = i)\} \\ E_1 \cap E_3 &= \{(icicccccccc), (iiicccccccc), (iciicccccccc)...\} \\ &= \{(x_1, x_2, ..., x_{11}) : \omega_k \in \Omega_0, k = 1..11 \wedge \omega_1 = i \wedge \omega_3 = i\}\end{aligned} \tag{6.8}$$

6.1.3 Calculating Probabilities of Independent Failure Events

The probability of a Failure Event E_N of node N is $P[E_N]$. We know how failures propagate through our System, so we can calculate the probability of a Failure Event using the probability of its parent Failure Events.

For an *AND Component*, the Failure Event of the *AND Proprocessing Node* will be active, if all parents' Failure Events are active. The probability for the AND Failure Event with n independent parent

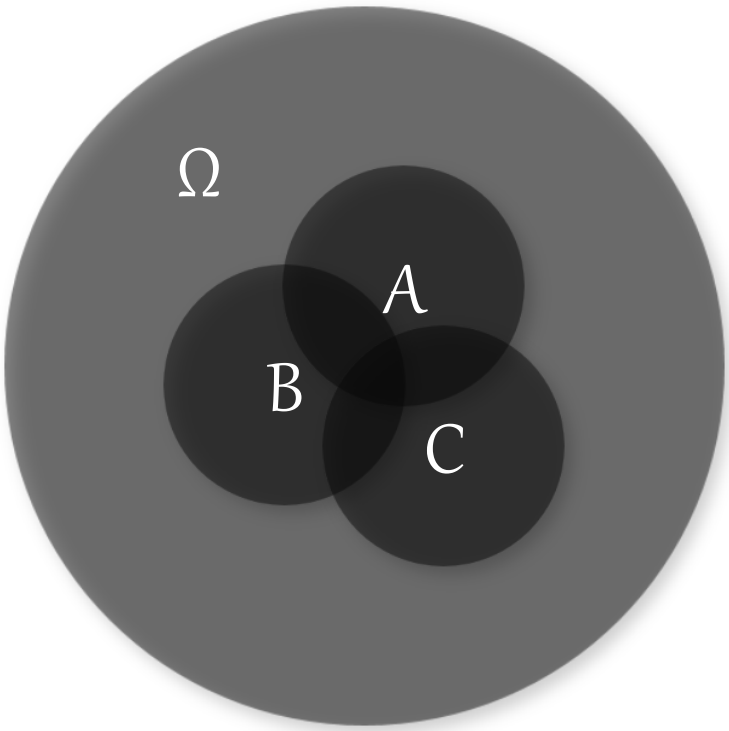


Figure 6.1: Sample space Ω with its events A, B and C. The events are sets of possible outcomes and can occur all at the same time, meaning that their corresponding ports all propagate incorrect information. Therefore the events are overlapping.

Failure Events $E_i,\ i = 1...n$ is calculated by:

$$P[AND] = P\left[\bigcap_{i=1}^{n} E_i\right] = \prod_{i=1}^{n} P[E_i] \tag{6.9}$$

We calculate the probability of a *VOTER Failure Event* of Node N by summing up all possibilities, which might make the Voter fail. In the case of a n-Voter, its Event will be inactive, if at least n parents are also inactive, where n is the natural number vote count with $n \in \{0, ..., |Parents(N)|\}$. Therefore, at least $|Parents(N)|+1 - votec(N)$ parent events must be active, such that the Voter's Failure Event also becomes active.

$$P[vote_2(A, B, C)] = P[A \cap B] + P[A \cap C] + P[B \cap C] - 2 \cdot P[A \cap B \cap C] \tag{6.10}$$

Recalling that Failure Events are a set of outcomes, two Failure Events can be active at the same time. The current state of the System is contained in both sets. Consequently the Failure Events do overlap. But the part $A \cap B \cap C$ is included in $A \cap B$, as well as in $A \cap C$ and $B \cap C$, so we would add it three times, and therefore have to subtract it two times. Figure 6.1 shows the Venn-Diagram of the three outcomes A, B and C.

All combinations of events at nodes, which are not preprocessing nodes, are combined via the operator OR. For an *OR Failure Event* to occur, only one of the parent Failure Events needs to be active. Therefore we compute the union of the parent Failure Events.

$$P[A \cup B] = P[A] + P[B] - P[A \cap B] \tag{6.11}$$

Again, we need to subtract the additional part, because $A \cap B$ is part of both A and B and we have added it two times.

6.1.4 Calculating Failure Probabilities with Random Variables

A different way to express a probability is to use *Random Variables* instead of Events. *Random Variables* stand for the state of one node. Therefore they are more suited for our case, than using events.

Durrett [2010, chapter 1, section 3] defines Random Variables as:

Definition 6.1.4. A *Random Variable* is a function $X : \Omega \mapsto \mathbb{R}$, that maps a value of Ω to a real number.

Definition 6.1.5. A *discrete Random Variable* maps every value $\omega \in \Omega$ to a value $\omega' \in \mathcal{B}$, where $\mathcal{B}$ is finite. In the case of a *binary Random Variable*, $\mathcal{B} = \{0, 1\}$.

Definition 6.1.6. The *Indicator Function* for an event E is a binary random variable $\mathbb{1}_E(\omega) : \Omega \mapsto \{1, 0\}$, such that

$$\mathbb{1}_E(\omega) = \begin{cases} 1 & \text{if} \quad \omega \in E \\ 0 & \text{else} \end{cases}$$

In our case, we have a binary random variable for each node $i \in N$ with the Failure Event E_i, meaning $|N|$ binary random variables $X_i(\omega) : \Omega \mapsto \{0, 1\}$, with $X_i(\omega) = \mathbb{1}_{E_i}(\omega)$ for $i \in \{1, .., |N|\}$.

Consequently, the random variable X_n of a node $n \in N$ therefore takes the value 0, if n sends correct information to its children, and 1 otherwise. The probability $P[E_n]$ for an Failure Event of node n is equivalent to $P_{X_n}(1)$. For simplification, we will give the random variables the same name as the corresponding nodes and write shortly $P_n(1)$.

Calculating the probability $P_{A,B}(1, 1)$ is the same as calculating the probability of $P[E_A \cap E_B]$, where E_A and E_B are the Failure Events corresponding to the nodes A and B.

We can now transform the previous probability rule for AND Events to Equation 6.12.

$$\begin{aligned} P[E_A \cap E_B] &= P_{A,B}(1, 1) \\ &= P_A(1) \cdot P_B(1) \end{aligned} \tag{6.12}$$

$$\begin{aligned} P[vote_2(E_A, E_B, E_C)] &= P_{A,B,C}(0, 1, 1) + P_{A,B,C}(1, 0, 1) + P_{A,B,B}(0, 1, 1) + P_{A,B,C}(1, 1, 1) \\ &= \sum_{a \in \{1,0\}} P_{A,B,C}(a, 1, 1) - P_{A,B,C}(1, 1, 1) \\ &+ \sum_{b \in \{1,0\}} P_{A,B,C}(1, b, 1) - P_{A,B,C}(1, 1, 1) \\ &+ \sum_{c \in \{1,0\}} P_{A,B,C}(1, 1, c) - P_{A,B,C}(1, 1, 1) + P_{A,B,C}(1, 1, 1) \end{aligned} \tag{6.13}$$

Applying the marginal probability rule we can calculate the following result:

$$P[vote_2(E_A, E_B, E_C)] = P_{A,B}(1, 1) + P_{A,C}(1, 1) + P_{B,C}(1, 1) - 2P_{A,B,C}(1, 1, 1) \tag{6.14}$$

6.1.5 Introducing Time

The previously described formulae can be used to derive the failure probability of a logical gate as combination of two or more failure probabilities. Since the failure probability of one component does not always stay the same, we have to introduce time into our formulae. Kapur and Pecht [2014] define a value $R(t)$ as the probability that a component is operating at time t, meaning its *reliability*. This value is

equal to $1 - F(t)$ namely the *failure probability* at time t. Furthermore they introduce a function $h(t)$, the *hazard* or *failure rate*.

The *reliability* is calculated as shown in Equation 6.15 and the failure probability is shown in Equation 6.16.

$$R(t) = \exp\left(-\int_0^t h(\tau)\ d\tau\right) \tag{6.15}$$

$$F(t) = 1 - R(t) = 1 - \exp\left(-\int_0^t h(\tau)\ d\tau\right) \tag{6.16}$$

Combined with the random variable for success or failure, this leads to $P_C(0|t) = R_C(t)$, telling that component c works fine at time t. On the contrary we have $P_C(1|t) = F_C(t)$ telling that our component c is faulty after time t.

The MTTF is calculated as the expected value of the failure density function $f(t) = -\frac{\partial R(t)}{\partial t}$. We can therefore calculate it by:

$$MTTF = \int_0^\infty t \cdot f(t) \quad dt \tag{6.17}$$

In case the MTTF is finite it can be calculated by the formula shown in Equation 6.18. [Høyland and Rausand, 2004]

$$MTTF = \int_0^\infty R(t) \quad dt \tag{6.18}$$

6.1.6 Failure Rate Distributions

For our approach, we are going to limit the possible *failure rates* to the most common shapes and take provided parameters to calculate the full function.

6.1.6.1 Constant Failure Rate and Exponential Distribution

The simplest failure rate would be a constant value, independent of time.

$$h(t) = r \qquad \text{with} \quad r > 0, t > 0 \tag{6.19}$$

Høyland and Rausand [2004, page 26] describe this as Exponential Distribution because of its exponential density function $f(t)$.

$$f(t) = \frac{\partial R(t)}{\partial t} = \begin{cases} re^{-rt} & \text{if} \quad r > 0, t > 0 \\ 0 & \text{else} \end{cases} \tag{6.20}$$

We now get

$$R(t) = \exp\left(-\int_0^t h(\tau)\ d\tau\right) = \exp\left(-\int_0^t r\ d\tau\right) = \exp(-rt) \tag{6.21}$$

$$MTTF = \int_0^\infty \exp(-rt) \quad dt = \left[\frac{-1}{r}\exp(-rt)\right]_0^\infty = \frac{1}{r} \tag{6.22}$$

Figure 6.2 shows the reliability with a constant rate r in dependence of time for three different failure rates.

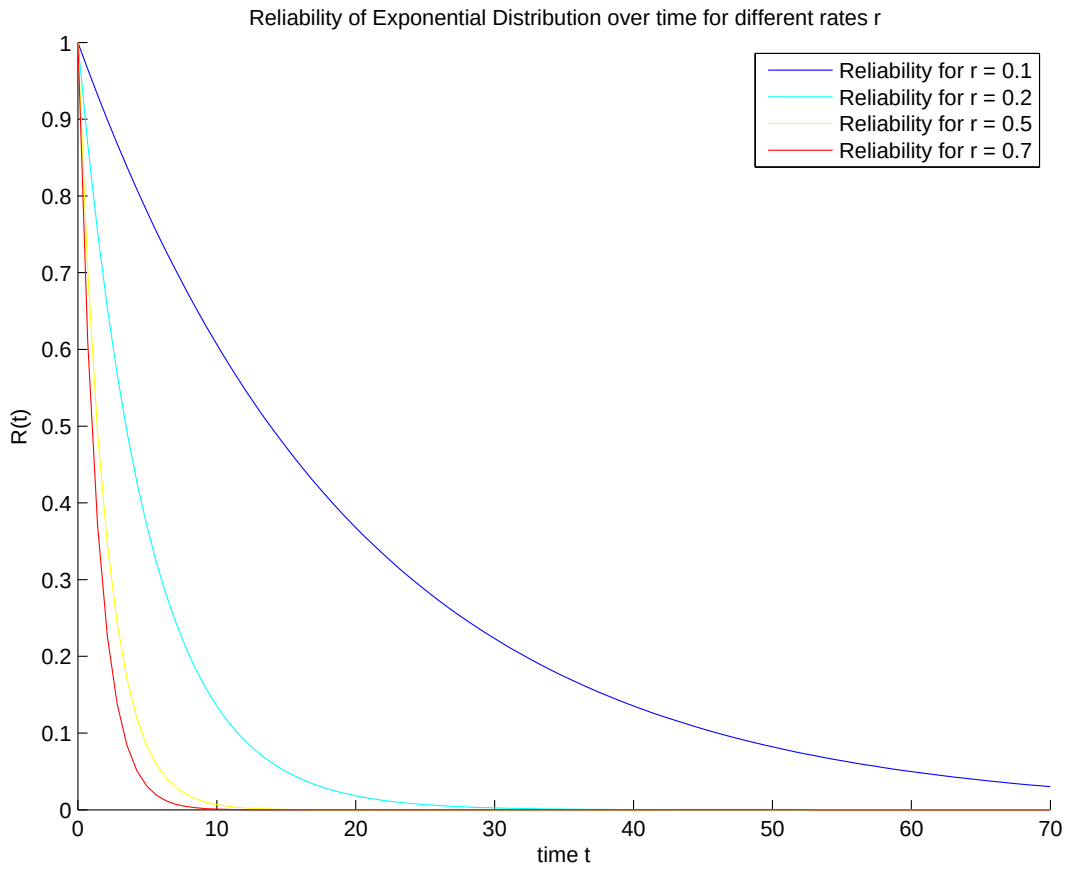


Figure 6.2: Reliability of exponential distribution in dependence of time given constant rate r

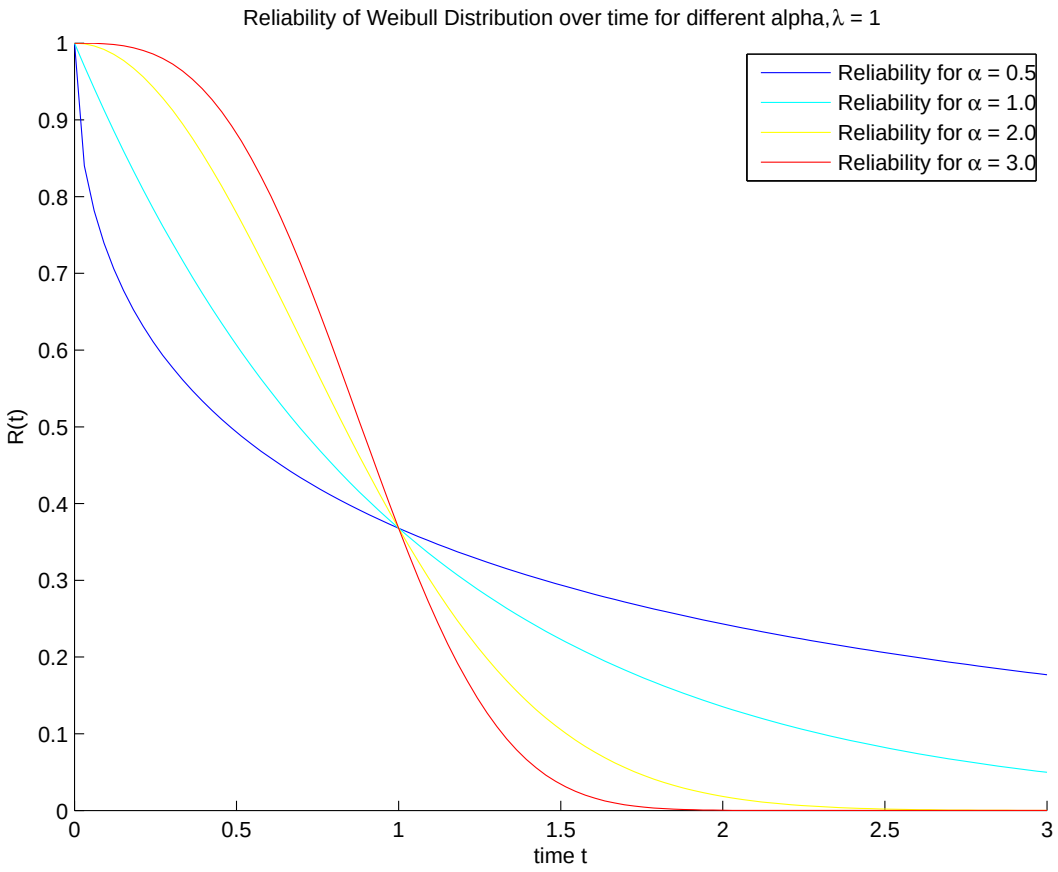


Figure 6.3: Reliability of Weibull distribution in dependence of time given the parameters α and λ

6.1.6.2 Weibull Distribution

Another failure distribution that we want to consider is the *Weibull* distribution. Developed by Prof. Waloddi Weibull, it is a very flexible distribution. Høyland and Rausand, 2004

Equation 6.23 shows the Reliability $R(t)$, Equation 6.24 the failure rate for the Weibull distribution $Weibull(\alpha, \lambda)$, where $\alpha > 0$ and $\lambda > 0$ are parameters for the distribution.

$$R(t) = e^{-(\lambda t)^{\alpha}} \tag{6.23}$$

$$h(t) = \alpha \lambda^{\alpha} t^{(\alpha-1)} \tag{6.24}$$

$$MTTF = \frac{1}{\lambda}\Gamma\left(\frac{1}{\alpha} + 1\right) \tag{6.25}$$

6.1.6.3 Further Distributions

Further distributions, that have not been discussed here would be for example the binomial and geometric distribute Høyland and Rausand, 2004, page 25, which has not been considered since it is a discrete distribution in dependence of trials instead of time.

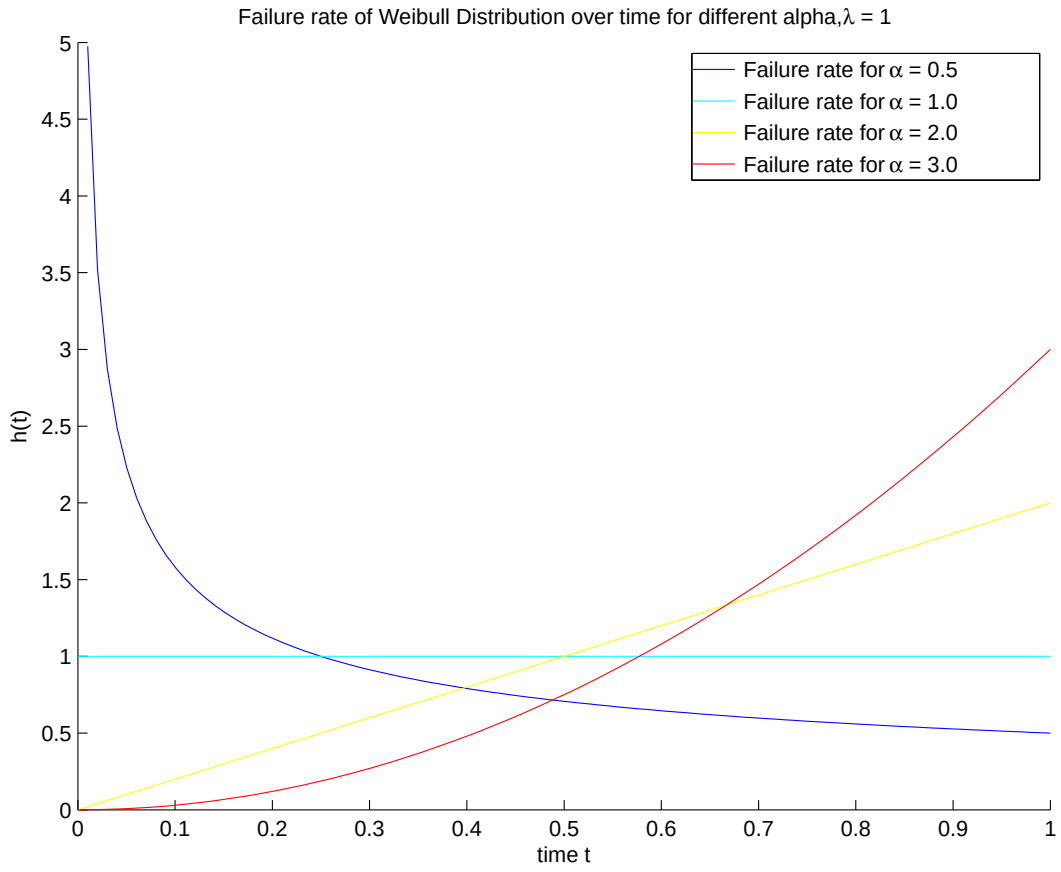


Figure 6.4: Failure rate of Weibull in dependence of time given the parameters α and λ

6.1.7 Dealing with Conditional Dependence between Random Variables

So far we only considered a straight forward tree architecture as Dependency Graph. In such a graph all parent events are independent. Unfortunately in our models we will not always have this property. Instead the models support Common Cause Failures. Those Failures are indicated by an acyclic loop in the graph.

We recall the definition of a loop from Definition 5.3.2:

Definition 6.1.7. A Loop in a graph is a undirected path, which starts and ends in the same node.

Since we already removed all Cycles (see Section 5), only acyclic loops are left. There exists at least one edge which is directed against the other edges' directions. An acyclic loop may contain two paths from one node to another one. The Dependency Graph has no tree-like structure and two parent Failure Events are not guaranteed to be independent.

6.1.8 Dominators and Shared Predecessors

In his book, Appel [1997, section 18.1], defines so called *Dominators*.

Definition 6.1.8. A *Dominator* $d \in D(n)$ of node n is a node, which is included in every directed path from any system input node of a graph towards the node n. The node n is also a Dominator of itself.

Furthermore, he introduced the function $idom(n)$, which returns the *immediate Dominator* for a node m in a graph with only one system input s_0.

The following properties of $idom(n)$ hold $\forall n \neq s_0$:

- idom(n) is a Dominator of n.
- idom(n) is not n itself.
- idom(n) dominates no other Dominator of n but n itself.

Derived from $idom(n)$, we define a new function $shPred(n_1, n_2)$, which returns all nodes, which are *predecessors* of both n_1 and n_2.

$$shPred(n_1, n_2) = (Pred(n_1) \cup n_1) \cap (Pred(n_2) \cup n_2)$$

$idom(n) \in shPred(\vec{Parent}(n))$

If two event nodes have a *shared predecessor*, their events are not independent. We cannot use the product of their probabilities to calculate their intersection.

Last, we define a function $clBPred(n)$, which returns the *closest branching predecessors*. The following conditions must hold for any $clBPred(n)$:

- $clBPred(n)$ is a *shared predecessor* of any two nodes.
- There exists at least one path between n and any $c_1 \in clBPred(n)$, which contains no node $c_2 \in clBPred(n)$ with $c_2 \neq c_1$.

Consequently at any node $c \in clBPred(n)$, to paths from c towards n must branch off.

Example 6.1.4: Dependency Graph containing a Loop

An example for a Dependency Graph containing acyclic loops is shown in Figure 6.5. This will be used later in the example for calculating the Reliability for a given Dependency Graph.
The Dominators (without considering Failure Nodes) are defined as:

$$\begin{aligned} idom(a.out) &= System.in \\ idom(b.out) &= a.out \\ idom(c.out) &= a.out \\ idom(d.out) &= c.out \\ idom(e.out) &= c.out \\ idom(f) &= a.out \\ idom(g) &= a.out \end{aligned} \tag{6.26}$$

Shared predecessors (without considering Failure Nodes) of parents of node f are:

$$\begin{aligned} shAnc(d.out, e.out) &= \{c.out, a.out, System.in\} \\ shAnc(e.out, b.out) &= \{a.out, System.in\} \\ shAnc(d.out, b.out) &= \{a.out, System.in\} \end{aligned} \tag{6.27}$$

Likewise, those of node g are:

$$shAnc(f.out, b.out) = \{b.out, a.out, System.in\} \tag{6.28}$$

The union of all shared Predecessors is

$$\{a.out, c.out, System.in, Failure(a.out), Failure(c.out), b.out, Failure(b.out)\}$$

The closest branching predecessors are:

$$\begin{aligned} clBPred(a.out) &= \{\} \\ clBPred(b.out) &= \{a.out\} \\ clBPred(c.out) &= \{a.out\} \\ clBPred(d.out) &= \{c.out\} \\ clBPred(e.out) &= \{c.out\} \\ clBPred(f.out) &= \{c.out, b.out\} \\ clBPred(g.out) &= \{b.out, c.out\} \\ clBPred(f) &= \{c.out, b.out\} \\ clBPred(g) &= \{b.out, c.out\} \end{aligned} \tag{6.29}$$

6.1.8.1 Rules of d-Separation

Since the previous defined formulae to calculate the probabilities at gates only apply to independent events, we have to identify dependence between our random variables. The dependencies of our random variables are represented in our Dependency Graph, we can evaluate this graph to find all dependencies between the variables. In order to do so, we use the so-called rules of d-Separation. [Langseth and Portinale, 2007]

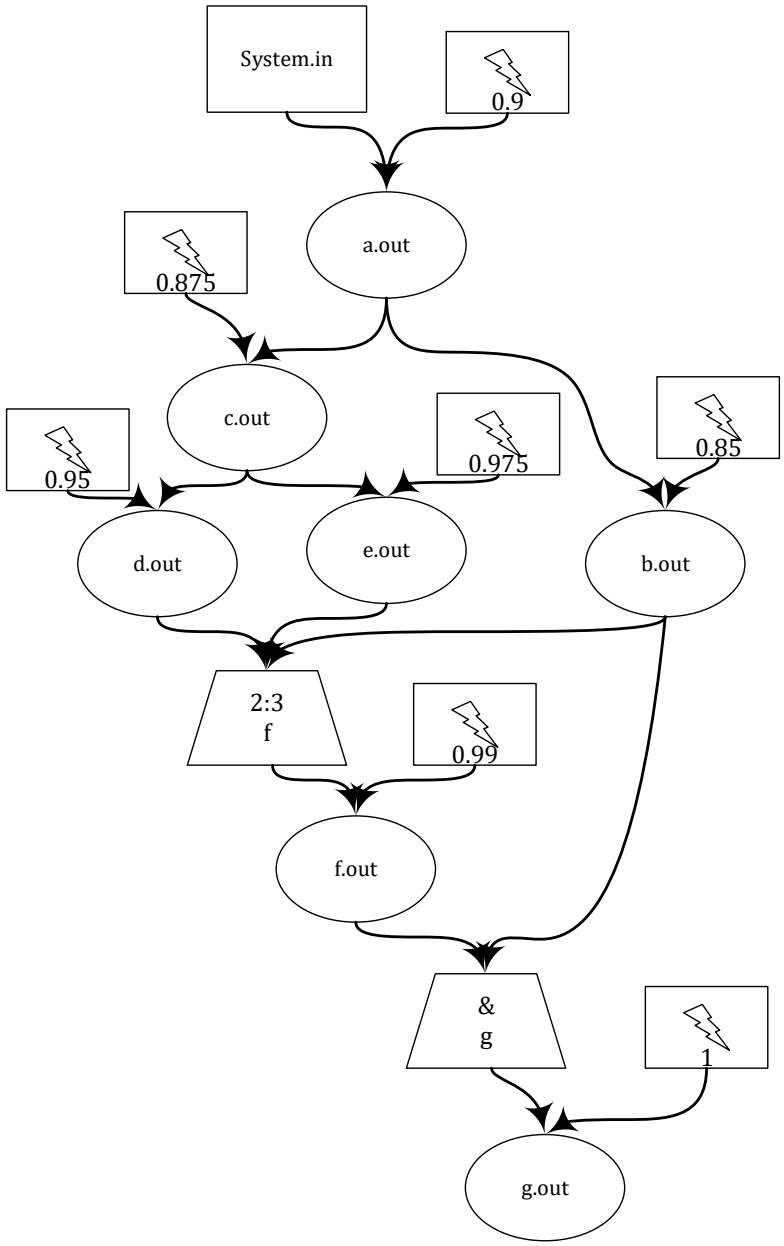


Figure 6.5: This Dependency Graph will be used as an example for calculating the Reliability of nodes in a Dependency Graph. The It contains some acyclic loops, where the parents are not independent.

Definition 6.1.9. Two random variables A and B are *independent*, if $P(X, Y) = P(X) \cdot P(Y)$. We write $X \perp Y$.

Definition 6.1.10. Two random variables X and Y are *conditional independent* given a set of random variables $\vec{Z}$, such that $P(X, Y \mid \vec{Z}) = P(X \mid \vec{Z}) \cdot P(Y \mid \vec{Z})$. We write $X \perp Y \mid \vec{Z}$.

Koski and Noble [2009b] define two random variables of a Bayesian Network to be *conditional independent*, if there exists no active undirected path between their corresponding nodes.

Definition 6.1.11. We call a path *active*, if there exists no node in the path, which is blocking the path.

Definition 6.1.12 (d-Separation Rule 1)**.** A path is blocked by a set of nodes Z, if two directed edges on the path meet tail-to-tail to a node $z \in Z$.

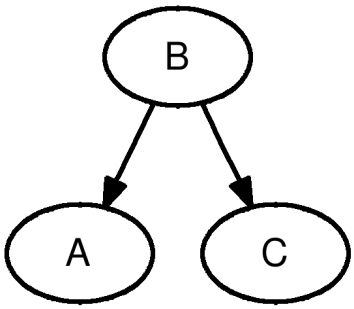


Figure 6.6: Rule 1: The path between node A and C is blocked, if we know the value of variable B.
$A \perp C \mid B$

Definition 6.1.13 (d-Separation Rule 2)**.** A path is blocked by a set of nodes Z, if two directed edges on the path meet head-to-tail to a node $z \in Z$.

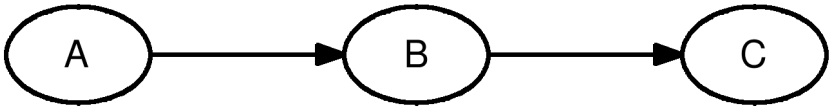


Figure 6.7: Rule 2: The path between node A and C is blocked, if we know the value of variable B.
$A \perp C \mid B$

Definition 6.1.14 (d-Separation Rule 3)**.** A path is blocked by a set of nodes Z, if two directed edges on the path meet head-to-head to a node x and neither x nor its descendants are included in Z.

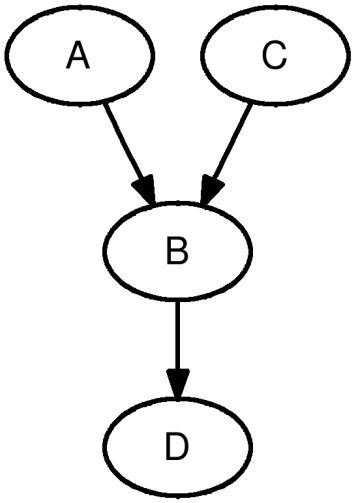


Figure 6.8: Rule 3: The path between node A and C is active, if B or its descendant D is known.
$A \not\perp C \mid B$,
$A \not\perp C \mid D$,
$A \perp C \mid \emptyset$

We can conclude, that any two parents of a node n are conditional independent given $clBPred(n)$.

6.1.8.2 Calculation of Failure Probability with Dependent Random Variables

If two parents of a node share the same predecessor, we cannot compute the Joint Probability using $P(X, Y) = P(X) \cdot P(Y)$.

Example 6.1.5: d-Separation rules applied to Example 6.1.4

For Example 6.1.4 we can derive:

$$\begin{aligned} &d.out \not\perp e.out \mid \emptyset \\ &d.out \perp e.out \mid \{c.out\} \\ &d.out \not\perp e.out \mid \{c.out, f\} \end{aligned} \tag{6.30}$$

$$d.out \perp e.out \perp b.out \mid \{c.out, a.out\}$$

This formula only holds if X and Y are independent random variables ($X \perp Y$). Instead, we have to use the *Chain Rule*:

$$P(X, Y) = P(X) \cdot P(Y|X) = P(Y) \cdot P(X|Y) \tag{6.31}$$

Let $\vec{A}$ be all *Shared Predecessors* of X and Y and $\mathcal{A}$ all possible state combinations of the nodes in A, the Joint Probability of X and Y is:

$$\begin{aligned} P_{X,Y}(x, y) &= \sum_{\vec{a} \in \mathcal{A}} P_{X,Y \mid \vec{A}}(x, y \mid \vec{a}) P_{\vec{A}}(\vec{a}) \\ &= \sum_{\vec{a} \in \mathcal{A}} P_{X \mid \vec{A}}(x \mid \vec{a}) P_{Y \mid \vec{A}}(y \mid \vec{a}) P_{\vec{A}}(\vec{a}) \end{aligned} \tag{6.32}$$

6.1.9 Calculating the Reliabilities in Complex Dependency Graphs

The previous sections defined how to calculate the reliability for simple Dependency Graphs without common cause failures. This section will introduce formulae for calculating the reliability for more complex systems. For every node, we express the Reliability as $R_N(t) = P_{N \mid I\vec{N},T}(0 \mid \vec{0}, t)$, where N is the Node's random variable, for which we want to calculate the reliability and $I\vec{N}$, a random vector containing all system input random variables. The system's input is always assumed to be correct, therefore it is given as a vector of zeros, $\vec{0}$. The reliability is the probability, that there is no error in the component associated with the random variable. Therefore we want the random variable to also take the value 0. T is the Random Variable providing the current time.

6.1.9.1 Reliability at a Failure Node

We can calculate the Reliability at a Failure Node at time t from the FRS and the the provided parameters. It is independent of any other event.

$$P_{C \mid I\vec{N},T}(0 \mid \vec{0}, t) = R_C(t) \tag{6.33}$$

6.1.9.2 Reliabilities at a Forward Node

Let N be a Forward Node's random variable. According to the definition of Forward Nodes, they have exactly one parent $Pa(N)$. All Forward Nodes propagate errors through the model with a probability of 1 and the probability of an fault in Forward Node N is $P_{N \mid Pa(N),t}(1 \mid 1, t) = 1$. Equally, the probability that an Forward Node introduces new faults $P_{N \mid Pa(N),t}(1 \mid 0, t)$ is 0. We can derive $P_{N \mid Pa(N),t}(0 \mid 1, t) = 0$ and $P_{N \mid Pa(N),t}(0 \mid 0, t) = 1$.

$$P_{N \mid I\vec{N},T}(0 \mid \vec{0}, t) = P_{Pa(N) \mid I\vec{N},T}(0 \mid \vec{0}, t)$$

Proof:

Using the rule of total probability we can derive:

$$\begin{aligned} P_{N \mid I\vec{N},T}(0 \mid \vec{0}, t) &= \sum_{p \in \{0,1\}} P_{N \mid Pa(N)}(0 \mid p) P_{Pa(N) \mid I\vec{N},T}(p \mid \vec{0}, t) \\ &= P_{N \mid Pa(N)}(0 \mid 0) P_{Pa(N) \mid I\vec{N},T}(0 \mid \vec{0}, t) + P_{N \mid Pa(N)}(0 \mid 1) P_{Pa(N) \mid I\vec{N},T}(1 \mid \vec{0}, t) \\ &= 1 \cdot P_{Pa(N) \mid I\vec{N},T}(0 \mid \vec{0}, t) + 0 \cdot P_{Pa(N) \mid I\vec{N},T}(1 \mid \vec{0}, t) \\ &= P_{Pa(N) \mid I\vec{N},T}(0 \mid \vec{0}, t) \end{aligned} \tag{6.34}$$

6.1.9.3 Reliability at a Node with two or more Parents

As long as the parents of a node are independent, we can use the rules for independent random variables to calculate the reliability of a node.

If a loop exists, this means, that there are at least two parents of the node, for which a *Shared Predecessor* exists. Equally, also the set of *closest branching predecessors* is not empty.

Let N be such a node's random variable, $\vec{Pa}(N)$ a vector of its parents' random variables and $\mathcal{P}$ all combinations of their potential states. Then Equation 6.35 expresses the Reliability of R.

$$P_{N \mid I\vec{N},T}(0 \mid \vec{0}, t) = \sum_{\vec{p} \in \mathcal{P}(N)} P_{N \mid \vec{Pa}(N),T}(0 \mid \vec{p}, t) P_{\vec{Pa}(N) \mid I\vec{N},T}(\vec{p} \mid \vec{0}, t) \tag{6.35}$$

In the most cases not all random variables in $\vec{Pa}$ will be independent. Therefore $P_{\vec{Pa}}(\vec{p}) \neq \prod_{Pa_i \in \vec{p}} P_{Pa_i}(p_i)$. Instead, we need to apply the chain rule $P_{\vec{Pa}}(\vec{p}) = \prod_{Pa_i \in \vec{Pa}} P_{Pa_i \mid Pa_1,...,Pa_{i-1}}(p_i \mid p_1, ..., p_{i-1})$.

Let $\vec{C}$ be all Nodes' Random Variables in $clBPred(N)$ and C all their possible assignments.

$$P_{\vec{Pa} \mid I\vec{N},T}(\vec{p} \mid \vec{0}, t) = \sum_{\vec{c} \in C} \prod_{p_i \in p} P_{Pa_i \mid \vec{C},T}(p_i \mid \vec{c}, t) P_{\vec{C} \mid I\vec{N},T}(\vec{c} \mid \vec{0}, t) \tag{6.36}$$

This formula can be applied recursively until the set $clBPred(N)$ is empty.

6.1.9.4 Reliability at an Output Node

An Output Node takes the logical OR of all its inputs.

$$P_{OUT \mid \vec{Pa}(OUT),T}(0 \mid \vec{p}, t) = \begin{cases} 1 & \text{if} \quad \vec{p} = \vec{0} \\ 0 & \text{else} \end{cases}$$

Consequently, we can express the reliability of the OR Event Node as:

$$P_{OUT \mid I\vec{N},T}(0 \mid \vec{0}, t) = P_{\vec{Pa}(OUT) \mid I\vec{N},T}(\vec{0} \mid \vec{0}, t)$$

6.1.9.5 Reliability at AN Preprocessing Nodes

If any ancestors' random variables is 0, an AND Preprocessing Node's random variable also takes the value 0.

$$P_{AND\,|\,\vec{Pa}(AND)}(0\,|\vec{p}) = \begin{cases} 1 & \text{if} \quad \vec{p} = \vec{1} \\ 0 & \text{else} \end{cases}$$

Where AND denotes a random variable of an AND-Node and $\vec{Pa}(AND)$ the node's ancestors.

Therefore, an AND-Gate is the same as a 1-VOTER.

6.1.9.6 Reliability at VOTER Preprocessing Nodes

Every Voter-Node has a tagged number $voteC$, which must be smaller than its number of parents $|\vec{Pa}(N)|$. C indicates, how many parent random variables must at least be set to 0, for the Voter random variable to be 0.

$$P_{VOTE\,|\,\vec{Pa}(VOTE)}(0\,|\,\vec{p}) = \begin{cases} 1 & \text{if} \quad \sum_{i=1}^{n} p_i <= n - voteC \\ 0 & \text{else} \end{cases}$$

Example 6.1.6: Reliabilities in Example 6.1.4

In Example 6.1.4 the parent nodes of the VOTER Preprocessing Node f - $d.out$, $e.out$ and $b.out$ - are dependent, since their pairwise shared predecessors are $c.out, a.out, System.in$ - a non empty set. It has two closest branching predecessors, $c.out$ and $a.out$. The random variables $d.out$, $e.out$ and $b.out$ are conditional independent given the state of these closest Branching Predecessors. We can use this to calculate the reliability of f:

$$\begin{aligned} P_F(0) &= \sum_{d,e,b\in\{0,1\}} P_{F\,|\,D,E,B}(0\,|\,d,e,b) P_{D,E,B}(d,e,b) \\ &= P_{D,E,B}(0,0,1) + P_{D,E,B}(1,0,0) + P_{D,E,B}(0,1,0) + P_{D,E,B}(0,0,0) \\ &= \sum_{d,e,b\in\{0,1\}} P_{F\,|\,D,E,B}(0\,|\,d,e,b) \left[\sum_{c,a\in\{0,1\}} P_{D\,|\,C,A}(d\,|\,c,a) P_{B\,|\,C,A}(b\,|\,c,a) P_{E\,|\,C,A}(e\,|\,c,a) \right] P_{C,A}(c,a) \end{aligned} \tag{6.37}$$

For the AND Preprocessing Node g, the parents are again not independent. Differently than before, the parent $b.out$ is included in the set $clBPred(g)$.

$$\begin{aligned} P_G(0) &= \sum_{b,f\in\{0,1\}} P_{G\,|\,B,F}(0\,|\,b,f) P_{B,F}(b,f) \\ &= P_{B,F}(0,1) + P_{B,F}(1,0) + P_{B,F}(0,0) \\ &= \sum_{b,a\in\{0,1\}} \left[P_{B\,|\,B,A}(0\,|\,b,a) P_{F\,|\,B,A}(1\,|\,b,a) + P_{B\,|\,B,A}(1\,|\,b,a) P_{F\,|\,B,A}(0\,|\,b,a) \right. \\ &\quad \left. + P_{B\,|\,B,A}(0\,|\,b,a) P_{F\,|\,B,A}(0\,|\,b,a) \right] P_{B,A}(b,a) \\ &= \sum_{a\in\{0,1\}} \left[P_{F\,|\,B,A}(1\,|\,0,a) P_{B|A}(0|a) + P_{F\,|\,B,A}(0\,|\,1,a) P_{B|A}(1|a) \right. \\ &\quad \left. + P_{F\,|\,B,A}(0\,|\,0,a) P_{B|A}(0|a) \right] P_A(a) \end{aligned} \tag{6.38}$$

6.1.10 Algorithm for Reliability Calculation

The trivial algorithm for Reliability Calcualtion would be to calculate the reliability for every node separately. Of course this is not very efficient. A different approach would be to iterate over all possible assignments for Failure Nodes. The number of Failure Nodes is equal the number of output ports in the Connected Components System, so it will always be greater or equal the number of components. The number of failure states is $2^{count(FailureNodes)}$. This lets the algorithm's run time increase exponentially with the number of output ports in the model.

A more efficient solution is the combination of the trivial algorithm with the knowledge of closest Branching Predecessors. The main idea is to first calculate the *Relative Relibaility*, the reliability of each node, given either all possible states of their *closest Branching Prdecessors* or their parents', if there is no *clBPred* for a node. Afterwards, we can calculate the *(Absolute) Reliability* by taking the product of the *Relative Reliabilities*.

$$P_{N\,|\,\vec{IN},T}(0\,|\,\vec{0},t) = \sum_{\vec{c}\in\rfloor\updownarrow\mathcal{BP}\nabla\rceil\lceil(\mathcal{N})} P_{N\,|\,\vec{clBPred}(N),T}(0\,|\,\vec{c},t)P_{\vec{clPred}(N)\,|\,\vec{IN},T}(\vec{c}\,|\,\vec{0},t) \tag{6.39}$$

The number of *closest Branching Predecessors* will always be smaller or at least equal the number of Failure Nodes, such that this algorithm will always have a shorter run time.

Example 6.1.7: Repetition of Closest Branching Predecessors in the Dependency Graph of Figure 6.5

$$\begin{aligned}
clBPred(a.out) &= \{\}\\
clBPred(b.out) &= \{a.out\}\\
clBPred(c.out) &= \{a.out\}\\
clBPred(d.out) &= \{c.out\}\\
clBPred(e.out) &= \{c.out\}\\
clBPred(f.out) &= \{c.out, b.out\}\\
clBPred(g.out) &= \{b.out, c.out\}\\
clBPred(f) &= \{c.out, b.out\}\\
clBPred(g) &= \{b.out, c.out\}
\end{aligned} \tag{6.40}$$

Example 6.1.8: Relative Reliability in the Dependency Graph of Figure 6.5 - I
We calculate the *Relative Reliability* of the nodes in Dependency Graph in Figure 6.5 using the *closest Branching Predecessors* as defined in Example 6.1.7.

$$
\begin{aligned}
RR(a.out \mid t) &= P_{a.out \mid System.in, Failure(a.out), T}(0 \mid 0, 0, t) \\
&= 0.9 \\
\\
RR(b.out \mid a.out = 0, t) &= P_{b.out \mid a.out, Failure(b.out), T}(0 \mid 0, 0, t) \\
&= 0.85 \\
RR(b.out \mid a.out = 1, t) &= P_{b.out \mid a.out, Failure(b.out), T}(0 \mid 1, 0, t) \\
&= 0 \\
\\
RR(c.out \mid a.out = 0, t) &= P_{c.out \mid a.out, Failure(c.out), T}(0 \mid 0, 0, t) \\
&= 0.875 \\
RR(c.out \mid a.out = 1, t) &= P_{c.out \mid a.out, Failure(c.out), T}(0 \mid 1, 0, t) \\
&= 0 \\
\\
RR(d.out \mid c.out = 0, t) &= P_{d.out \mid c.out, Failure(d.out), T}(0 \mid 0, 0, t) \\
&= 0.95 \\
RR(d.out \mid c.out = 1, t) &= P_{d.out \mid c.out, Failure(d.out), T}(0 \mid 1, 0, t) \\
&= 0 \\
\\
RR(e.out \mid c.out = 0, t) &= P_{e.out \mid c.out, Failure(e.out), T}(0 \mid 0, 0, t) \\
&= 0.975 \\
RR(e.out \mid c.out = 1, t) &= P_{e.out \mid c.out, Failure(e.out), T}(0 \mid 1, 0, t) \\
&= 0 \\
\\
RR(e.out \mid c.out = 0, t) &= P_{e.out \mid c.out, Failure(e.out), T}(0 \mid 0, 0, t) \\
&= 0.975 \\
RR(e.out \mid c.out = 1, t) &= P_{e.out \mid c.out, Failure(e.out), T}(0 \mid 1, 0, t) \\
&= 0 \\
\\
RR(f.out \mid c.out = c, b.out = b, t) &= P_{f.out \mid f, Failure(f.out), T}(0 \mid 0, 0, t) \\
&= 0.99 P_{f \mid c.out, b.out, T}(0 \mid c, b, t) \\
\\
RR(g.out \mid c.out = c, b.out = b, t) &= P_{g.out \mid g, Failure(g.out), T}(0 \mid 0, 0, t) \\
&= P_{g \mid c.out, b.out, T}(0 \mid c, b, t)
\end{aligned}
\tag{6.41}
$$

Example 6.1.9: Relative Reliability in the Dependency Graph of Figure 6.5 - II

$$
\begin{aligned}
RR(f \mid b.out = 0, c.out = 0, t) &= P_{f \mid b.out,c.out,T}(0 \mid 0, 0, t) \\
&= \sum_{b,d,e \in \{0,1\}} P_{f \mid b.out,e.out,d.out,T}(0 \mid b, d, e, t) \\
&\quad P_{b.out \mid b.out,c.out,T}(b \mid 0, 0, t) P_{d.out \mid b.out,c.out,T}(d \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(e \mid 0, 0, t) \\
&= \sum_{d,e \in \{0,1\}} P_{f \mid b.out,e.out,d.out,T}(0 \mid 0, d, e, t) \\
&\quad P_{d.out \mid b.out,c.out,T}(d \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(e \mid 0, 0, t) \\
&= P_{d.out \mid b.out,c.out,T}(0 \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(0 \mid 0, 0, t) \\
&\quad + P_{d.out \mid b.out,c.out,T}(1 \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(0 \mid 0, 0, t) \\
&\quad + P_{d.out \mid b.out,c.out,T}(0 \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(1 \mid 0, 0, t) \\
&= P_{d.out \mid c.out,T}(0 \mid 0, t) P_{e.out \mid c.out,T}(0 \mid 0, t) \\
&\quad + P_{d.out \mid c.out,T}(1 \mid 0, t) P_{e.out \mid c.out,T}(0 \mid 0, t) \\
&\quad + P_{d.out \mid c.out,T}(0 \mid 0, t) P_{e.out \mid c.out,T}(1 \mid 0, t) \\
&= 0.95 \cdot 0.975 + 0.05 \cdot 0.975 + 0.95 \cdot 0.025 \\
&= 0.99875
\end{aligned}
$$

$$
\begin{aligned}
RR(f \mid b.out = 1, c.out = 0, t) &= P_{f \mid b.out,c.out,T}(0 \mid 1, 0, t) \\
&= P_{d.out \mid b.out,c.out,T}(0 \mid 0, 0, t) P_{e.out \mid b.out,c.out,T}(0 \mid 0, 0, t) \\
&= 0.95 \cdot 0.975 \\
&= 0.92625
\end{aligned}
$$

$$
\begin{aligned}
RR(f \mid b.out = 0, c.out = 1, t) &= P_{f \mid b.out,c.out,T}(0 \mid 0, 1, t) \\
&= \sum_{d,e \in \{0,1\}} P_{f \mid b.out,e.out,d.out,T}(0 \mid 0, d, e, t) \\
&\quad P_{d.out \mid c.out,T}(d \mid 1, t) P_{e.out \mid ,c.out,T}(e \mid 1, t) \\
&= P_{f \mid b.out,e.out,d.out,T}(0 \mid 0, 1, 1, t) \\
&\quad P_{d.out \mid c.out,T}(1 \mid 1, t) P_{e.out \mid c.out,T}(1 \mid 1, t) \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
RR(f \mid b.out = 1, c.out = 1, t) &= P_{f \mid b.out,c.out,T}(0 \mid 1, 1, t) \\
&= \sum_{d,e \in \{0,1\}} P_{f \mid b.out,e.out,d.out,T}(0 \mid 1, d, e, t) \\
&\quad P_{d.out \mid c.out,T}(d \mid 1, t) P_{e.out \mid c.out,T}(e \mid 1, t) \\
&= P_{f \mid b.out,e.out,d.out,T}(0 \mid 0, 1, 1, t) \\
&\quad P_{d.out \mid c.out,T}(1 \mid 1, t) P_{e.out \mid ,c.out,T}(1 \mid 1, t) \\
&= 0
\end{aligned}
$$

(6.42)

Example 6.1.10: Relative Reliability in the Dependency Graph of Figure 6.5 - III

$$
\begin{aligned}
RR(g \mid c.out = 0, b.out = 0, t) &= P_{f \mid c.out,b.out,T}(0 \mid 0, 0, t) \\
&= \sum_{f,b \in \{0,1\}} P_{g \mid f.out,b.out,T}(0 \mid f, b, t) \\
&\quad P_{b.out \mid c.out,b.out,T}(b \mid 0, 0, t) P_{f.out \mid c.out,b.out,T}(f \mid 0, 0, t) \\
&= \sum_{f \in \{0,1\}} P_{g \mid f.out,b.out,T}(0 \mid f, 0, t) \\
&\quad P_{f.out \mid c.out,b.out,T}(f \mid 0, 0, t) \\
&= P_{f.out \mid c.out,b.out,T}(1 \mid 0, 0, t) + P_{f.out \mid c.out,b.out,T}(0 \mid 0, 0, t) \\
&= 1 \\
\\
RR(g \mid c.out = 1, b.out = 0, t) &= P_{f \mid c.out,b.out,T}(0 \mid 1, 0, t) \\
&= \sum_{f \in \{0,1\}} P_{g \mid f.out,b.out,T}(0 \mid f, 0, t) \\
&\quad P_{f.out \mid c.out,b.out,T}(f \mid 1, 0, t) \\
&= P_{f.out \mid c.out,b.out,T}(1 \mid 1, 0, t) + P_{f.out \mid c.out,b.out,T}(0 \mid 1, 0, t) \\
&= 1 \\
\\
RR(g \mid c.out = 0, b.out = 1, t) &= P_{f \mid c.out,b.out,T}(0 \mid 0, 1, t) \\
&= \sum_{f \in \{0,1\}} P_{g \mid f.out,b.out,T}(0 \mid f, 1, t) \\
&\quad P_{f.out \mid c.out,b.out,T}(f \mid 0, 1, t) \\
&= P_{f.out \mid b.out,c.out,T}(0 \mid 1, 0, t) \\
&= 0.99 P_{f \mid b.out,c.out,T}(0 \mid 1, 0, t) \\
&= 0.99 \cdot 0.92625 \\
\\
RR(g \mid c.out = 1, b.out = 1, t) &= P_{f \mid c.out,b.out,T}(0 \mid 1, 1, t) \\
&= \sum_{f \in \{0,1\}} P_{g \mid f.out,b.out,T}(0 \mid f, 1, t) \\
&\quad P_{f.out \mid c.out,b.out,T}(f \mid 1, 1, t) \\
&= P_{f.out \mid c.out,b.out,T}(0 \mid 1, 1, t) \\
&= 0
\end{aligned}
\tag{6.43}
$$

Example 6.1.11: Absolute Reliability in the Dependency Graph of Figure 6.5 - I

$$
\begin{aligned}
R(a.out\,|\,t) &= RR(a.out\,|\,t) \\
&= 0.9
\end{aligned}
$$

$$
\begin{aligned}
R(b.out\,|\,t) &= RR(b.out\,|\,a.out = 0, t)R(a.out\,|\,t) + RR(b.out\,|\,a.out = 1, t)(1 - R(a.out\,|\,t)) \\
&= RR(b.out\,|\,a.out = 0, t)R(a.out\,|\,t) \\
&= 0.85 \cdot 0.9 \\
&= 0.765
\end{aligned}
$$

$$
\begin{aligned}
R(c.out\,|\,t) &= RR(c.out\,|\,a.out = 0, t)R(a.out\,|\,t) + RR(c.out\,|\,a.out = 1, t)(1 - R(a.out\,|\,t)) \\
&= RR(c.out\,|\,a.out = 0, t)R(a.out\,|\,t) \\
&= 0.875 \cdot 0.9 \\
&= 0.7875
\end{aligned}
$$

$$
\begin{aligned}
R(d.out\,|\,t) &= RR(d.out\,|\,c.out = 0, t)R(c.out\,|\,t) + RR(d.out\,|\,c.out = 1, t)(1 - R(c.out\,|\,t)) \\
&= RR(d.out\,|\,c.out = 0, t)R(c.out\,|\,t) \\
&= 0.95 \cdot 0.7875
\end{aligned}
$$

$$
\begin{aligned}
R(e.out\,|\,t) &= RR(e.out\,|\,c.out = 0, t)R(c.out\,|\,t) + RR(e.out\,|\,c.out = 1, t)(1 - R(c.out\,|\,t)) \\
&= RR(e.out\,|\,c.out = 0, t)R(c.out\,|\,t) \\
&= 0.975 \cdot 0.7875
\end{aligned} \tag{6.44}
$$

$$
\begin{aligned}
R(f.out\,|\,t) &= RR(f.out\,|\,c.out = 0, b.out = 0, t)R(c.out\,|\,t)R(b.out,\,|\,t) \\
&\quad + RR(f.out\,|\,c.out = 1, b.out = 0, t)(1 - R(c.out\,|\,t))R(b.out,\,|\,t) \\
&\quad + RR(f.out\,|\,c.out = 0, b.out = 1, t)R(c.out\,|\,t)(1 - R(b.out,\,|\,t)) \\
&\quad + RR(f.out\,|\,c.out = 1, b.out = 1, t)(1 - R(c.out\,|\,t))(1 - R(b.out,\,|\,t)) \\
&= 0.99RR(f\,|\,c.out = 0, b.out = 0, t)R(c.out\,|\,t)R(b.out,\,|\,t) \\
&\quad + 0.99RR(f\,|\,c.out = 0, b.out = 1, t)R(c.out\,|\,t)(1 - R(b.out,\,|\,t)) \\
R(g.out\,|\,t) &= RR(g.out\,|\,c.out = 0, b.out = 0, t)R(c.out\,|\,t)R(b.out,\,|\,t) \\
&\quad + RR(g.out\,|\,c.out = 1, b.out = 0, t)(1 - R(c.out\,|\,t))R(b.out,\,|\,t) \\
&\quad + RR(g.out\,|\,c.out = 0, b.out = 1, t)R(c.out\,|\,t)(1 - R(b.out,\,|\,t)) \\
&\quad + RR(g.out\,|\,c.out = 1, b.out = 1, t)(1 - R(c.out\,|\,t))(1 - R(b.out,\,|\,t)) \\
&= R(c.out\,|\,t)R(b.out,\,|\,t) \\
&\quad + (1 - R(c.out\,|\,t))R(b.out,\,|\,t) \\
&\quad + RR(g.out\,|\,c.out = 0, b.out = 1, t)R(c.out\,|\,t)(1 - R(b.out,\,|\,t))
\end{aligned}
$$

Chapter 7

Conclusions

In these days, we use technical systems not only as additional support. In some cases, we even completely rely on them and trust them with our lives, like in the case of medical systems or vehicles. These systems are called "safety critical" because a failure in their functions can have an impact on our safety. Therefore we want these systems to be as reliable as possible. To archive a higher reliability, safety critical systems often contain redundant components, which may provide the desired functions if the original component fails.

The process of evaluating the likelihood of a function outage is called reliability engineering. In this thesis we proposed a method for reliability evaluation on a technical system consisting of communicating components. We made use of the already well known Bayesian Network approach for solving this and extended it to be applicable for more complex networks, namely those containing cycles - or also called feed back loops.

This is done by applying cycle unrolling, a technique inspired by the loop unrolling or code unwinding methods from compiler construction and model checking. We showed that it is sufficient to unroll a cycle two times in order to calculate the correct reliability, if we are neglect the time it takes for a wrong information to spread. This assumption let us perform the calculation on a snapshot of the model, where the system is in a fixed state.

For cycle-less systems, we propose an efficient method for calculating the reliability of every component within two steps: We determined special nodes in the Bayesian Network, the so called closest Branching Predecessors, a concept derived from dominator nodes. These nodes allow us, to calculate a conditional reliability, the relative reliability. Using this relative reliabilities, we can now calculate the absolute reliability of every component, without performing the same calculation twice.

Bibliography

Aljazzar, H. et al. [2009]. "Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples". In: *Quantitative Evaluation of Systems, 2009. QEST '09. Sixth International Conference on the*. Sep 2009, pages 299–308. doi:10.1109/QEST.2009.8 (cited on page 3).

Amari, S., G. Dill and E. Howald [2003]. "A new approach to solve dynamic fault trees". In: *Reliability and Maintainability Symposium, 2003. Annual*. 2003, pages 374–379. doi:10.1109/RAMS.2003.1182018 (cited on page 11).

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 22nd Oct 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page xi).

Appel, Andrew W. [1997]. *Modern compiler implementation in ML*. 1st Edition. Cambridge Univ. Press, 1997. ISBN 9780521582742 (cited on page 51).

Arabian-Hoseynabadi, H., H. Oraee and P. J. Tavner [2010]. "Failure Modes and Effects Analysis (FMEA) for wind turbines." *International journal of electrical power and energy systems.* 32.7 [Sep 2010], pages 817–824. doi:10.1016/j.ijepes.2010.01.019 (cited on pages 1, 7).

Barber, David [2012]. *Bayesian Reasoning and Machine Learning*. Cambridge Books Online, 2012. ISBN 9780521518147 (cited on page 27).

Biere, A. et al. [2009]. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009. ISBN 9781586039295 (cited on page 13).

Blischke, Wallace R. and D. N. P. Murthy [2000]. *Reliability: Modeling, Prediction, and Optimization*. English. 1st Edition. John Wiley & Sons Inc, 2000. ISBN 9780471184508 (cited on page 2).

Bobbio, Andrea et al. [2008]. "Reliability Analysis of Systems with Dynamic Dependencies". In: *Bayesian Networks*. John Wiley & Sons, Ltd, 2008, pages 225–238. ISBN 9780470994559. doi:10.1002/9780470994559.ch13 (cited on page 11).

Bobbio, A. et al. [2001]. "Improving the analysis of dependable systems by mapping fault trees into Bayesian networks". *Reliability Engineering & System Safety* 71.3 [2001], pages 249–260. ISSN 09518320. doi:10.1016/S0951-8320(00)00077-6 (cited on pages 2, 13).

Chiozza, Maria Laura and Clemente Ponzetti [2009]. "FMEA: A model for reducing medical errors". *Clinica Chimica Acta* 404.1 [2009]. Errors in Laboratory Medicine and Patient Safety, pages 75–78. ISSN 0009-8981. doi:10.1016/j.cca.2009.03.015 (cited on page 3).

Commission, U.S. Nuclear Regulatory [1998]. *Fault Tree Handbook (Nureg-0492)*. 1998. ISBN 9781782662495 (cited on pages 1, 9, 10).

Cooper, K.D. and L. Torczon [2004]. *Engineering a Compiler*. Computer Architecture Compilers. Morgan Kaufmann, 2004. ISBN 9781558606982 (cited on page 13).

Dugan, J. B., S. J. Bavuso and M. A. Boyd [1992]. “Dynamic fault-tree models for fault-tolerant computer systems”. *IEEE Transactions on Reliability* 41.3 [Sep 1992], pages 363–377. ISSN 0018-9529. doi:10.1109/24.159800 (cited on page 11).

Durrett, Rick [2010]. *Probability : Theory and Examples*. English. Cambridge University Press, 2010. ISBN 9780521765398 (cited on page 46).

Grabert, Matthias and Johann-Friedrich Luy [2012]. “Vehicle reliability field data analysis - Best practise at Mercedes-Benz cars”. English. In: IEEE, 2012, pages 203–203. doi:10.1109/IIRW.2012.6468958 (cited on page 1).

Høyland, Arnljot and Marvin Rausand [2004]. *System reliability theory: models and statistical methods*. English. 1st Edition. Wiley, 2004. ISBN 9780471593973 (cited on pages 1, 2, 47, 49).

Hura, G. S. and J. W. Atwood [1988]. “The use of Petri nets to analyze coherent fault trees”. *IEEE Transactions on Reliability* 37.5 [Dec 1988], pages 469–474. ISSN 0018-9529. doi:10.1109/24.9864 (cited on page 12).

IEEE Standard Glossary of Software Engineering Terminology [1990]. Technical report. 1990. doi:10.1109/ieeestd.1990.101064 (cited on page 2).

Kapur, Kailash C. and Michael Pecht [2014]. *Reliability Engineering*. English. 1st Edition. John Wiley & Sons Ltd, 2014. ISBN 9781118140673 (cited on pages 3, 10, 46).

Khakzad, Nima, Faisal Khan and Paul Amyotte [2011]. “Safety analysis in process facilities: Comparison of fault tree and Bayesian network approaches”. English. *Reliability Engineering and System Safety* 96.8 [2011], pages 925–932. ISSN 09518320. doi:10.1016/j.ress.2011.03.012 (cited on pages 3, 13, 27).

Koski, Timo and John Noble [2009a]. *Bayesian Networks: An Introduction*. English. 1st Edition. Wiley, 2009. ISBN 9780470684023 (cited on page 3).

Koski, Timo and John Noble [2009b]. “Conditional Independence, Graphs and d-Separation”. In: *Bayesian Networks*. John Wiley & Sons, Ltd, 2009, pages 37–79. ISBN 9780470684023. doi:10.1002/9780470684023.ch2 (cited on page 53).

Langseth, Helge and Luigi Portinale [2007]. “Bayesian networks in reliability”. *Reliability Engineering and System Safety* 92.1 [2007], pages 92–108. ISSN 0951-8320. doi:10.1016/j.ress.2005.11.037 (cited on pages 13, 52).

Malhotra, M. and K. S. Trivedi [1994]. “Power-hierarchy of dependability-model types”. *IEEE Transactions on Reliability* 43.3 [Sep 1994], pages 493–502. ISSN 0018-9529. doi:10.1109/24.326452 (cited on pages 5, 8).

Merle, G. et al. [2010]. “Probabilistic Algebraic Analysis of Fault Trees With Priority Dynamic Gates and Repeated Events”. *IEEE Transactions on Reliability* 59.1 [Mar 2010], pages 250–261. ISSN 0018-9529. doi:10.1109/TR.2009.2035793 (cited on page 11).

Rausand, Marvin and Arnljot Høyland [2003]. *System Reliability Theory: Models, Statistical Methods and Applications*. 2nd Edition. Wiley-Interscience, 2003. ISBN 9780471593973. doi:10.1002/9780470316900 (cited on page 8).

Reibman, A. L. and M. Veeraraghavan [1991]. “Reliability modeling: an overview for system designers”. *Computer* 24.4 [Apr 1991], pages 49–57. ISSN 0018-9162. doi:10.1109/2.76262 (cited on pages 5, 7).

Rohatgi, Vijay K. and A. K. Md. Ehsanes Saleh [2011]. *An Introduction to Probability and Statistics*. English. 2nd Edition. John Wiley & Sons Inc, 13th Sep 2011. ISBN 9780471348467 (cited on page 43).

Snooke, Neal and Chris Price [2012]. "Automated {FMEA} based diagnostic symptom generation". *Advanced Engineering Informatics* 26.4 [2012]. EG-ICE 2011 + SI: Modern Concurrent Engineering, pages 870–888. ISSN 1474-0346. doi:10.1016/j.aei.2012.07.001 (cited on page 7).

Stirzaker, David [1999]. *Probability and Random Variables: A Beginner's Guide*. Cambridge University Press, 1999. ISBN 9780521644457 (cited on pages 42, 43).

Tarjan, Robert Endre [1972]. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1.2 [1972], pages 146–160 (cited on pages 5, 28).

Vesely, W. et al. [2002]. *Fault Tree Handbook with Aerospace Applications*. Handbook. National Aeronautics and Space Administration, 2002 (cited on pages 1, 10).