Fault Tree Analysis using the Neo4j Traversal Framework

Kevin Bartik

Prof. Dr.-Ing. Peter Liggesmeyer

Bachelor Thesis

Bachelorarbeit

Fault Tree Analysis using the Neo4j Traversal Framework

Kevin Bartik

Lehrstuhl Software Engineering: Dependability
am Fachbereich Informatik der
Technische Universität Kaiserslautern

Tag der Ausgabe	: 08.11.2017
Tag der Abgabe	: 04.05.2017
Gutachter	: Prof. Dr.-Ing. habil. Liggesmeyer
Betreuer	: Dipl.-Inform. Felix Möhrle

Hereby I declare that I have self-dependently composed the bachelor's thesis at hand. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

I declare that the submitted written version is consistent with the attached digital version.

Kaiserslautern, den 04.05.2017

Kevin Bartik

Zusammenfassung. Im Bereich der Funktionssicherheit gibt es viele Verfahren, die nötige Informationen liefern, um Systeme sicher und zuverlässig zu machen. Eines der meistbekannten Verfahren ist Fehlerbaumanalyse. Mit größeren und komplexeren Systemen wird die Sicherheitsanalyse bedeutend schwieriger und die Ergebnisse sind weniger genau. Neuere Verfahren, wie z.B. Komponenten-Fehlerbäume helfen dabei, die Komplexität von Fehlerbäumen zu reduzieren und die Vorteile moderner Softwareentwicklung zu nutzen. Die Software, die heutzutage für die Beschreibung und Analyse dieser Fehlermodelle eingesetzt wird, nutzt aber weiterhin konventionelle Speicherungs- und Berechnungsmethoden. Der Zweck dieser Arbeit ist die Implementierung einer kleinen Anwendung zur Analyse von Komponenten-Fehlerbäumen mit Verwendung von Neo4j, einer Graphdatenbank. Die angestrebten Vorteile dieser Implementierung sind verbesserte Speicher- und Zugriffsmöglichkeiten in der Graphdatenbank sowie eine kontinuierliche Analyse der Komponenten-Fehlerbäume bei Veränderungen und die Speicherung von Zwischenergebnissen. Das Ziel ist eine Verbesserung der durchschnittlich benötigten Analysezeit bei häufig veränderten Komponenten-Fehlerbäumen. Die entwickelte Anwendung konnte zeigen, dass dieses Ziel sowie die meisten Vorteile erreichbar sind und in dieser Arbeit wurde mit verschiedenen Ansätzen gezeigt wie eine effiziente Nutzung der Neo4j Graphdatenbank zur Analyse von Komponenten-Fehlerbäumen möglich ist und welche Vorteile diese haben.

Abstract. In the world of safety engineering, many safety techniques provide the information to make systems safe and reliable. One of the most well-known of these techniques is fault tree analysis. However, with larger and more complex systems, safety analysis becomes increasingly difficult, and results are less accurate. Newer techniques such as the component fault tree, help to reduce the complexity of traditional fault trees and make use of modern software design techniques. Software tools that are used to model and analyze these failure models still rely on old calculation methods and inefficient storage. The aim of this thesis is the implementation of component fault tree analysis in a small prototype with the use of the Neo4j graph database as a backend. The predicted advantages of such an approach are improved storage and access efficiency of component fault trees. The goal is to improve the average execution time of analysis on frequently altered component fault trees by reusing analysis results. The developed prototype showed that it is possible to achieve this goal and most of the advantages. This work shows different approaches to using the Neo4j graph database for analysis of component fault trees as well as the benefits and drawbacks of each approach.

Table of Contents

1 Introduction 8

2 Safety Analysis 10

2.1 Safety-critical systems 10

2.2 Fault Tree Analysis 11

2.3 Analysis Techniques 13

2.4 Component Fault Trees 18

2.5 Component-Integrated Component Fault Trees 21

2.6 Tools for Safety Analysis 23

2.6.1 Enterprise Architect 23

2.6.2 IESE I-SafE Extension 24

3 Neo4j Graph Database 28

3.1 Basics of Neo4j 28

3.2 Cypher Graph Query Language 30

3.2.1 Cypher Patterns 30

3.2.2 Cypher Clauses 31

3.3 Neo4j Browser 33

3.4 Neo4j Java Driver 34

3.5 Neo4j Java API 35

4 Importing Safety Models to the Prototype 38

4.1 Idea and Setup 38

4.2 Neo4j Graph Data Model 39

4.3 Importing Safety Models from Enterprise Architect 43

4.3.1 Relevant Data and Visual Representation 43

4.3.2 Using the Prototype to import Safety Models 46

4.4 Enterprise Architect Extension for the Prototype 48

4.4.1 Socket Communication 48

4.4.2 Update from the Enterprise Architect Extension 50

4.4.3 Analysis Window of the Enterprise Architect Extension 52

5 Analysis of Safety Models with the Prototype 53

5.1 Prerequisites for the Analysis 53

5.1.1 Neo4j Traversal Framework 53

5.1.2 Traversal Results 56

5.1.3 Analysis Results 58

5.2 Traversal Implementation of the Prototype 61

5.2.1 Traversal State 61

5.2.2 Traversal for finding Cycles 63

5.2.3 Traversal without the Neo4j Traversal Framework 65

5.2.4 Traversal with the Neo4j Traversal Framework 66

5.3 Analysis Implementation of the Prototype 70

5.3.1 Analysis Implementation for the Traversal Results 70

5.3.2 Storing and Reusing Analysis Results in the Neo4j Database 75

5.3.3 Connecting Traversal and Analysis Results 77

5.3.4 Performing the Analysis 78

6 Evaluation of the Prototype 81

6.1 Using the final Prototype 81

6.1.1 Using the GUI for Analysis 81

6.1.2 Additional Functionality of the GUI 84

6.2 Evaluation of the Prototype 86

6.2.1 Evaluation of the Analysis without Reuse 86

6.2.2 Evaluation of the Analysis with Reuse 88

6.3 Limitations of the Prototype 89

6.3.1 Practical Quantitative Analysis 89

6.3.2 Performance Considerations 90

7 Conclusion and further work 91

References 92

List of Figures 94

List of Tables 95

List of Abbreviations 96

1 Introduction

The evolution of new technical systems especially complex autonomous systems has been very rapid in the last few years. Many of these systems carry certain risks, in particular, risks for the safety of nearby people and the environment. Consequently, before such systems come to the market, their safety needs to be evaluated. This process of evaluation is called safety analysis and is standard practice for all systems that have the potential to cause catastrophes, for instance, nuclear power plants. The idea of structured safety analysis has been around for at least 70 years, at that time, military weapon systems started to become so advanced and dangerous that failure was unacceptable. Therefore, much research was done on finding better methods for safety analysis, and various techniques became the standard for the industry.

One of the most well-known methods is fault tree analysis which has seen use in almost every field of engineering for the last few decades. Many extensions and new ways of performing fault tree analysis have been introduced, and there are various industry standards. Today, there are many commercial software tools such as FaultTree+[1] available which help to perform fault tree analysis. However, often these tools only feature the traditional fault tree analysis, and many alternative standards are only implemented in specialized tools without the high-performance analysis provided by commercial applications.

The idea of this work is to show an alternative to the conventional analysis methods that improves the performance in non-commercial analysis tools. Recent developments in graph databases, such as Neo4j, provide new efficient ways to store heavily connected data and allow high-performance access. A graph database has benefits for fault tree analysis because the fault tree model is inherently a graph. Thus, a graph database stores a fault tree efficiently and together with high-performance graph traversal algorithms the analysis is very fast. Additionally, the analysis results can be directly stored in the graph of the fault tree which improves the performance of the analysis in the case of modifications to the fault tree.

The purpose of this thesis was the development of a prototype that utilizes the Neo4j graph database to store component fault trees, an extension of fault trees and performs a qualitative safety analysis on them. Also, the prototype was implemented to be able to perform a continuous analysis so that the analysis is performed whenever the component fault tree is modified. In this case, the Neo4j graph database stores analysis results and reuses them to improve analysis performance. The prototype uses the Neo4j Traversal Framework for graph traversal and analysis of the component fault trees. The Neo4j

[1] https://www.isograph.com/software/reliability-workbench/fault-tree-analysis/
(accessed: 15.03.2017)

Traversal Framework is a core component of Neo4j, and it allows fast retrieval of relevant data in the graph which can be used for the analysis. This work explains the Neo4j Traversal Framework in detail and describes further use in safety analysis beyond the analysis of component fault trees. Furthermore, the prototype makes use of intelligent multithreading during the traversal and analysis which improves performance.
The remainder of this thesis is structured as follows: Section 2 introduces safety analysis with the focus on fault tree analysis and component fault trees. Enterprise Architect and the I-SafE application that support modeling and analysis of component fault trees are also presented. In Section 3, the basics of the Neo4j graph database and its query language Cypher are introduced. An introduction to the Neo4j Java API concludes this section. Next, Section 4 describes the ideas, the goals and the setup of the prototype that was implemented. The Neo4j graph data model, as well as the implementation of the data transfer from Enterprise Architect to the Neo4j graph database, are presented. Also, the implementation of a small Enterprise Architect extension is shown which is used for continuous analysis in the prototype. Then, in Section 5, the realization of two approaches to the analysis of component fault trees are explained. Section 6 demonstrates the prototype, and both implementations of the analysis are evaluated. Also, the limitations of the prototype are discussed. Lastly, Section 7 concludes this thesis, presents the results and provides ideas for further work.

2 Safety Analysis

Many important concepts from safety analysis are used in the realization of the prototype. Therefore, the relevant terminology and methodology of safety analysis need to be established. In the following subsections, the main concepts for safety-critical systems and safety analysis, as well as some tools for safety analysis, are introduced. Firstly, in Sections 2.1 – 2.3 general safety terms, traditional fault tree analysis, and the most common analysis techniques are explained. Then in Section 2.4 and 2.5, the fault tree extension component fault tree, and the more refined component-integrated component fault tree are introduced. These two are the safety models that are utilized in the prototype because they have some advantages for the analysis. Lastly, Section 2.6 gives a brief introduction to the safety analysis tools that were used for the prototype.

2.1 Safety-critical systems

The modern world has large, complex and potentially dangerous systems e.g. nuclear power plants, airplanes, and chemical facilities. Any critical failure in these systems can endanger human life, cause significant economic loss or damage the environment. These systems are called safety-critical, and they require much safety analysis to be considered safe. “Safety is a state where the danger (risk) of a personal or property damage is reduced to an acceptable value” (DIN EN ISO 8402) [1], and safety analysis aims at proving or at least reasonably assuring that the actual risk is below the acceptable risk [1]. In safety analysis, a failure is inconsistent behavior w.r.t. specified behavior while running a system, a fault is a statically existent cause of a failure, and an error is a basic cause of the fault [1]. Failures can lead to hazards meaning that the occurrence of an accident becomes dependent on factors that are not controllable by the system [1]. In this case, accident means an undesired event that causes significant harm to people, goods or the environment [1]. Therefore, the goal of safety analysis is to identify failures and evaluate the probability of failure occurrence.
There are two main types of safety analysis, inductive and deductive. Inductive approaches such as Failure Modes and Effects Analysis (FMEA) start with a basic cause in the system and continue to identify effects of this primary cause. This type of safety analysis is very effective to find failure causes during development and fix them early on. On the other hand, deductive approaches such as fault tree analysis, start with an undesired event, and the following analysis consists of finding all relevant causes that can lead to this event. Fault tree analysis is an essential part of this work, and it is presented in the following section.

2.2 Fault Tree Analysis

Fault tree analysis[2] (FTA) is a top-down, deductive failure analysis that aims to find all related causes for a certain undesired event [2]. For example, in a system *Airplane,* the undesired event is *Airplane crash,* and the next step is to identify causes that can lead to this event, for example, *Human error* or *Engine failure*. These intermediate events can be analyzed further, and this leads to chains of events that can cause the undesired event at the top. FTA is a very simple but also effective way to perform safety analysis and has been in use since the 1960s for safety-critical systems in the military. Not much later it became the industry standard for all safety-critical systems.

FTA always begins with the creation of a fault tree (FT) which is started with the undesired event, called top event. Then during the analysis, lower-level events that can cause the top event are identified and added to the FT. If more events can cause the top event or if different combinations of events cause it then a logical gate is added between the top event and the lower-level events. In FTs, various events and logical gates are used, and the most common ones are displayed in Figure 1. A full list of all symbols used in FTA can be found in [2].

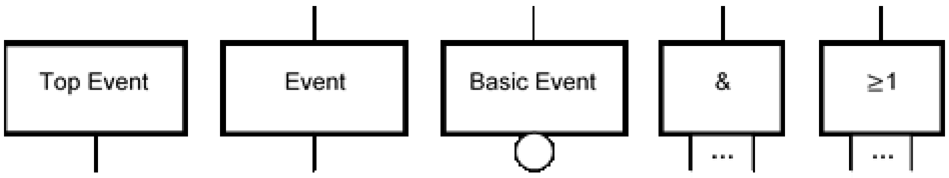


Figure 1 Top Event, Event, Basic Event, AND gate and OR gate

A top event is usually an unacceptable event or critical failure in a system or on the boundary of the system. Basic events are events on the lowest level in a FT where further refinement is not necessary or not possible. All events between the top event and basic events are also called intermediate events. The gates work with Boolean logic, so to cause the event above an AND (&) gate, all events below it must occur and to cause the event above an OR (≥1) gate, at least one event below it must happen. Other gates described in [2] are XOR (=1), Priority AND, INHIBIT, Combination (≥n) and NOT. There are even more gates and events described in other FTA standards, but this work only utilizes the events and gates from Figure 1 as well as the XOR, NOT and Combination gate which are explained in Section 5 and are not part of this introduction. For larger FT, where the size exceeds one page, transfer symbols are used to allow the concatenation of independent subtrees. In Figure 2, an example of a FT is presented.

[2] https://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf (accessed: 15.03.2017)

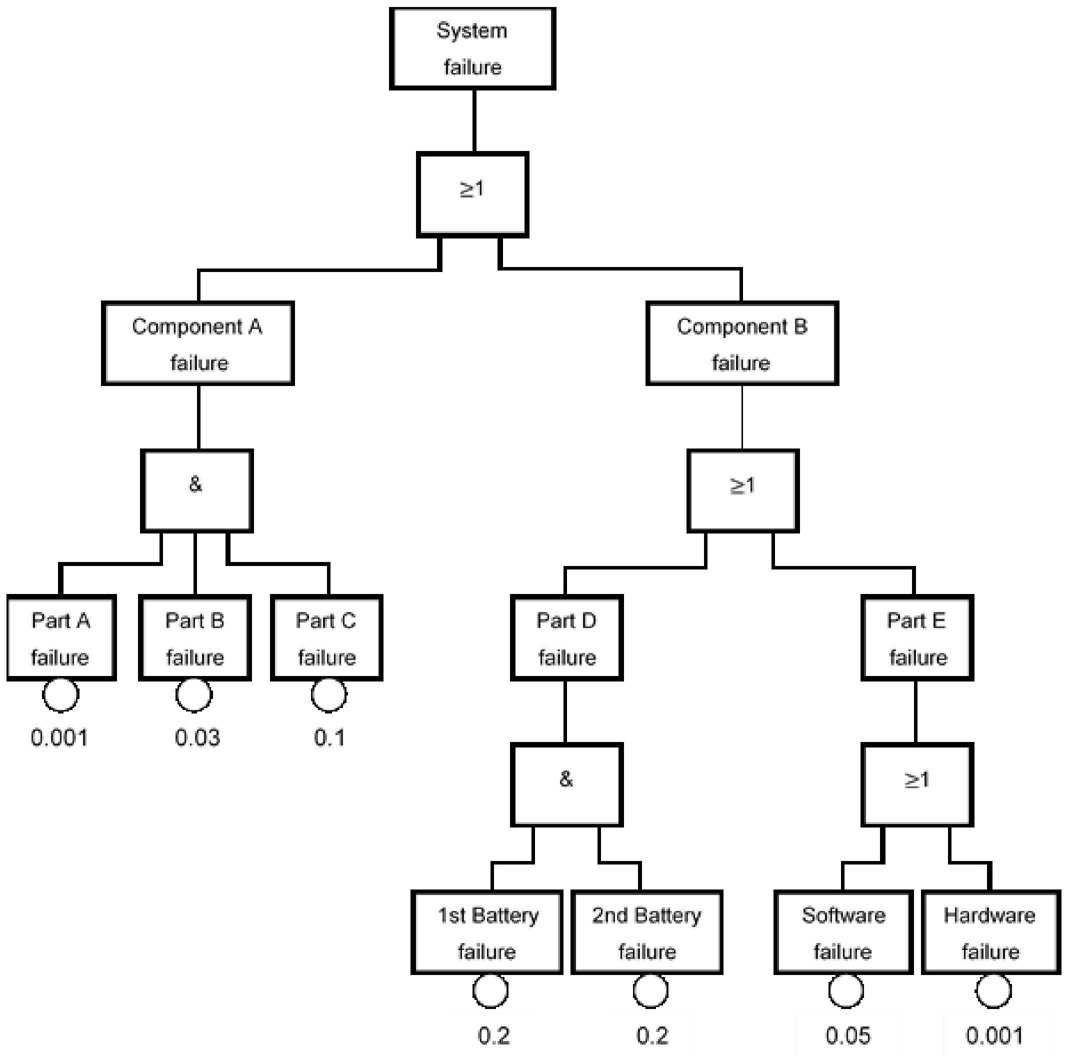


Figure 2 Fault tree example

The FT example has the top event *System failure* and right below is an OR gate which has two input events. In this case, either *Component A failure* or *Component B failure* will cause the top event. While *Component A failure* requires the failure of all its *Parts*, *Component B failure* will occur if one of its *Parts* fails. Also, *Part D* and *Part E* have more refinement into lower-level events. The numbers below the basic events are the failure probabilities of them. After the FT is completed, the next step is to analyze it further and interpret the results. The analysis of an FT can be done qualitatively and quantitatively. For both qualitative and quantitative analysis, the minimal cut set approach is available, but for more efficient quantitative analysis in larger FT, the Binary Decision Diagram (BDD) approach is more advisable [2]. The next section introduces these techniques.

2.3 Analysis Techniques

A minimal cut set is a minimal collection of basic events that is required to cause the top event [2]. Construction of all minimal cut sets can be done recursively starting at the top event. The following steps describe a simple algorithm to construct all minimal cut sets for simple FT with the events and gates from Figure 1:

Step 1: Create a set. Add the top event.

Step 2: Pick a set that contains intermediate events or the top event.

Step 3: Pick an intermediate event or the top event from that set and go to the event or gate below it. Depending on the event or gate, do one of the steps 4-6 and then continue with step 7.

Step 4: In the case of an intermediate or basic event: Add that event to the set. Remove the event above it from the set.

Step 5: In the case of an AND gate: Add all input events of the gate to the set. Remove the output event of the gate from the set.

Step 6: In the case of an OR gate: Create a set with the contents of the current set for each input event of the gate and add that input event to the set. Remove the output event of the gate from all sets.

Step 7: Continue with step 2 if any set still contains intermediate events.

Step 8: Remove any set that is not minimal.

Use of the algorithm on the example FT from Figure 2 is illustrated below.

Step 1

{System failure}

Step 2 -> Step 3 -> Step 6

{Component A failure}

{Component B failure}

Step 7 -> Step 2 -> Step 3 -> Step 5

{Part A failure, Part B failure, Part C failure}

{Component B failure}

Step 7 -> Step 2 -> Step 3 -> Step 6

{Part A failure, Part B failure, Part C failure}

{Part D failure}

{Part E failure}

Step 7 -> Step 2 -> Step 3 -> Step 5

{Part A failure, Part B failure, Part C failure}

{1st Battery failure, 2nd Battery failure}

{Part E failure}

Step 7 -> Step 2 -> Step 3 -> Step 6

{Part A failure, Part B failure, Part C failure}

{1st Battery failure, 2nd Battery failure}

{Software failure}

{Hardware failure}

For qualitative analysis, minimal cut sets provide very useful information as they can show weaknesses in a system. If a minimal cut set contains few events, then these events are more important to the safety of the system than others. In this example, the top event is triggered in case of *Software failure* or *Hardware failure* so both of these events must have a very low probability otherwise the top event will have a high probability. On the other hand, if the event *Part C failure* has a rather high probability and *Part A failure* and *Part B failure* both have low probabilities then the top event still has a low probability, and therefore, *Part C failure* is an event with a lower impact on the system. Qualitative analysis is very important in FTA, and most software tools can calculate the minimal cut sets of a FT. However, this calculation is not simple for larger FTs because of the high number of minimal cut sets, and each refinement step potentially creates more minimal cut sets. The implemented prototype features minimal cut set calculation and shows ways to improve the performance of the calculation.
In quantitative analysis, the goal is to calculate the exact probability of the top event. Of course, to perform a meaningful analysis, the probabilities of basic events need to be known, and any stochastically dependent events must be known as well. For the purpose of this introduction, only a quantitative analysis with stochastically independent events is shown. The most simple way of performing the quantitative analysis is the bottom-up calculation. This method starts at the basic events and calculates the probability of higher level events using different formulas depending on the next gate. In the example from Figure 2, the failure probabilities have already been calculated for the basic events. The formulas (1) and (2) show how to calculate the probabilities for the two gates used in this introduction [1]. The $\text{output}_{\text{AND}}$ and $\text{output}_{\text{OR}}$ are the output events of the gate, and the input_k is an input event while n is the number of all input events of that gate.

(1) AND gate: $P(\text{output}_{\text{AND}}) = \prod_{k=1}^{n} P(\text{input}_k)$

(2) OR gate: $P(\text{output}_{\text{OR}}) = 1 - \prod_{k=1}^{n}(1 - P(\text{input}_k))$

These formulas are used to calculate the probability of all events in the example which is shown below.

P (*Component A failure*)	= P (*Part A failure*) * P (*Part B failure*) * P (*Part C failure*)
	= 0.001 * 0.03 * 0.1
	= <u>0.000003</u>
P (*Part D failure*)	= P (*1st Battery failure*) * P (*2nd Battery failure*)
	= 0.2 * 0.2
	= <u>0.04</u>
P (*Part E failure*)	= 1 – ((1 – P (*Software failure*)) * (1 – P (*Hardware failure*)))
	= 1 – ((1 – 0.05) * (1 – 0.001))
	= <u>0.05095</u>
P (*Component B failure*)	= 1 – ((1 – P (*Part D failure*)) * (1 – P (*Part E failure*)))
	= 1 – ((1 – 0.04) * (1 – 0.05095))
	= <u>0.088912</u>
P (*System failure*)	= 1–((1–P(*Component A failure*))*(1–P(*Component B failure*)))
	= 1 – ((1 – 0.000003) * (1 – 0.088912))
	= <u>0.088914733264</u>

The bottom-up calculation can be inefficient for larger FTs, but it is also a very simple method and does not require any additional steps. With the use of minimal cut sets, an approximation of the top event probability (P_{TE}) can be made [1] without calculating the probabilities of intermediate events. The top event probability is the sum of the probabilities of all minimal cut set (P_{mcs}). The probability of a minimal cut set is the product of the probabilities of all events (P_i) in that minimal cut set. The formulas (3) and (4) are shown below.

(3) $P_{TE} = \sum_{\text{all mcs}} P_{mcsi}$

(4) $P_{mcs} = \prod_{\text{all events in mcs}} P_i$

Below, the approximation of the top event *System failure* is calculated.

P (*System failure*)	= (0.001 * 0.03 * 0.1) + (0.2 * 0.2) + 0.05 + 0.001
	= <u>0.091003</u>

This approximation only produces accurate results for basic events with very small probabilities (<0.1), and since the example has some higher probabilities, the result is not precise. However, it is possible to calculate the exact probability of the top event with minimal cut sets, but it requires similar calculation effort compared to the simple bottom-up calculation.

The other way of performing the quantitative analysis is to use Binary Decision Diagrams (BDDs) [3]. BDDs are a newer approach to quantitative analysis in fault trees, and they have some benefits and drawbacks compared to minimal cut sets. They can be thought of as a graphical representation of a data structure for a logic function [2]. They are constructed bottom-up from the fault tree, and each basic event represents a single node BDD. There are several ways to draw BDDs but for the following examples, the right line from a node means true (1), and the left dashed line represents false (0). The BDDs of *Part A failure* (*PA*), *Component A failure*, and *Part E failure* from the example FT are depicted in Figure 3.

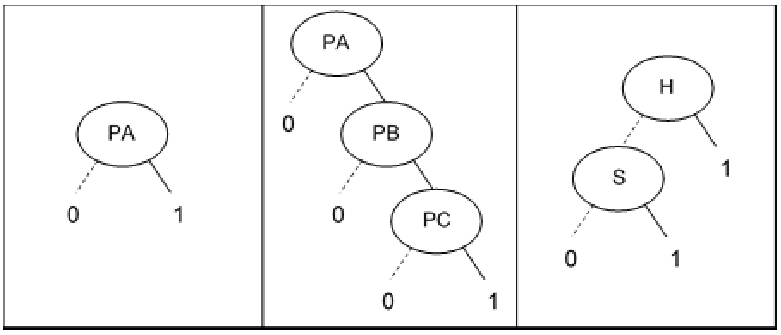


Figure 3 BDDs of Part A failure | Component A failure | Part E failure

The left BDD is *Part A failure* (*PA*) which is a basic event so it can either be false (0) or true (1). The BDD in the middle shows *Component A failure* which is *Part A failure* (*PA*), *Part B failure* (*PB*) and *Part C failure* (*PC*) connected to an AND gate. Only if every *Part* returns true, the BDD is true. In this case, the *Parts* can be ordered differently, but for larger BDDs the ordering of the variables has a significant impact on the size and complexity of the BDD [4]. Lastly, the right BDD shows *Part E failure* meaning *Hardware failure* (*H*) and *Software failure* (*S*) connected to an OR gate. In Figure 4, the complete BDD of the top event *System failure* is depicted in two ways, one being a regular BDD and the other one being a Reduced Ordered BDD (ROBDD) which is a BDD that has been reduced to the minimum size with optimal variable ordering and the removal of unnecessary nodes.

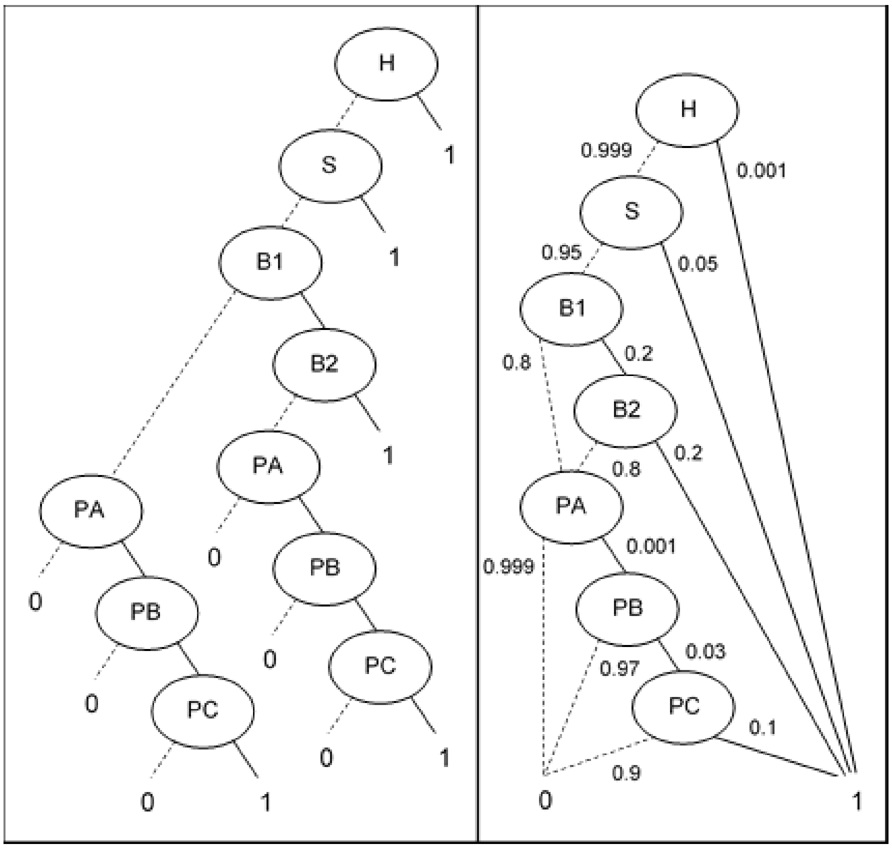


Figure 4 BDD and ROBDD of the top event System failure

Finding the optimal variable ordering is a problem with a high complexity, and therefore it is not always possible to find the best variable ordering in a reasonable time [4].
When the BDD is completed, the calculation of the top event probability is done by adding the probabilities of each success path together. A success path is a path that results in true (1), and the product of probabilities for each part of the success path is the probability of the success path. In the ROBDD from Figure 4, the probabilities for all paths are visible, and they equal to those from Figure 2. The probability of the top event is the sum of all success paths probabilities. The example is calculated below for each success path and the top event.

P (H -> 1) = <u>0.001</u>
P (H -> S -> 1) = 0.999 * 0.05
= <u>0.04995</u>
P (H -> S -> B1 -> B2 -> 1) = 0.999 * 0.95 * 0.2 * 0.2
= <u>0.037962</u>
P (H -> S -> B1 -> PA -> PB -> PC -> 1) = 0.999 * 0.95 * 0.8 * 0.001 * 0.03 * 0.1
= <u>0.00000227772</u>
P (H -> S -> B1 -> B2 -> PA -> PB -> PC -> 1) = 0.999 * 0.95 * 0.8 * 0.8 * 0.001 * 0.03 * 0.1
= <u>0.000001822176</u>
P (*System failure*) = <u>0.088914733264</u>

The result is equal to the result that was obtained using the bottom-up technique. Overall, quantitative analysis using BDDs requires extra effort in the form of creating a BDD first and then calculating the probabilities. However, there are significant performance benefits when using BDDs for more complex quantitative analysis. In realistic FTs, the failure probabilities are more complex and include failure rates over time. Therefore, the failure rate is not a discrete number but function of failure rates over time. For example, mechanical parts tend to fail more often the longer they are active, so the failure rate increases over time. The probability calculation of such a quantitative analysis is very complex and not part of this work. The implemented prototype only features a simple quantitative analysis using failure probability that is constant and bottom-up calculation because the focus was on qualitative analysis using minimal cut sets. Further work must be done to evaluate if it is possible to effectively use the approach of this work in more complex quantitative analysis. In Section 6.3.1, a discussion about reuse of complex quantitative results in the Neo4j database can be found.
All these analysis techniques can be used for the following two fault tree extensions that are called component fault trees and component-integrated component fault trees. They are introduced in the next section.

2.4 Component Fault Trees

Since the late 1960s, FTA was used extensively and became very popular in many fields of engineering. A problem for many engineers is that traditional FTA has some disadvantages and cannot be used to describe every system precisely. Additionally, it lacks some of the basic system design principles that have developed over the years. Therefore, many FTA extensions and different FTA standards emerged, and they added modeling capabilities and analysis refinements, and some changed the whole structure of the FT. In this work, the FTA extensions component fault tree, as well as the component-integrated component fault tree, are used in the implemented prototype because they follow basic software design principles and have some analysis advantages.
The component fault tree (CFT), first introduced in [5], are a large contribution to FTA and have many advantages over traditional FTA. The idea of CFTs is the abstraction of an FT into components and to perform failure analysis while preserving the system structure within the failure model. Most systems can be divided into various components e.g. car consists of an engine, a transmission, and other parts. Therefore, it makes sense to analyze smaller components independently and later use those failure models for the analysis of the complete system. Before CFTs are explained, some adjustments to the traditional FT are introduced. One major problem in FTA is the use of repeated events which are events that are equal but appear multiple times in the tree. Because of the tree

structure, an event cannot have multiple paths to the top event, but in real systems, single events can have multiple effects on a system. For this reason, these events are repeated in the FT, and they are marked, so the computation method (minimal cut sets or BDDs) produces correct results. However, in a large FTs, repeated events can lead to inconsistencies during FT development and maintenance. To overcome this drawback, an FT can be changed to a Cause Effect Graph (CEG). A CEG is a directed acyclic graph instead of a tree, and this has some advantages. Firstly, repeated events are represented only once and have multiple paths to higher level events in the CEG which improve readability and maintainability for larger FT because complex systems tend to have many common causes and therefore many repeated events. Secondly, CEG may have more than one top event which helps to find relations between various undesired events. Figure 5 shows a CEG with two top events and a common cause.

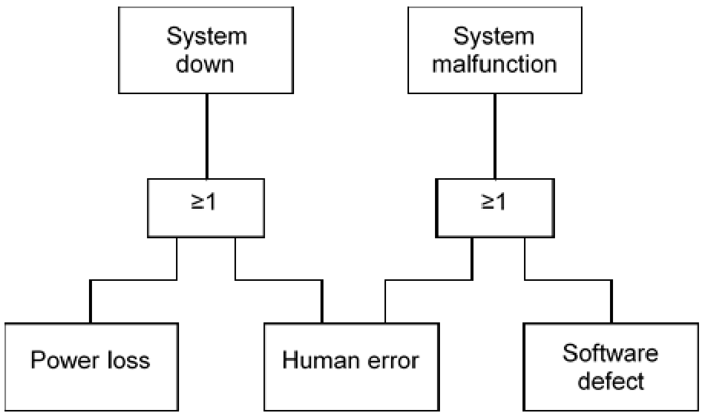


Figure 5 Cause Effect Graph

CEGs are the basis for CFTs because they remove the restriction that one event has only one effect, and they also allow multiple top events. The concept of CFTs goes a step further and aims to reuse larger parts of an FT. As stated before, the idea of CFTs is to retain the system structure inside the failure model. A CFT is a CEG with two additional symbols, the input port (inport), and the output port (outport). They are represented as solid triangles and serve as entry and exit failure modes of a CFT. The inports can be considered as basic events that are dependent on an outside source, and the outports can be regarded as top events that may serve as events to inports of other CFTs. CFTs can contain other CFTs as part of their failure model, and this means that CFTs are reusable. There are some restrictions to prevent cycles inside a CFT, for example, a CFT cannot contain itself, and two equal sub-CFTs cannot be connected to each other. In Figure 6, an example CFT is shown to illustrate the differences to traditional FTs.

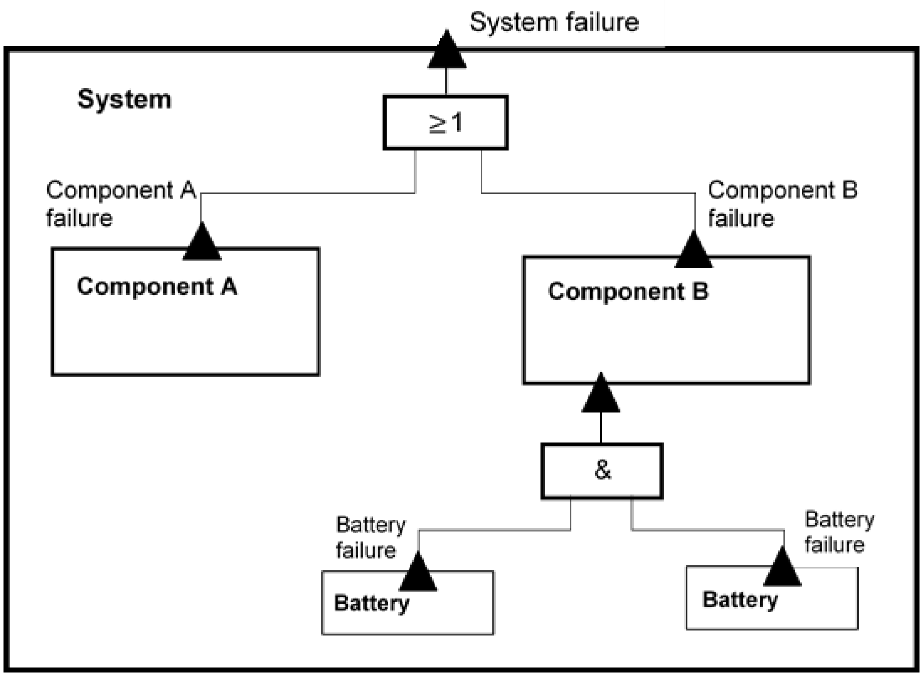


Figure 6 CFT example

This model represents the example FT from Figure 2 with some modifications to show the advantages of CFTs. The main difference is that *Part D* is split into two equal components *Battery* and they are external to *Component B*. The detailed CFTs of *Component A* and *Component B* are shown in Figure 7.

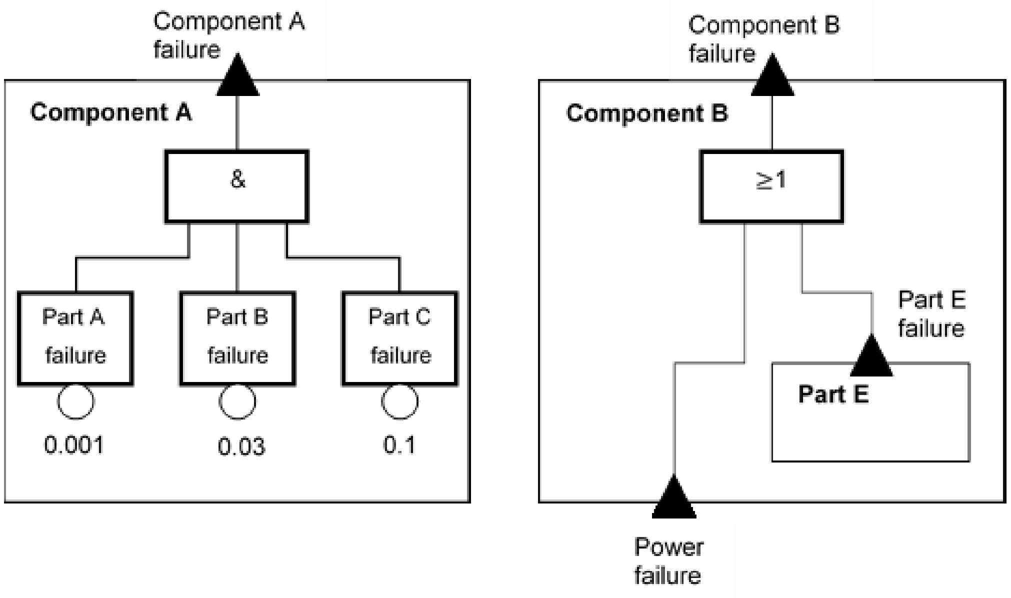


Figure 7 CFTs of Component A and B

Component A is unchanged compared to the previous example FT, and *Component B* has one inport for a potential power failure, and it has the component *Part E* inside. The CFTs of *Battery* and *Part E* are depicted in Figure 8.

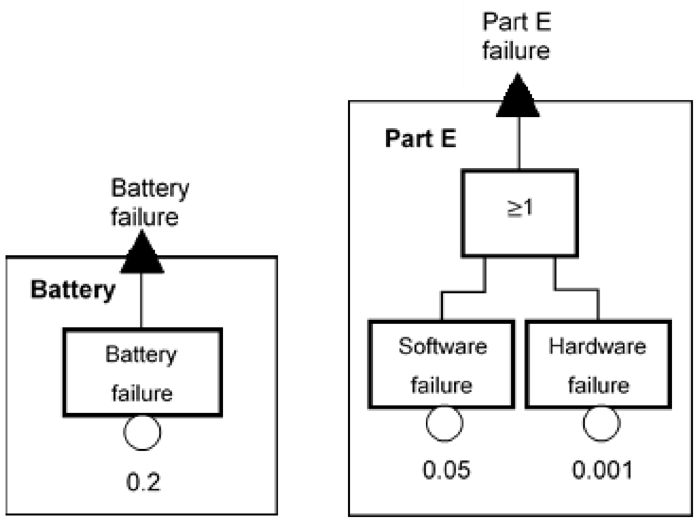


Figure 8 CFTs of Battery and Part E

The CFT *Battery* is reused in the system because both *Batteries* are equal and have the same failure probability. Analysis can be done individually for each component at the outports. For quantitative analysis of a CFT with inports, all of them need to be connected to an outside source because otherwise, their contribution cannot be included in the calculation. Overall, the analysis techniques used are the same as in FTA, and both minimal cut sets and BDDs still work.

2.5 Component-Integrated Component Fault Trees

Component-Integrated Component Fault Trees (C^2FTs) [6], [7] are an extension to CFTs that link them directly to the functional components of the system. C^2FTs are based on the principle of Component-based Software Engineering (CBSE) which relies on the hierarchical abstraction of functional components. When they were first introduced, C^2FTs were intended for safety analysis in software engineering, but in a simplified form which is presented here, they can be used for other systems as well. The idea is to have both a specification and a realization of a component in the system. The specification hides all internal details and only specifies functionality and any inputs and outputs of a component. On the other hand, the realization shows internal details and the specifications of internal components. Figure 9 shows an example.

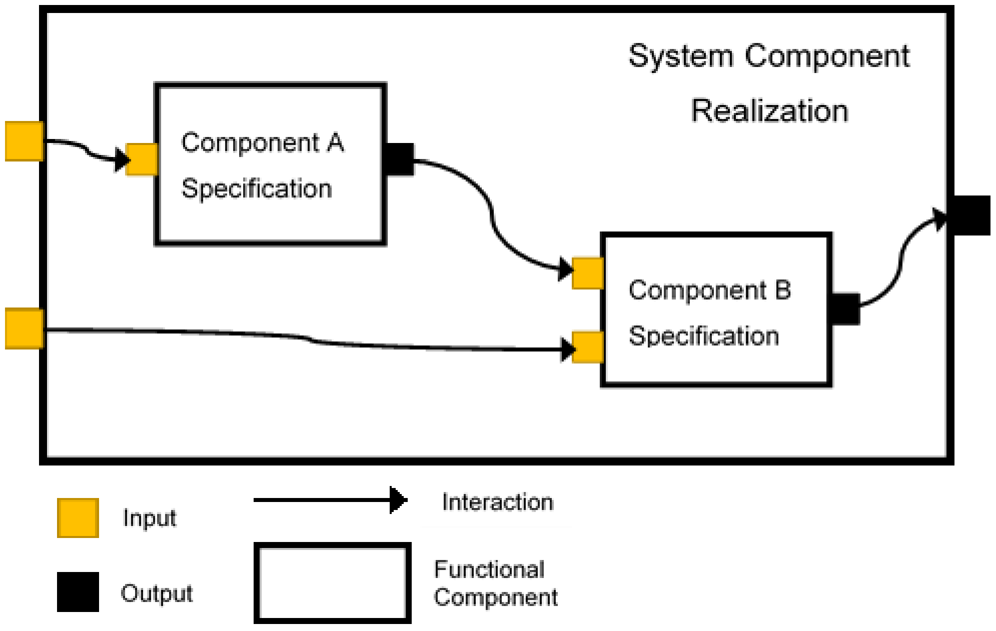


Figure 9 Realization and Specification

In the example, the *System Component* is visible in its realization. It has two inputs and one output and contains two subcomponents. They are represented in their specification and interact with the system and each other over inputs and outputs. These two components also have a realization which contains their subcomponents. Depending on how complex the system is and how many components interact with each other there can be many abstraction levels. With this approach, distributed development is simple because components are developed to meet their specification, and internal details do not need to be visible. This abstraction makes integration easier and helps developers to keep an overview of the system. Another benefit is the protection of intellectual property because the realization stays hidden. These techniques can be applied for CFTs as well, and they are already visible in the CFT examples from the previous section. In Figure 6, only the specification of the *Components A and B* is seen, and in Figure 7 the realization of these components is displayed. Lastly, in Figure 8, the realization of the remaining subcomponents is shown. To create a complete C^{2}FT, all these CFTs are connected to the functional components of the system. These functional components are the technical description of the real components, and a CFT can be assigned to them as a failure model. Furthermore, CFT inports and outports can be directly assigned to inputs and outputs of a functional component to trace failures inside a system. Additionally, various failure types such as *omission failure* or *value failure* can be assigned to the inports and outports of CFTs. These failure types help to trace the causes of failures inside the functional component. Another benefit of C^{2}FTs is a better structure of the system and all failure models, and this helps to keep the overview of the system and allows easy maintenance. As stated before, C^{2}FTs are only presented in a simplified form of those described in the original work. In the remainder of this thesis, CFTs and C^{2}FTs are both called CFTs as they are treated the same way in the prototype. In the following section, the safety analysis application I-SafE and its platform Enterprise Architect are presented which support modeling and analyzing CFTs and other failure models.

2.6 Tools for Safety Analysis

The prototype was not intended to be a modeling tool for CFTs, and the goals were to provide efficient storage and analysis for an external safety analysis tool. For this work, three safety analysis tools were available which supported CFT modeling and analysis. These are the Embedded Systems Safety and Reliability Analyzer (ESSaRel), Magic Draw with some additional plugins and the I-SafE application developed by Fraunhofer IESE that uses Enterprise Architect (EA) as a platform. While the ESSaRel provides the functionality for modeling and analyzing CFTs directly without any plugins, it is not used in practice anymore and will not receive any updates. The same issue applies to MagicDraw because newer versions of MagicDraw do not support the plugins needed for CFT modeling and analysis. Therefore, the I-SafE application with its platform EA became the tool of choice. All CFTs used in this work are created with I-SafE first, and then they are transferred to the prototype for efficient storage and analysis. In the following subsections, the key features of EA are highlighted, and the I-SafE application is introduced.

2.6.1 Enterprise Architect

Enterprise Architect[3] (EA) is a software modeling tool that is used widely in the industry. It supports many standards that are used in software development, and it covers most aspects of the software development cycle. The basis for EA is the UML 2.5 and other open standards such as BPMN 2.0 and SysML 1.3. It provides a visual platform to design and construct software systems and other processes that are required during software development [8]. Its defining features are built-in Requirements Management, Project Management with full Life-Cycle Modeling, Test Management, flexible documentation options, a multi-user environment for development teams and various other modeling techniques and standards [8]. A screenshot of the EA user interface can be seen in Figure 10.

[3] http://www.sparxsystems.com/products/ea/index.html (accessed: 15.03.2017)

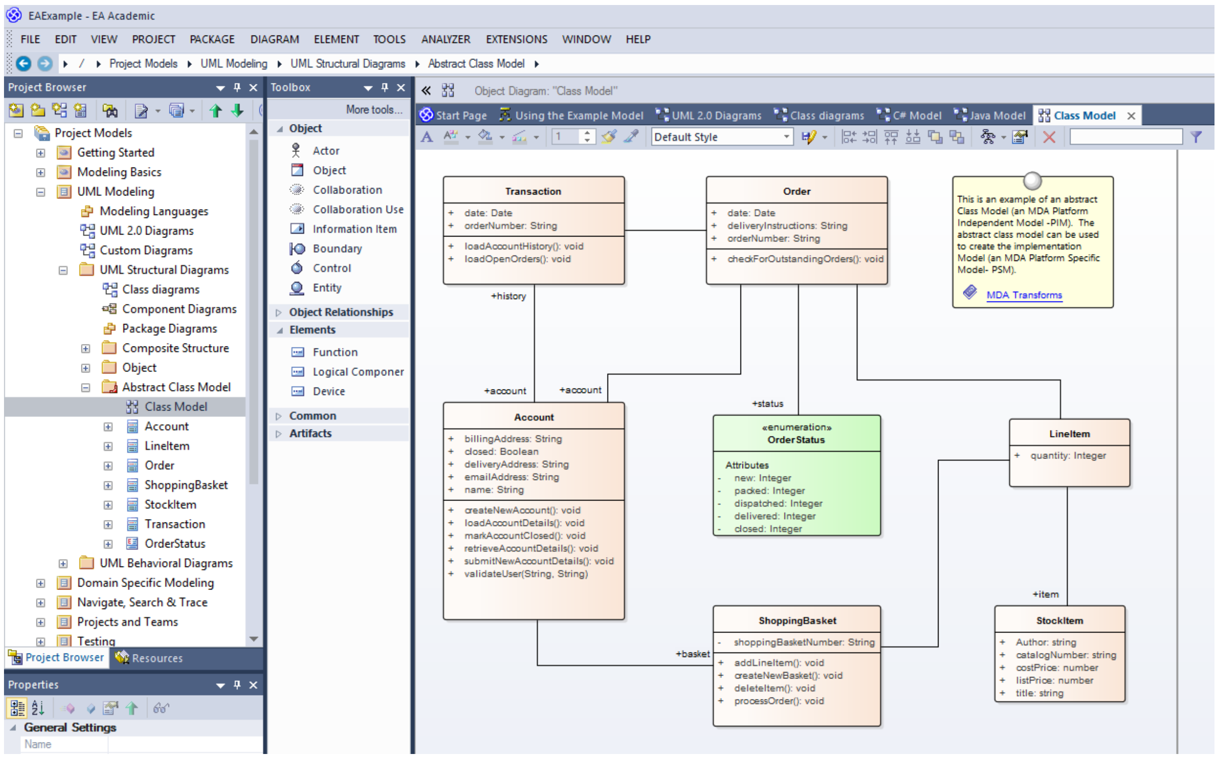


Figure 10 Enterprise Architect 12.1 UI

The Project Browser visible on the left side manages all packages, elements, and diagrams that are created. The toolbox and the diagram view are used to modify diagrams. Those are the important parts of the UI for this work. A summary of other features can be found in [8]. One of the most useful features of EA is the support of user-created add-ons or extensions that allow integrating other models and modeling techniques into EA. The Fraunhofer IESE wrote the application I-SafE which adds many safety modeling techniques to EA including CFTs and C²FTs. Another feature that made EA the optimal modeling tool for this work was the Java API that allows access to the EA repository.
Overall, Enterprise Architect has many features that make it a suitable platform for safety analysis and a design tool for failure models. The following section describes the important features of the I-SafE application.

2.6.2 IESE I-SafE Extension

The I-SafE application[4] was developed by the Fraunhofer IESE and is a new safety analysis tool with a wide variety of functions. The very popular EA serves as a platform, and the application makes use of its modeling capabilities. I-SafE implements the integration of CFTs, C²FTs, FMEA and Markov Chains into diagrams that are used in EA. The integration of CFTs into UML is described in [9]. Furthermore, I-SafE adds many new

[4] http://www.iese.fraunhofer.de/en/competencies/safety_engineering/tools_safety/isafe.html (accessed: 15.03.2017)

components which allow the attachment of failure models. I-SafE uses the principles that were explained in Section 2.5 for functional components and their failure models. With the next two examples, these concepts are visualized. Figure 11 shows a model of a component *Infusion Pump* in its realization. Inside, many subcomponents in their specification are visible, and there are many connections (dashed lines) that represent data flows. Inputs of a component are located on the left side, while outputs are located on the right side.

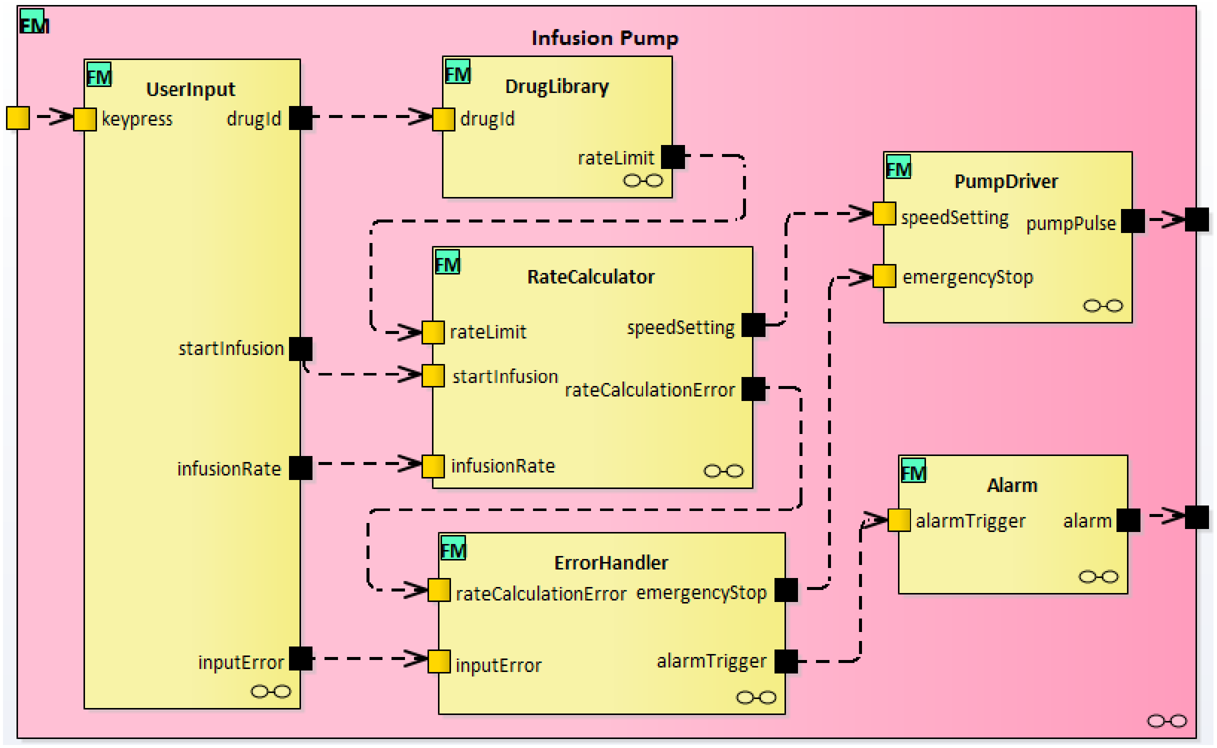


Figure 11 Logical Component Infusion Pump modeled with I-SafE

The component *Infusion Pump* has an input *keypress* located at the upper left-hand corner which is meant to be an input of commands by the user on a keypad. This input is used by the component *UserInput,* and it processes the input and returns data to other components via its outputs. The data flow continues through other components until it reaches the outputs of the *Infusion Pump*. In this example, two outputs are visible on the right side. The first output returns a pump pulse that could be given to a mechanical pump component in a higher-level abstraction and the second output returns an alarm signal if there is a problem in the component. In this model, all components also have a failure model attached to them which is visible by the small *FM* in the top left corner. The next figure shows a CFT failure model which belongs to the component *DrugLibrary*. This component is part of the *Infusion Pump* and visible in the top middle of Figure 11.

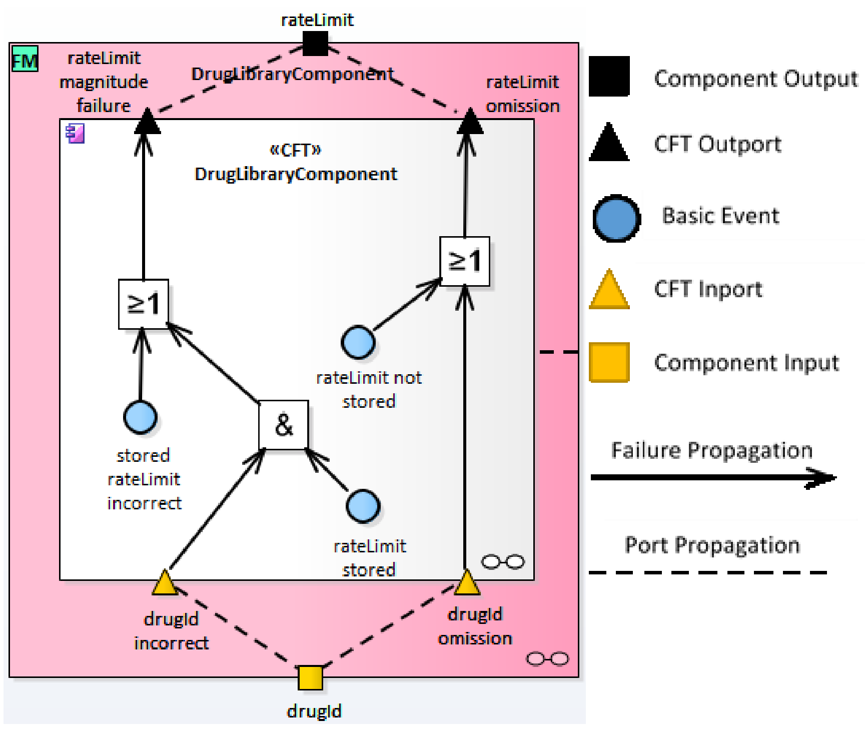


Figure 12 CFT of DrugLibraryComponent

The CFT of *DrugLibraryComponent* is displayed inside the functional component. Input and output of the functional component are connected to the inports and outports of the CFT. In I-SafE, CFTs can also contain other CFTs as instances, and Figure 13 shows an example of this.

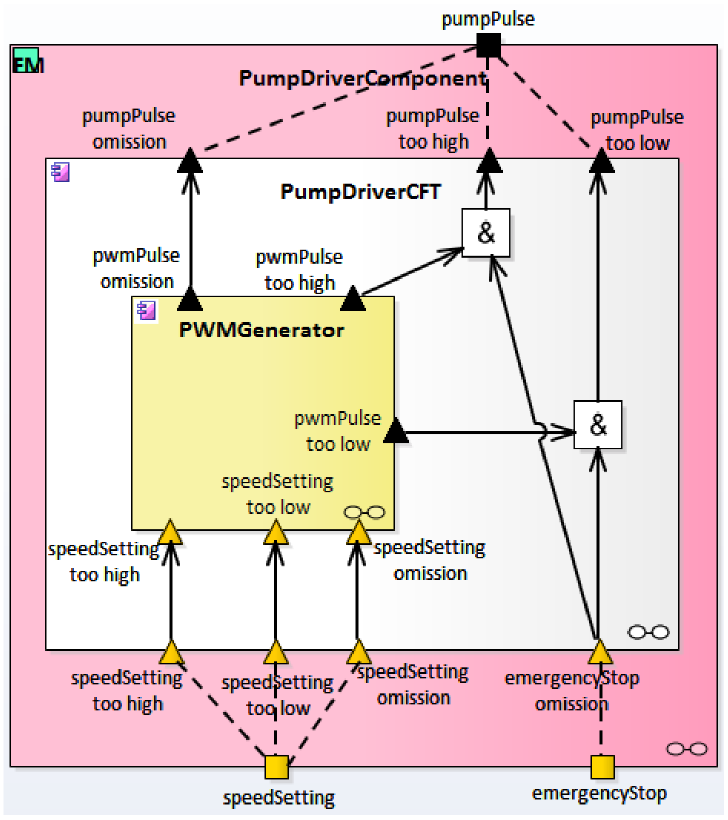


Figure 13 CFT of PumpDriverComponent

The CFT of *PumpDriverComponent* contains an internal CFT called *PWMGenerator* as an instance. Overall, the I-SafE application is easy to use, and the representation of the failure model is very clear. Therefore, I-SafE was the best safety analysis tool available for this work that could model CFTs. In all following sections, the term EA is used to describe Enterprise Architect with the I-SafE extension. The next section presents all important aspects of the graph database Neo4j.

3 Neo4j Graph Database

Most information in the world is connected, and there are huge networks on the internet that connect people and other things together. When data is stored in relational databases, connections are often saved as foreign keys in entities, to store the data efficiently and to boost access performance. The disadvantage is that access to heavily connected data requires many *table join*s which reduce performance. Therefore, it makes sense to consider other approaches to storing highly connected data. One such approach is a graph database which stores data as nodes and edges connected in a graph. The advantage of these graph structures is that they allow very fast data traversal between different entities and they also store data more intuitively [10].
The purpose of this work was to implement a prototype for safety analysis using a graph database backend. Therefore, some basics about graph databases need to be established. In the following subsections, the graph database Neo4j is introduced, and the main concepts are explained. Then, an introduction to the Neo4j query language Cypher is given. After that, the Neo4j Browser is presented, and lastly, the Neo4j language drivers that connect the Neo4j database to an application are described.

3.1 Basics of Neo4j

Neo4j[5] is an open source graph database developed by Neo Technology, Inc. It is implemented in Java and the most popular graph database according to the database ranking site db-engines.com[6]. The Neo4j Community Edition is licensed under GPLv3, and there also exists a commercial edition. The Neo4j developers describe Neo4j as an ACID-compliant transactional database with native graph storage and processing.
In Neo4j everything is stored as nodes and relationships. Nodes represent the entities and relationships are the edges or links between entities. Inserting nodes and connecting them with relationships creates the graph. Both nodes and relationships can have properties, such as name or identifier. These properties hold the information of a node or relationship. A node can also have one or more labels attached to it. These labels help to group nodes into sets, for example, to differentiate between a person node and a car node. Instead of labels, relationships have a relationship type which identifies their purpose. An example of a graph in Neo4j is shown in Figure 14.

[5] https://neo4j.com/ (accessed: 15.03.2017)
[6] http://db-engines.com (accessed: 15.03.2017)

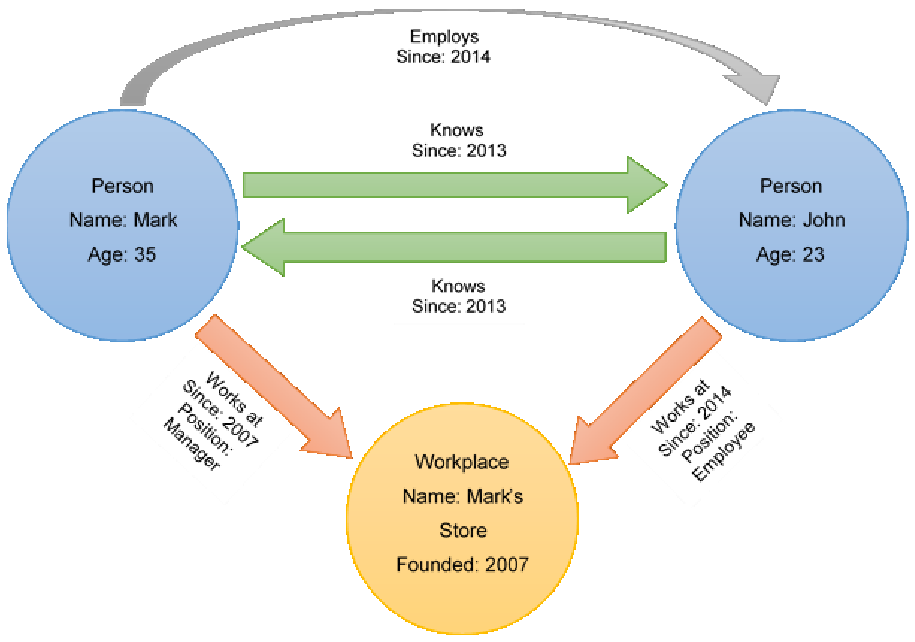


Figure 14 A simple graph database

The example has three nodes of which are visualized by filled circles. The relationships are visualized by thick arrows. The example represents a database that stores workplaces, persons and their relationships to each other. The two upper nodes are labeled *Person* and have the properties *Name* and *Age*. The bottom node is labeled *Workplace* and has the properties *Name* and *Founded*. The top relationship between the two *Person* nodes has the relationship type *Employs* and the property *Since*. The two middle relationships between the *Person* nodes have the relationship type *Knows* and the property *Since*. The relationships between the *Person* nodes and the *Workplace* node have the relationship type *Works at* and the properties *Since* and *Position*.

In Neo4j, there are some constraints on the creation of nodes and relationships. For nodes, all parameters are optional, and they can exist in the database with no labels, no properties, and no relationships. On the other hand, relationships do have some constraints. They must have exactly one relationship type which is the label of a relationship, and they also must have exactly one start node and one end node which give a direction to the relationship. However, the start and the end node can be the same node and relationships can be traversed in both directions. As relationships can also store properties, they are also used to represent entities. Nodes and relationships are all stored with an internal identifier, but these are reused in case of deletion so the developer should use an alternative identifier system. Nodes can only be deleted if all relationships that connect them are deleted first as a relationship must always have a start and an end node. Labels can be added and removed from a node while a relationship type cannot be changed. Otherwise, limitations are minimal and enormous graphs can be created and traversed.

Graph traversal is critical in Neo4j because nodes and their related nodes must be found and accessed quickly. Traversing also means that rules are defined to obtain only data that is relevant. Neo4j uses paths which are a chain of nodes that are connected with relationships to traverse the graph. There are several path algorithms implemented, for instance, one to find the shortest path, and they are useful depending on how data is stored in the database. Section 5 explains these graph traversals in more detail. There are two main uses of a Neo4j database. First as an embedded database using the Neo4j Java API and second as a client-server database using the Cypher Query Language (Cypher). In this thesis, an embedded Neo4j database is used which relies mostly on the Neo4j Java API. In the following, Cypher is briefly introduced because it is a major part of Neo4j and some Cypher commands are used in the prototype where they are more concise than using the API.

3.2 Cypher Graph Query Language

Cypher[7] is a declarative graph query language that provides expressive and efficient querying and updating of the graph database [11]. The structure and some keywords are inspired by the Structured Query Language (SQL) which is widely used in relational databases. Therefore, it is easy to pick up if the developer has previous experience with SQL. The general idea is to use patterns of nodes and relationships inside clauses to find these same patterns inside the graph database. The most important patterns, clauses, and functions are presented in the following subsections.

3.2.1 Cypher Patterns

Cypher is a pattern-based language, and this makes it very easy to understand and learn. Patterns are a very simple way to specify nodes, relationships or paths inside the Cypher query. The next few pattern samples show how nodes are specified in a query.

```
1  ()
2  (node)
3  (:Label)
4  (node:Label)
5  (node:Label {property: Value})
6  (node:Label {property1: Value, property2: Value})
7  (john:Person:Actor {Name: “John”, Age: 45, Married: true})
```

The first line shows a node with no further specification. The second line adds a variable which can be utilized again in the same or later clauses of the Cypher query. In the third line, there is no variable but a label that specifies that this node must have that label. The

[7] http://www.opencypher.org/ (accessed: 15.03.2017)

next line shows a variable and a label used together. The lines 5 and 6 demonstrate how one or more properties are added to the node pattern. The last line shows an example with more realistic labels and properties. Also, two labels are used instead of one. The next few patterns illustrate how relationships are described in Cypher.

```
1  -->
2  -[relationship]->
3  -[:Type]->
4  -[relationship:Type]->
5  -[relationship:Type {property: Value}]->
6  (movie:Movie)<-[role:ACTED_IN {Roles: [“Farmer”, “Business Man”]}]-(john:Actor)
```

The first line shows a generic relationship is with no further specification. This first pattern can also be used with no direction. The next line shows a relationship with a variable and line three shows a relationship with a type. Line four combines variable and type, and the fifth line demonstrates how properties are added to the relationship pattern. A relationship pattern must always have a node on each side, and the direction arrow can be moved to the other side. Line six shows an example of a full relationship pattern with a reversed direction. Also, this example shows that properties can be a list of values. Overall, patterns play a major role in Cypher statements, and they can be very long depending on what data is requested. The Cypher clauses which include these patterns are described in the next section.

3.2.2 Cypher Clauses

There are various clauses to find, add, change, remove and return data with Cypher. The “MATCH” clause is used to match a pattern in the database, and the “RETURN” clause specifies how the data is returned. The following example shows the use of both clauses.

```
MATCH (movie:Movie)<-[role:ACTED_IN]-(john:Actor {Name: “John”, Age: 45})
RETURN movie.Title, collect(role.Roles)
```

This Cypher query searches the graph for movies in which the actor *John* has played a role and returns each movie title with a list of all roles that were played by *John* in that particular movie. This pattern is visible in the “MATCH” clause. In the “RETURN” clause, the Cypher function `collect()` is used to collect all roles that were played by *John* in a list for each movie title. If `collect()` is not used, then multiple rows with the same movie title for each role are returned.

The “WHERE” clause is like the “WHERE” clause in SQL, and it allows to specify certain properties the data must have. Patterns can still be used in a “WHERE” clause. The

“CREATE” clause is used to add nodes and relationships to the database. A query with multiple clauses used together can be seen in the next Cypher example.

```
MATCH (a:Person), (b:Person)
WHERE a.Name = “John” AND b.Name = “Mark” AND NOT (b.Age < 20)
CREATE (a)-[r:KNOWS]->(b)
RETURN r
```

This Cypher query searches for two nodes with the label *Person*, and the “WHERE” clause specifies that the nodes must have the property *Name* and that one node must have the name *John* and the other must have the name *Mark*. Also, the node *b* must have the property *Age* which must be smaller than 20. If such two nodes are found the “CREATE” clause creates a relationship between them and the “RETURN” clause returns all created relationships. In this case, “RETURN” returns rows with relationship data but also an object for a relationship can be returned. This object also exists for nodes and these objects are most prevalently used for visualization in the Neo4j Browser which is described in Section 3.3.
Another important clause is “MERGE” which is a combination of “MATCH” and “CREATE”. If the pattern after a “MERGE” already exists, the clause simply matches the data and otherwise, the pattern is created. In the previous Cypher query, the “CREATE” could be replaced by “MERGE”, but this would change the query. With “CREATE”, the described relationship is created even if it already exists which is prevented if “MERGE” is used. The clauses “SET” and “REMOVE” are used to add, change or remove node labels and properties from nodes and relationships. Relationship types cannot be modified. “DELETE” is a clause that is used to delete nodes and relationships, but nodes can only be removed once all their relationships have been deleted. An example of “DELETE” is shown below.

```
MATCH (n)
DETACH DELETE n
```

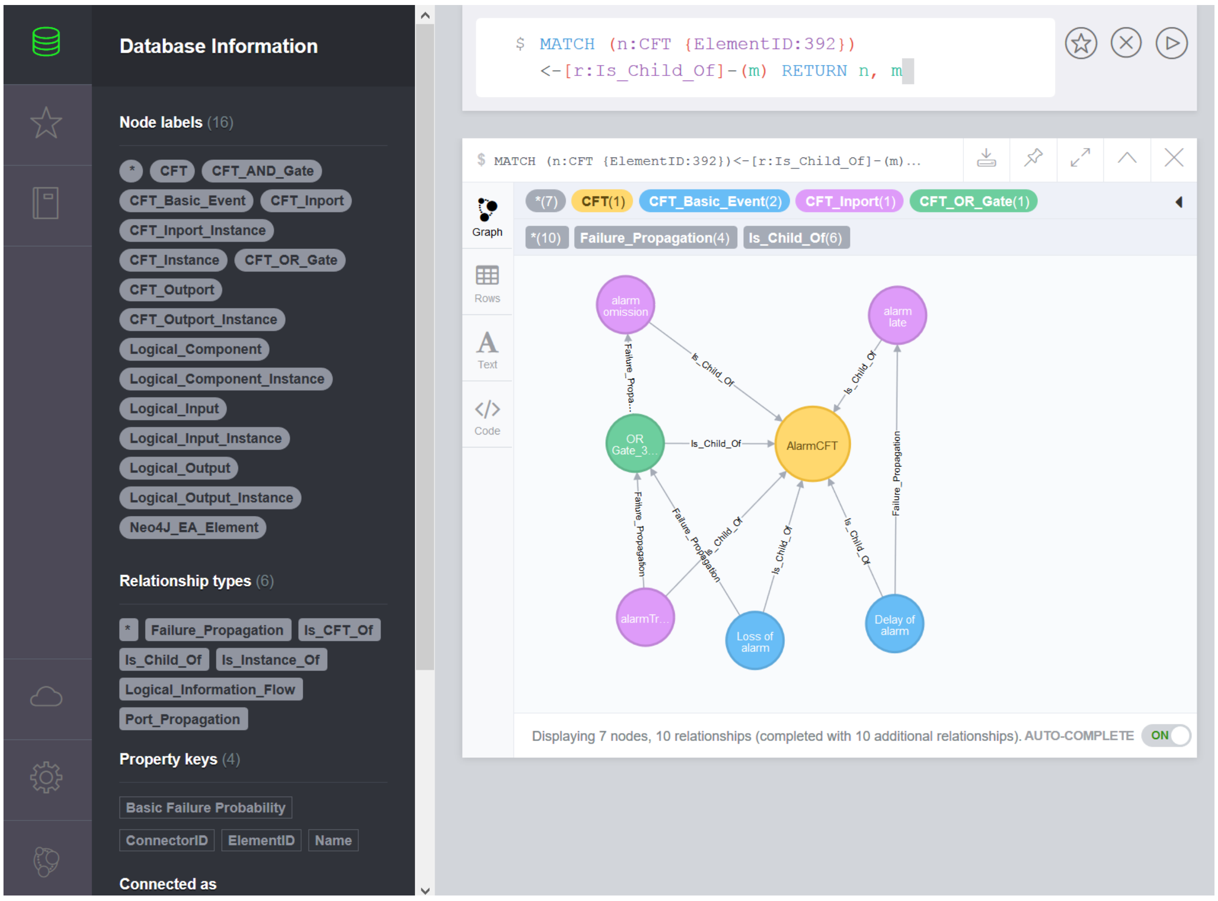
In this example, “DELETE” is used together with “DETACH”. Both clauses delete all relationships of the node that is matched, and after that, the node is removed. This Cypher statement clears the entire database since every node matches that pattern. Cypher offers many more clauses, but the presented clauses serve as a sufficient introduction to Cypher and represent examples of clauses that appear in the implementation of the prototype.
A very useful document for working with Cypher is the Cypher Refcard[8] which explains all relevant Cypher commands and provides small examples. Another helpful tool to

[8] http://neo4j.com/docs/cypher-refcard/current/ (accessed: 15.03.2017)

understand Neo4j and Cypher is the Neo4j Browser which is presented shortly in the next section.

3.3 Neo4j Browser

As Neo4j is a graph database, it is possible to represent the data model visually. This visualization helps to understand the structure of the data that is stored in Neo4j. The Neo4j Browser is part of the Neo4j Community Edition (Neo4j CE) and a simple UI, integrated into a browser, to interact with the database. It provides several useful features including a Cypher command line, graph visualization, and interactive guides. In Figure 15, a screenshot of the Neo4j Browser is displayed.



```
$ MATCH (n:CFT {ElementID:392})
  <-[r:Is_Child_Of]-(m) RETURN n, m
```

```
$ MATCH (n:CFT {ElementID:392})<-[r:Is_Child_Of]-(m)...
```

Displaying 7 nodes, 10 relationships (completed with 10 additional relationships). AUTO-COMPLETE ON

Figure 15 Neo4j Browser

The Neo4j Browser is a simple tool provided by Neo4j to access the database and interact with it. On the top, a command line is visible that allows running Cypher queries and other helpful commands to show tutorials and guides. The sidebar offers options such as database information which shows node labels, relationship types, and property keys. Also, there are options to customize how nodes and relationships look in the graph visualization. A Graph Style Sheet can be uploaded for more individual styles, and the

Neo4j Browser also allows some additional database configurations. In this work, the Neo4j Browser was used very often to fix problems and get an overview of the database. It allows running quick Cypher queries for testing and deleting certain nodes and relationships. Overall, the Neo4j Browser is a very useful tool when working with Neo4j.

3.4 Neo4j Java Driver

Neo4j relies on database drivers that connect applications to the Neo4j database and interact with it. In most large-scale use cases the Neo4j database runs on a server, and the clients can interact with it through the Neo4j driver. Neo4j database drivers are available for many programming languages, and this section introduces the official Neo4j Java driver. The prototype does not use any database drivers, but a small introduction is given because it is a core part of Neo4j and used for most other applications.

The official Neo4j Java Driver provides access to a Neo4j database for applications. It uses the bolt protocol which is a client-server protocol developed by Neo4j developers. The Neo4j Java driver is used to request a connection to the database. With that database connection, a session can be started, and transactions can be run in the session. In a transaction, Cypher is used to query the Neo4j database, and it returns a result which can be processed. A result is a stream of records. The record is an ordered mapping of string keys to records. The use of the Neo4j Java driver is shown in the example below.

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687", AuthTokens.basic(
"neo4j", "neo4j"));
try (Session session = driver.session())
{
  try (Transaction tx = session.beginTransaction())
  {
    tx.run("CREATE (a:Person {name: {name}, job: {job}})",
      parameters("name", "John", "job", "Actor"));
    tx.success();
  }
  try (Transaction tx = session.beginTransaction())
  {
    StatementResult result =
      tx.run("MATCH (n:Person) WHERE n.name = {name}
        RETURN n.name AS name, n.job AS job",
        parameters("name", "John"));
    while (result.hasNext())
    {
      Record record = result.next();
      System.out.println(String.format("%s %s", record.get("job").asString(),
        record.get("name").asString()));
    }
    tx.success();
  }
}
driver.close();
```

First, the driver is created from a running Neo4j database server. Then from the driver which represents the connection to the database, a session is started. In this example, two transactions are run in the session. The first transaction creates a node with the properties "name" and "job" which are set in the parameters to "John" and "Actor". After that, the first transaction is finished and the second transaction is started. The second transaction now matches all nodes where the property "name" is "John" and returns the properties "name" and "job" for each matched node. The transaction then prints every record in the result on the console. After that, the transaction is finished, and the driver is closed. A driver allows multiple sessions, and there are specific sessions for write and read operations.

The Neo4j Java driver relies on a running Neo4j database server and also exclusively uses Cypher to query the database. The prototype of this work is designed as an embedded application and needs high performance to show practical benefits. The Neo4j Java driver is not suited for such a prototype because the Cypher queries are often slower than direct database access over the Java API. Additionally, the overhead of running a Neo4j database server besides the prototype is very high, and communication is many times slower than direct access to a database stored on the hard drive. Therefore, the prototype uses the Neo4j Java API for database access, and this API is introduced in the following section.

3.5 Neo4j Java API

The Neo4j Java API[9], also called Neo4j embedded[10], is the most efficient way for embedded applications to access a Neo4j database. It interacts with a locally stored Neo4j database, and it is meant for direct integration into an application. The database access is much faster because there is no communication with a server and there are many interfaces and methods that are available to retrieve and store data without the need of Cypher. Cypher can still be used and is sometimes more concise than using direct API methods. The prototype uses the core graph database API of Neo4j and the Neo4j Traversal Framework. A Neo4j database is started by creating a *GraphDatabaseService* which is the main link to the embedded database and needed for all transactions.

The most important API interfaces for general use of the API are *GraphDatabaseService*, *Node*, *Relationship*, *Label*, *RelationshipType*, and *Transaction*. The interface *GraphDatabaseService* provides methods to find and create nodes and access some general properties and objects in the database. Cypher queries can be executed with the *GraphDatabaseService*. The interface *Transaction* is started with the

[9] https://neo4j.com/docs/java-reference/current/javadocs/ (accessed: 15.03.2017)

[10] http://neo4j.com/docs/java-reference/current/#tutorials-java-embedded (accessed: 15.03.2017)

GraphDatabaseService and is used whenever data from the database is accessed and must always be finished so it can be closed. Nested transactions are possible. The interface *Node* is the representation of a node stored in the database, but it does not hold any properties in the object. Any access to specific information of the *Node*, like relationships or properties, are retrieved from the database using a transaction. The interface *Relationship* is the object representing relationships stored in the database and works like the *Node*. The interfaces *Label* and *RelationshipType* represent labels and relationship types. These interfaces and their methods provide the core operations that are performed on the database. A code example of using the Neo4j API is shown in the following.

```
1  String dbfilepath = “C:/Neo4jDatabaseFolders/Neo4JDatabase1”;
2  GraphDatabaseService graphDb;
3  graphDb = new GraphDatabaseFactory().newEmbeddedDatabase(dbfilepath);
4  try (Transaction tx = graphDb.beginTx()) {
5    Label person = Label.label("Person");
6    Node node1 = graphDb.createNode(person);
7    node1.setProperty(“Name”, “John”);
8    Node node2 = graphDb.createNode(person);
9    node2.setProperty(“Name”, “Mark”);
10   RelationshipType knows = RelationshipType.withName(“Knows”);
11   Relationship relship = node1.createRelationshipTo(node2, knows);
12   relship.setProperty("Since", 2013);
13   System.out.println(node1.getProperty(“Name”) + “ is aquainted with ” +
14    node2.getProperty(“Name”) + “ since ” + relship.getProperty(“Since”));
15   tx.success();
16 }
17 graphDb.shutdown();
```

First, a *GraphDatabaseService* is started using the provided file path to the location of the database. A database can only be started if no other services are using it. All interactions with the Neo4j database, such as reading or writing data, require a transaction which is started on line 4 and ends at line 15 of the code example. Inside this transaction, the label “Person” is created and applied to the two nodes that are created afterward. The property “Name” added to both nodes and next, the relationship type “Knows” for the following relationship is created. The relationship is created between the two nodes, and the property “Since” is applied to the relationship. A string containing information about the nodes and the relationship is printed onto the console. After the transaction is successfully finished, the *GraphDatabaseService* is shut down.
Besides the core functionality of the Neo4j Java API, the Neo4j Traversal Framework provides many interfaces and methods that help to traverse the graph and find valuable information quickly. This is one of the major benefits to the regular Neo4j drivers which only provide Cypher queries. It is difficult to traverse the graph continuously with Cypher because the language does not provide the constructs necessary for loops. Therefore, to

traverse the graph, many Cypher queries are needed which reduces performance significantly. The Neo4j Traversal Framework is explained in Section 5.

This section concludes the introduction to all important aspects of Neo4j, and more importantly, with the end of this section, all prerequisites for the following sections of this thesis have been explained. In the following, the shorter term database is used as a synonym for the Neo4j graph database for the sake of brevity. In Sections 4 and 5, the focus is the implementation of the prototype and Section 6 presents the results and gives an evaluation of the prototype.

4 Importing Safety Models to the Prototype

The prototype that was developed during this work performs analysis on component fault trees (CFTs) which were introduced in Section 2.4. Since the prototype was not developed to be a modeling tool for CFTs it relies on the CFT models created in EA/I-SafE which was introduced in Section 2.6. The CFTs are imported from EA and then analyzed by the prototype. The Neo4j graph data model which defines how data is stored in the database is designed to ensure a structured database that could be used for efficient traversal and analysis. The prototype can perform a continuous analysis which is the reuse of previous analysis results to calculate new analysis results. The continuous analysis is performed as soon as CFTs are modified, so the analysis results are always available and up-to-date. Most tools for FTA perform a conventional static analysis which starts the analysis from scratch every time. An EA extension was developed to provide continuous updates for the prototype and show the benefits of continuous analysis compared to conventional static analysis.

The following subsections describe all aspects of the prototype that were implemented before the analysis of CFTs was possible. In Section 4.1, the ideas and the setup of the prototype are explained, and in Section 4.2 the graph database model for Neo4j that was used for the implementation is described. Then, in Section 4.3, there is an overview of how the prototype transfers data from EA to Neo4j with an example, and it shows how the GUI of the prototype is used to create and update the Neo4j database. Section 4.4 concludes the chapter by describing the implementation of an EA extension that was used for continuous analysis in the prototype.

4.1 Idea and Setup

The main idea of this work is to use the graph database Neo4j for storing and analyzing CFTs. At first, it was considered if a Neo4j driver should be used for the connection to the database. However, early tests showed that the performance of Neo4j drivers was less than satisfactory, so the prototype was implemented using the Neo4j Java API. The prototype is split into a standalone application and an application that works together with an EA extension. Both applications of the prototype are implemented in Java and are implemented in a common project. The standalone application uses the Java API from EA and Neo4j to implement the data transfer and analysis. The analysis of the standalone application is static so updating the Neo4j database and analysis are separate tasks and not continuous. The developed EA extension is implemented in C# and communicates with the prototype. Therefore, the prototype can receive updates continuously and perform a continuous analysis that reuses previous results. Neo4j version 3.1.1 was used for the prototype. The prototype and EA extension were implemented for Windows and require EA 12.1 and I-SafE 1.0.0.0 to be installed. Newer versions of I-SafE may require changes to the prototype to work properly. The prototype uses Java 8 in the 32-bit version to ensure compatibility across different systems. The standalone application is portable and only requires EA to be installed. The EA extension must be installed before it can be used. The following sections describe the implementation of data transfer within the standalone application. The data transfer of the EA extension is described in Section 4.4.

4.2 Neo4j Graph Data Model

A Neo4j graph data model is needed to store information in a structured graph that can be traversed easily. In EA, the elements and connectors that are visible in diagrams and in the project browser are stored in a relational database. Since EA is used as the only source for CFTs, the Neo4j graph data model is kept very similar to the EA relational data model. Therefore, the diagrams in EA, as well as the elements in the EA project browser, are the main reference for the graph data model. Therefore, a graph visualization shows a graph that looks very similar to the corresponding EA diagram if the nodes and relationships are ordered correctly. A disadvantage of this approach is that any changes in the EA relational data model require changes to the prototype.
The default identifier system in Neo4j is not suitable for identifying nodes because it reuses identifiers from deleted nodes. Therefore, instead of creating a new system, the identifier system from EA is utilized in the Neo4j graph data model because in EA all elements and connectors have a unique identifier. Also, unique identifiers allow comparison of nodes and relationships in Neo4j to corresponding EA elements and connectors. The comparison is necessary to update the Neo4j database efficiently. Other information from EA elements and connectors is also needed for correct comparison because they might affect the analysis. The relevant properties of an EA element are identifier, name, stereotype, parent identifier, classifier identifier and all EA connectors that connect the EA element to other EA elements. The first two properties of an EA element are stored as the properties "ElementID" and "Name" in the corresponding node. The stereotype is stored as the label of the node, and the parent and classifier identifiers are stored as relationships. The connectors of each element are all stored as relationships. For an EA connector, the relevant properties are identifier, stereotype, and identifier of the origin element and the destination element. The identifier is stored as the property "ConnectorID" in the relationship. The stereotype is stored as a relationship type, and the identifiers for origin and destination element are implicitly stored in the relationship as start and end node.

In EA, elements are often part of other elements, or they belong to a certain element. For instance, a basic event belongs to a CFT. The parent identifier shows the parent element of a child element. This identifier is considered as a foreign key, and in Neo4j this is modeled as a relationship. Therefore, the parent identifier becomes a relationship in the graph data model. Like parent identifiers, EA also stores a classifier identifier which shows the classifier of an instance. Element instances are created by EA when an element that consists of other elements is used inside another element. For example, EA creates an instance of a CFT when it is used inside another CFT. Therefore, a CFT only contains CFT instances, not the actual CFT elements. An instance element stores the classifier identifier which is the identifier of the classifier element. In Neo4j, the classifier identifier like the parent identifier is stored as a relationship between instance element and classifier element.
If an element in EA has no parent, then the parent identifier is 0, and if an element is not an instance, then the classifier identifier is 0. Before continuing to describe the model, a small example is shown in Figure 16 and 17.

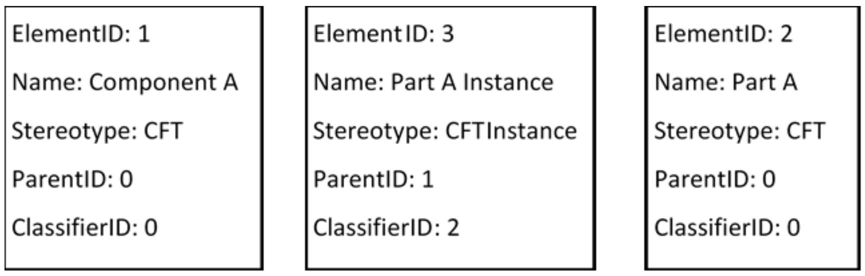


Figure 16 Three EA Elements

In this example, three EA elements are visible, and the element in the middle has a classifier identifier and a parent identifier. Therefore, it is an instance of the right element and is part of the CFT which is the left element. In Neo4j, this model is transformed into the graph depicted in Figure 17.

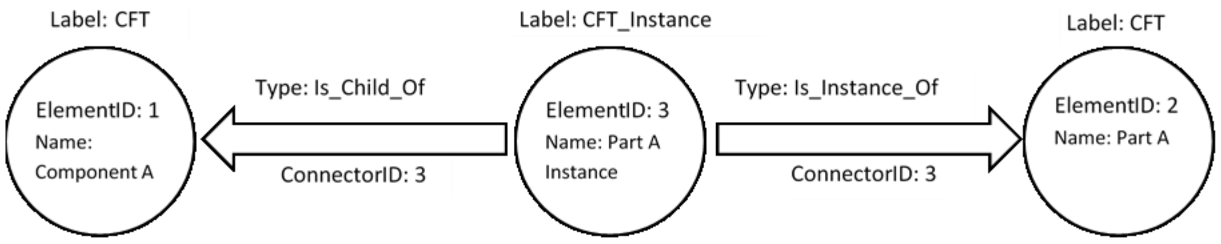


Figure 17 Three EA Elements in Neo4j

This example shows how the parent identifier and the classifier identifier are stored as relationships. It is also visible that the stereotype is used to create the label of a node.
This work is focused on the analysis of CFTs, and the number of stereotypes that are related to CFTs in EA is limited. Therefore, the Neo4j graph data model stores only relevant EA stereotypes as labels. The label that corresponds to each stereotype is slightly

modified to increase understandability. Tables 1 and 2 provide a description of each label as well as the corresponding EA stereotype. The EA stereotype IESELogicalComponent represents logical elements which can have a failure model such as a CFT attached to it.

Table 1 Neo4j Labels

Label *(EA Stereotype)*	**Description**
Neo4J_EA_Element *(all EA Elements)*	This label is added to all nodes that are created from EA elements.
CFT *(CFT, FT)*	A CFT / FT.
CFT_Instance *(CFTInstance, FTInstance)*	An instance of a CFT / FT.
CFT_Inport *(InputFailureMode)*	An inport of a CFT.
CFT_Inport_Instance *(InputFailureMode)*	An instance of a CFT inport and an inport of a CFT instance.
CFT_Outport *(OutputFailureMode)*	An outport / top event of a CFT / FT.
CFT_Outport_Instance *(OutputFailureMode)*	An instance of a CFT outport and an outport of a CFT instance.
Logical_Component *(IESELogicalComponent)*	A logical component.
Logical_Component_Instance *(IESELogicalComponentInstance)*	An instance of a logical component.
Logical_Input *(IESELogicalInport)*	An input of a logical component.
Logical_Input_Instance *(IESELogicalInportInstance)*	An instance of a logical component input and an input of a logical component instance.
Logical_Output *(IESELogicalOutport)*	An output of a logical component.
Logical_Output_Instance *(IESELogicalOutportInstance)*	An instance of a logical component output and an input of a logical component instance.

Table 2 Neo4j Labels

Label *(EA Stereotype)*	**Description**
CFT_AND_Gate *(FTAND)*	An AND gate used in CFTs / FTs.
CFT_OR_Gate *(FTOR)*	An OR gate used in CFTs / FTs.
CFT_MOON_Gate *(FTM/N)*	A Combination (M out of N) gate used in CFTs / FTs.
CFT_XOR_Gate *(FTXOR)*	A XOR gate used in CFTs / FTs.
CFT_Basic_Event *(FTBasicEvent)*	A basic event used in CFTs / FTs.
CFT_NOT_Gate *(FTNOT)*	A NOT gate used in CFTs / FTs.

The stereotypes of connectors are stored as relationship types, and some relationship types are needed for relationships that are derived from parent identifiers and classifier identifiers. Table 3 shows the relationship types of the Neo4j graph data model with a description.

Table 3 Neo4j Relationship Types

Relationship Type *(EA Stereotype)*	**Description**
Is_Child_Of *(created from parent identifier)*	A relationship from a child element to the parent element.
Is_Instance_Of *(created from classifier identifier)*	A relationship from an instance element to the classifier element.
Failure_Propagation *(FailurePropagation)*	A failure propagation between elements in a CFT / FT.
Port_Propagation *(PortFailureModeTrace)*	A port propagation between a CFT inport/outport and logical component input/output.
Logical_Information_Flow *(Logical Information Flow)*	A logical information flow between logical components.
Is_CFT_Of *(ComponentFailureModelTrace)*	A relationship from a CFT to a logical component.

The analysis of CFTs requires additional properties for the combination gate, basic events, and inports. Both CFT inports and basic events have many parameters that can be used for analysis. The prototype only uses the basic failure probability for a simple quantitative

analysis. In EA, this property is saved as the tagged value “cValue” for each CFT inport and basic event. In the Neo4j database, this property is stored as “Basic Failure Probability”.
A combination gate is an “m” out of “n” gate which means the “m” and the “n” must be known to the prototype for analysis. The “n” is derived from the number of incoming failure propagations to the combination gate, and in EA the combination gate has a property “m” which is stored as a tagged value of the specific element. This “m” describes how many lower events must fail (evaluate to true) before the combination gate evaluates to true. In the data transfer, this property is saved for each combination gate as a property called “MOONNumber”. If the “m” is 0 or smaller the analysis considers the combination gate to always evaluate to true and if the “m” is larger than the “n”, then it considers the combination gate to always evaluate to false.
The prototype uses more labels for internal applications and analysis results, but they are not part of the data transfer and are described in Section 5. Overall, the graph data model used for the prototype is designed to be very similar to the relational data model of EA. The labels and relationship types are important because they enable fast traversal between connected nodes. The following section presents how the data transfer from EA to Neo4j is realized.

4.3 Importing Safety Models from Enterprise Architect

This section serves as an introduction to the following sections and explains the process of transferring data from EA to Neo4j. The first subsection explains the update process in broader terms without too much emphasis on implementation details and shows how a CFT from EA is visually represented in the Neo4j database. The second subsection explains how the GUI of the prototype is used to select and update the Neo4j database.

4.3.1 Relevant Data and Visual Representation

The prototype needs an EA repository to update the Neo4j database. Because of the limited number of stereotypes that are imported into Neo4j, only diagram elements from EA that are displayed in Figure 18 are transferred to the Neo4j database. Other elements are not imported into the database because they are not relevant to the scope of this work.

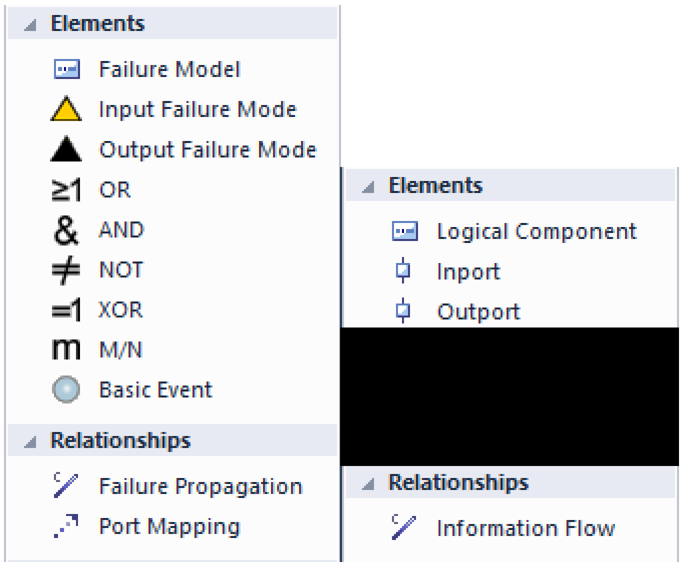


Figure 18 Allowed diagram elements in EA

When the prototype accesses the EA repository, the elements and their connectors are loaded into the prototype. In the Neo4j database, nodes represent EA elements and relationships represent EA connectors. The update process starts by updating the node of each EA element. During this process, relationships are created from the element and its connectors, but they are not added to the database until every element was updated in the database. The reason for this is that relationships can only be added when both the start and the end node exist for that relationship. The prototype can always check if a node exists, but during an update, there is the potential that nodes are replaced or deleted so the relationship cannot be created until all nodes are updated.
If the EA element has no corresponding node in the Neo4j database, then a new node is created for that element, and all necessary properties are added. If a corresponding node exists then the properties of the node are compared and modified if necessary. After all EA elements were updated, the prototype deletes any nodes that are EA elements but are no longer present in the EA repository. Before these nodes are deleted all their relationships are deleted. The last step of the Neo4j database update is to update all relationships which were created during the update of EA elements. Relationships are added between two nodes if they do not exist. They are deleted if they should not exist and replaced if the relationship changed.
When the update is completed, the Neo4j database contains all elements and connectors from EA that are necessary for the analysis of CFTs. In Figure 19 and 20, there is an example of how a CFT looks in Neo4j compared to EA. The visualization of the Neo4j Browser was used, but it was redrawn to highlight the details.

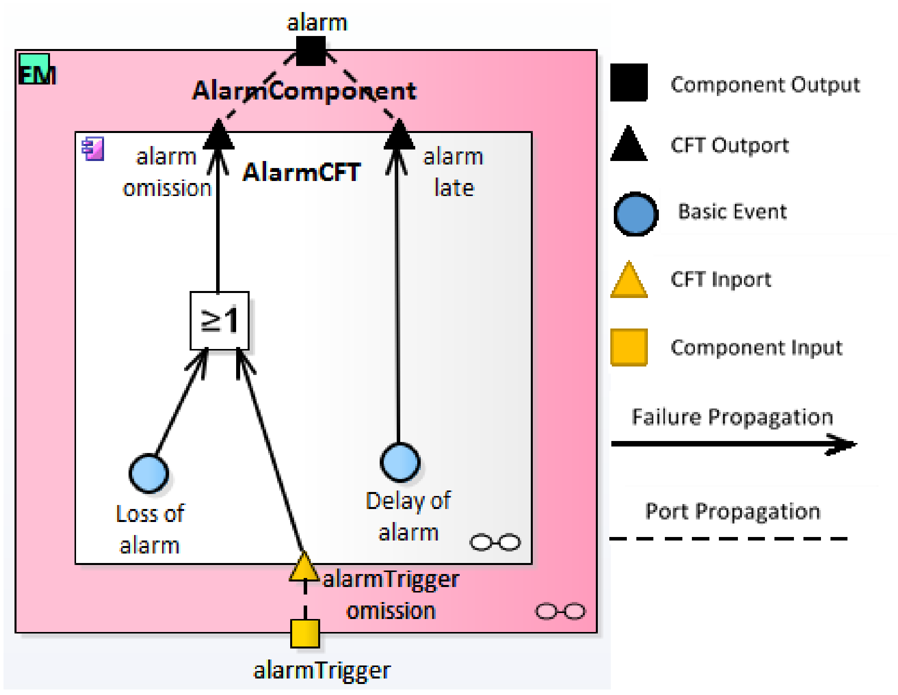


Figure 19 Alarm CFT in EA

The first figure shows the CFT of an *Alarm* component modeled in EA with two basic events (*Loss of alarm*, *Delay of alarm*), one CFT inport (*alarmTrigger omission*), two CFT outports (*alarm omission*, *alarm late*), and an OR gate. The *Alarm* component surrounding the CFT has one input (*alarmTrigger*) and one output (*alarm*).

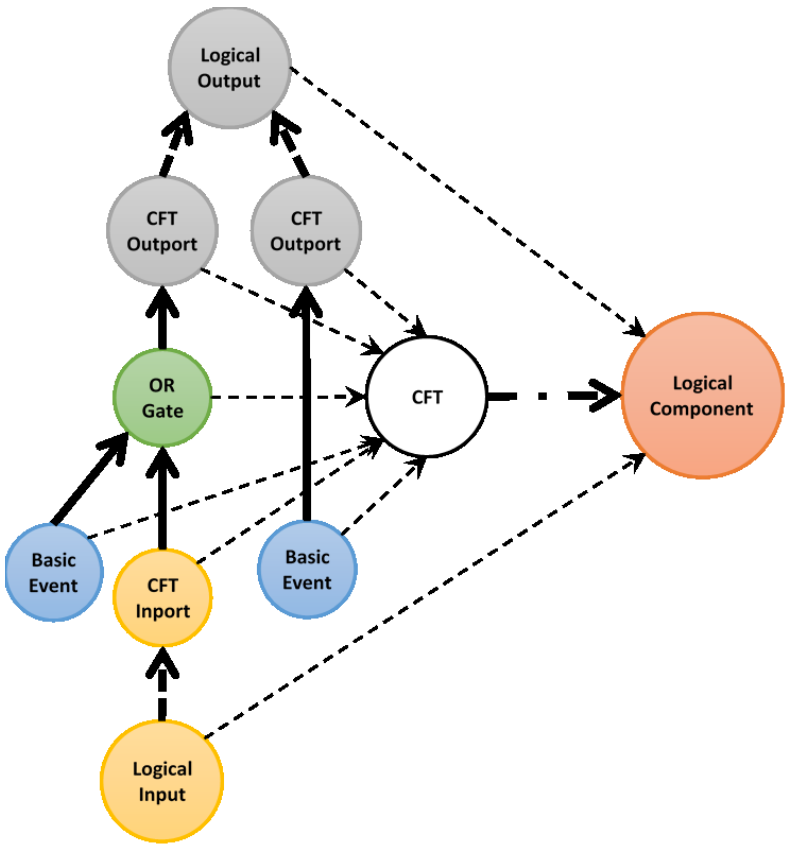


Figure 20 Alarm CFT in Neo4j

In the second illustration, the CFT has been transferred to the Neo4j database. The nodes are ordered to represent the EA diagram from the first figure. The relationships drawn with a thicker line (solid and dashed) were directly created from EA connectors, and the thinner dashed lines represent relationships that are created from the parent identifier in EA. Failure propagations have thick solid lines, and the port propagations have a thick dashed line. The thick dashed and dotted line between CFT and Logical Component represents two relationships. One is the *Is_Child_Of* relationship, and the other is the *Is_CFT_Of* relationship.

4.3.2 Using the Prototype to import Safety Models

The prototype has a graphical user interface (GUI) that provides the functionality to open a Neo4j database and also update the database with an EA repository. The relevant parts of the GUI for this specific section are depicted in Figure 21. The GUI shows the standalone prototype after start up with no Neo4j database selected. The text box with the label “Neo4j Database Status” indicates that the Neo4j database is not ready.

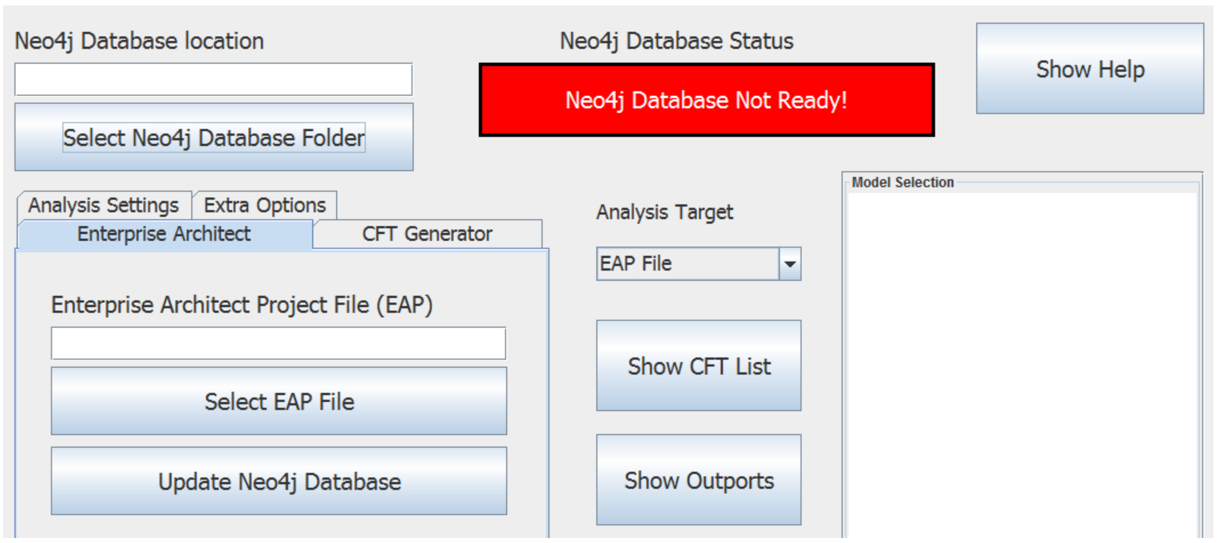


Figure 21 Neo4j database not ready in the GUI

After a location for a Neo4j database was selected with the button “Select Neo4j Database Folder” or by manually typing in a folder path and confirming with the enter key, the prototype starts a Neo4j database at that location. If the Neo4j database already exists in the selected location, the prototype opens it. Otherwise, the prototype creates a Neo4j database at the selected location. In the case that the Neo4j database did not exist or does not contain any CFTs for analysis the GUI shows that the database is ready for an update. This is depicted in Figure 22.

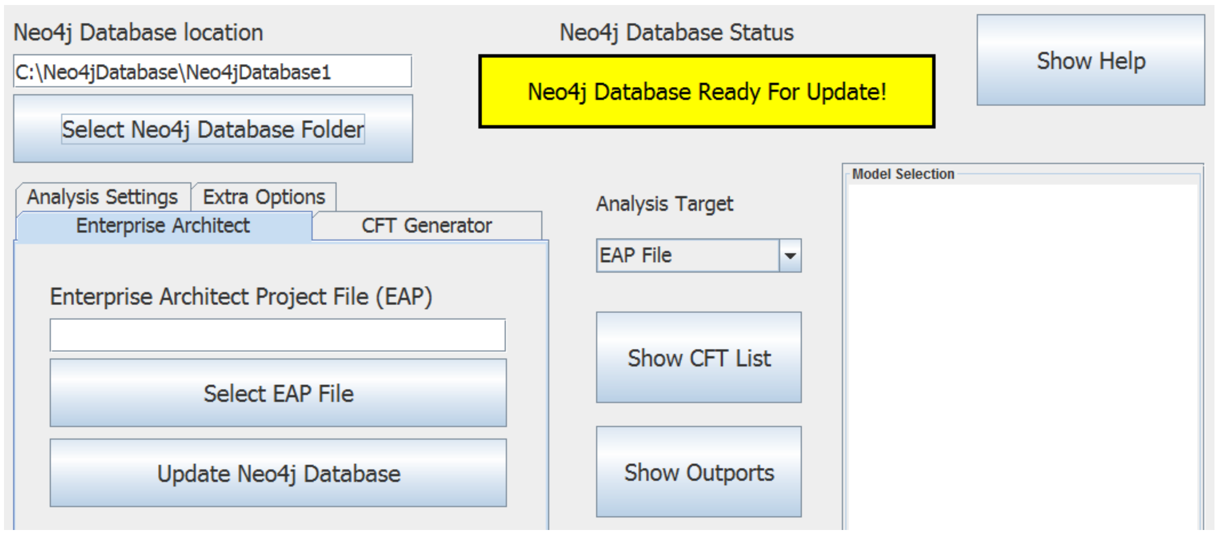

Neo4j Database location
C:\Neo4jDatabase\Neo4jDatabase1
Neo4j Database Status
Neo4j Database Ready For Update!
Analysis Target
EAP File
Enterprise Architect Project File (EAP)

Figure 22 Neo4j database ready for update in the GUI

The update is performed by selecting an EA project file and using the button "Update Neo4j Database". Then, the Neo4j database is updated with all relevant data from the EA project file, and after that, it is ready for analysis which is shown in Figure 23. An update of the Neo4j database can be performed after the EA repository was modified. Then, the prototype only updates elements and connectors that were modified.

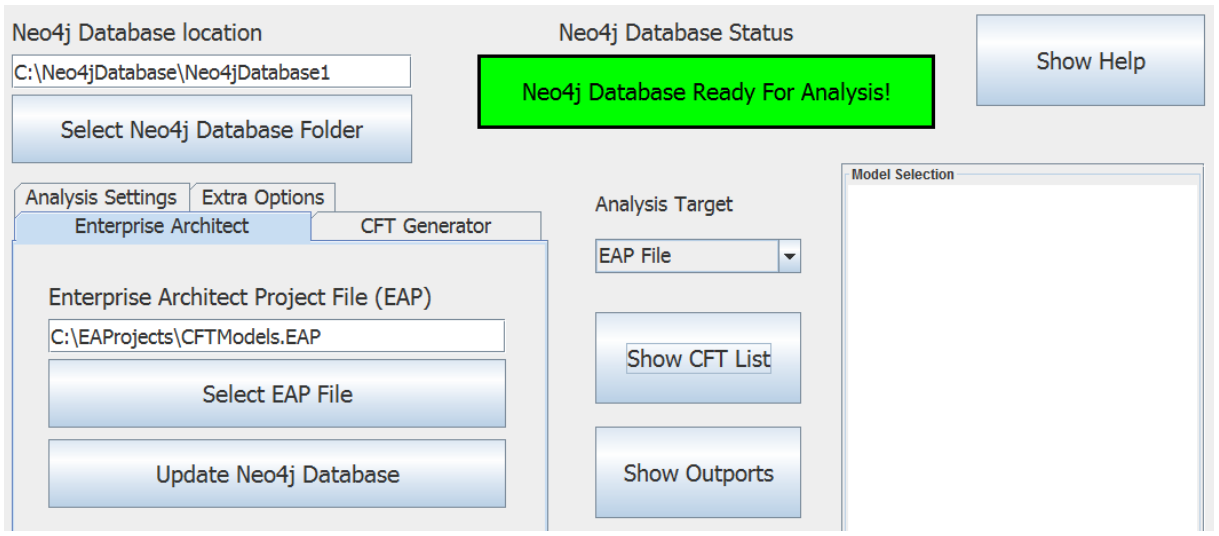

Neo4j Database location
C:\Neo4jDatabase\Neo4jDatabase1
Neo4j Database Status
Neo4j Database Ready For Analysis!
Analysis Target
EAP File
Enterprise Architect Project File (EAP)
C:\EAProjects\CFTModels.EAP

Figure 23 Neo4j database ready for analysis in the GUI

After the Neo4j database was updated, the analysis can be performed. This process is not part of this section because it is not relevant to the data transfer from EA to the prototype and is presented in Section 6 where the complete GUI of the standalone prototype is presented.

4.4 Enterprise Architect Extension for the Prototype

The EA extension that was implemented during this work integrates the prototype with EA. It is written in C# and uses Windows Forms. It provides functionality to choose the file path of the Neo4j database, and it can open an analysis window that was implemented in the prototype. The extension has an optional continuous update which updates the Neo4j database immediately when a relevant element in EA is modified. In addition to that, there is the continuous analysis which performs a full analysis on all CFTs whenever the Neo4j database is updated. This analysis relies on reuse, and small incremental modifications to a CFT do not cause much analysis effort since most previous analysis results can be reused. The extension itself transfers all data and commands to the prototype. The communication is handled with a socket between the applications.

4.4.1 Socket Communication

The extension becomes active after EA is started and a new project is opened. First, the extension tries to launch the prototype and establish a connection with it over a socket.
The class *SocketConnection* exists in the prototype and in the EA extension, and it is used to initiate the socket communication. The EA extension uses the command line of Windows and navigates to the file path of the JRE and starts the prototype with the arguments socket IP-address, socket port and file path to the installation folder. The first two arguments are used to connect to the socket opened by the EA extension, and the last argument is used by the prototype to save logs and store an initial Neo4j database for the project in EA.
After the prototype was started, it connects to the socket of the extension with the provided IP-address and port. It starts the class *SocketReader* which is a thread that reads from the socket and can process commands and data. In the EA extension, the socket communication is handled exclusively by the *SocketConnection*. The communication is always initiated by the EA extension, and it uses several commands to control the prototype. The list of commands with a description can be seen in Table 4.

Table 4 List of commands

Command	Description
start neo4j path **+ file path**	Starts a Neo4j database at the file path.
start full update	Switches the prototype to full update mode. Clears the Neo4j database and starts a batch inserter.
add elements **+ element string**	Used in full update mode. Adds elements with the batch inserter. The elements are inside the element string.
add connectors **+ connector string**	Used in full update mode. Adds connectors with the batch inserter. The connectors are inside the connector string.
end full update	Ends full update mode and shuts down the batch inserter. Starts the Neo4j database with the last file path.
change neo4j path	Shuts down the Neo4j database. The old Neo4j database is copied to a new file path by the extension.
update **+ command string**	Normal update. The commands are add, delete or update for a single element or connector.
exit	Shuts down the Java prototype. The extension sends this command when EA is shut down.
open analysis window	Opens the analysis window of the prototype.
analyze and store results	Performs a full analysis with reuse and stores all analysis results in the Neo4j database.
set developer mode	Opens a window that shows socket traffic between EA extension and the prototype.

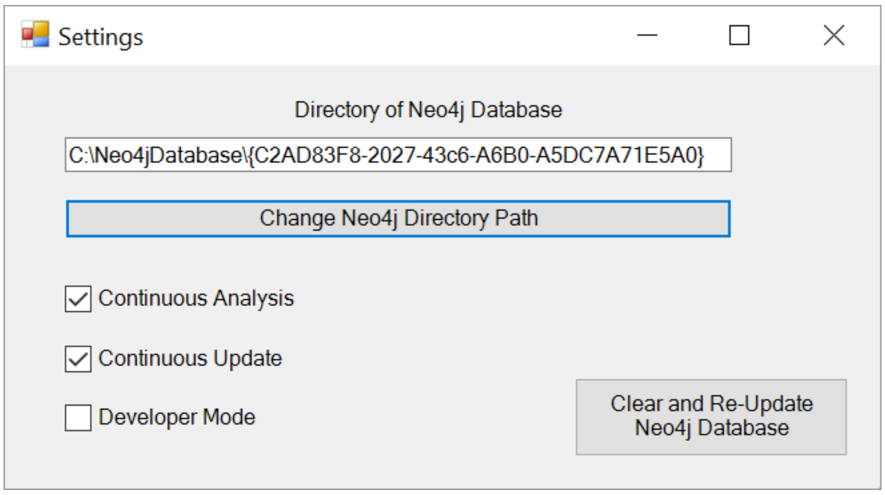
The extension sends string commands which are parsed by the prototype. Various string separators split commands and the different data. For example, an element is separated by the command and other elements with the main separator. A part separator separates the stereotypes and tagged values of an element, and a tagged value separator separates the name and value of a tagged value. These separators allow a simple string parser to get all necessary information. The extension only receives a message that acknowledges that the command was received and processed. If any errors such as a socket disconnection appear, the prototype automatically shuts down and is restarted. Besides the analysis window, the prototype runs in the background.

4.4.2 Update from the Enterprise Architect Extension

The EA extension can update the prototype manually or automatically (continuous update). When a project is opened, then the EA extension first checks if a Neo4j database already exists for that project. If no database exists it creates a new one and asks the user if they want to update the Neo4j database. If the user accepts then all relevant EA elements and connectors are transferred to the prototype which adds them to the Neo4j database. This operation is done with a batch inserter to insert large projects very fast. If the Neo4j database was updated, the EA extension starts to watch for changes in the EA project. Otherwise, the next update that is performed is a full update. When the prototype was updated, the date of the last update is saved. Once the date of the last update is not "never" the EA extension starts to monitor EA and changes to the project. Changes are saved as string commands and are added to a *SetQueue* which is the implementation of a queue of strings that never accepts a string twice. Therefore, equal changes which occur when a change triggers multiple events that are monitored by the extension are not added again. The implementation of *SetQueue* only implements basic methods and uses a regular Queue and HashSet to store strings. The set only gets cleared when the project is closed or if a full update is performed. The queue gets cleared after each update. The set needs to hold string commands that were already sent because in the case of continuous update the update is performed immediately and the string commands of duplicate change events are always added sequentially. Therefore, the set cannot be cleared each time the queue is cleared. The queue is necessary because commands must be sent in the order they were stored. For example, an element cannot be deleted before it was added.

The changes that are detected by the EA extension are the addition, deletion, and modification of elements and connectors. The EA extension considers every selection of an element or connector as a modification because EA does not provide the option to detect element modifications like the name or tagged values. When the next update is sent to the prototype, it determines if the element or connector changed by comparing it to the corresponding node in the Neo4j database. The EA extension can monitor changes by implementing methods provided by the EA interface that is called if certain events occur. The implemented methods specifically inform the EA extension of selection, post-addition, and pre-deletion of elements, connectors and diagram objects. Diagram objects are elements on a diagram. In the case of selection, the element or connector is first stored and later transformed into a string command after another selection has occurred. The

reason for this is that an element or connector can only be changed after it was selected so an immediate update cannot show changes after the selection.
The elements and connectors that were added, deleted or modified are transformed into a command string that specifies the update and contains the relevant information about the element or connector. Then, they are added to the *SetQueue* and if the continuous update is activated the prototype is updated with the single element *SetQueue*. Otherwise, the *SetQueue* stores string commands until the user performs a manual update. If an update is performed all string commands are sent to the prototype where they are parsed, and the Neo4j database is updated accordingly.
The EA extension provides some options that can be changed by the user. Figure 24 shows the settings window with all options.


Settings

Directory of Neo4j Database

C:\Neo4jDatabase\{C2AD83F8-2027-43c6-A6B0-A5DC7A71E5A0}

☑ Continuous Analysis

☑ Continuous Update

☐ Developer Mode

Change Neo4j Directory Path
Clear and Re-Update Neo4j Database

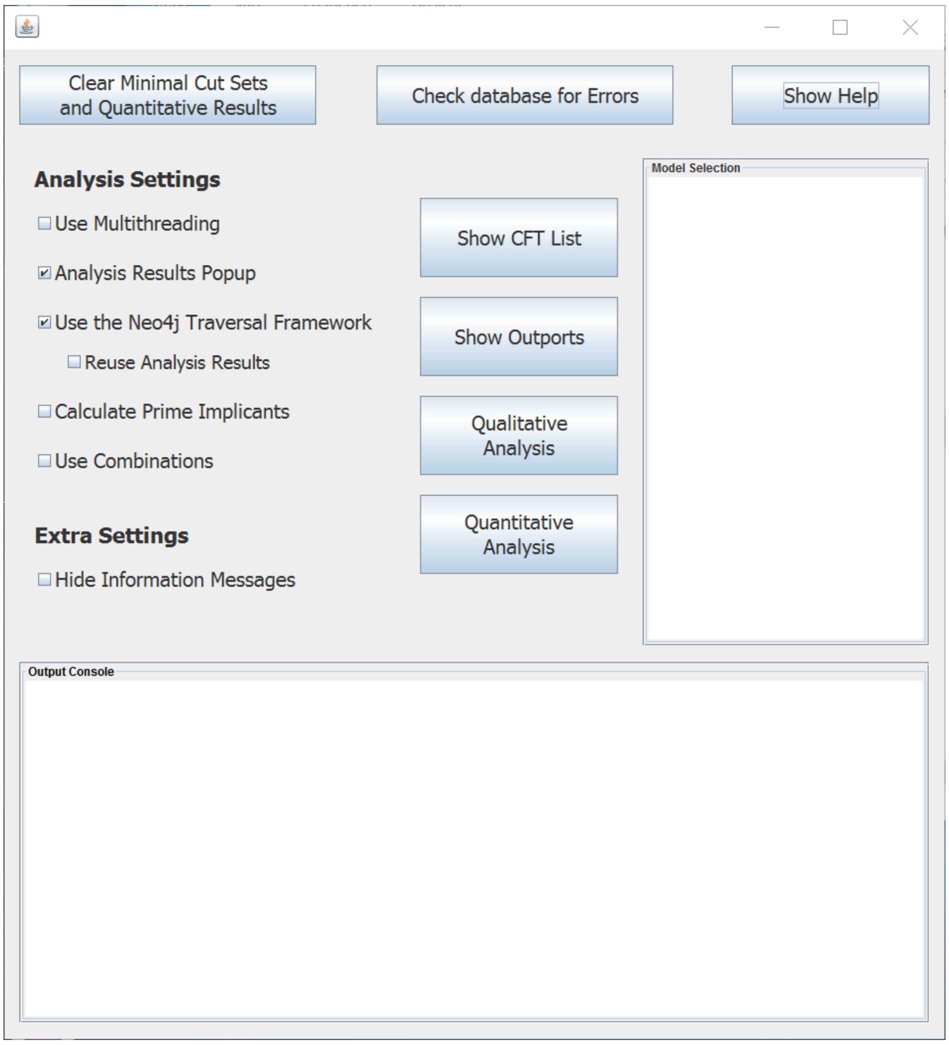
Figure 24 EA Extension Settings

The location of the Neo4j database can be changed to another directory, but the standard location is the same as the installation path of the EA extension. The folder containing the Neo4j database is named after the EA project GUID to ensure that all EA projects have a unique Neo4j database. The option continuous update makes sure that the Neo4j database is updated immediately after each relevant change in the EA project. The option continuous analysis starts a full analysis after each update and stores all analysis results in the Neo4j database for reuse. In the case of small incremental changes, the analysis effort is low because previous analysis results can be reused. The Developer Mode is used to show socket communication and information of the prototype. It is intended to trace errors. The last option is to clear the Neo4j database and perform a full update. This is meant to bring the Neo4j database back into a consistent state in the case severe errors occurred, or updates were not correctly performed. For example, if the user modifies the project without the EA extension being active the changes are not saved, and the Neo4j database cannot recover if the EA extension is activated again. All settings are stored in an XML file that is stored in the installation folder.

The EA extension can be deactivated at any time, but the last update of the Neo4j database is then set to “never” because further changes to the project cannot be detected. If the EA project is closed, the EA extension updates the Neo4j database.

4.4.3 Analysis Window of the Enterprise Architect Extension

The EA extension has the menu option to open a Neo4j analysis window. This window is implemented by the prototype and looks similar to the standalone prototype window. However, it offers less functionality because it is only used for analysis. Figure 25 shows the mentioned analysis window.



Analysis Settings

- ☐ Use Multithreading
- ☑ Analysis Results Popup
- ☑ Use the Neo4j Traversal Framework
  - ☐ Reuse Analysis Results
- ☐ Calculate Prime Implicants
- ☐ Use Combinations

Extra Settings

- ☐ Hide Information Messages

Figure 25 Analysis Window for the EA Extension

The design is simple and shows all necessary analysis settings. All the buttons and options are explained in Section 6. The outport console shows some additional information during the analysis and also prints the analysis results. There are also dedicated windows for the analysis results which are presented in Section 6.

5 Analysis of Safety Models with the Prototype

In this section, the traversal and analysis of component fault trees (CFTs) in the prototype are explained. The prototype implements various traversals and analysis techniques. In Section 5.1, the fundamentals of the Neo4j Traversal Framework, as well as the traversal and analysis results of the prototype, are explained. In Section 5.2, the traversal implementations of the prototype are explained. Section 5.3 describes how the analysis is performed by the prototype. Every section builds on the concepts of the previous section. The traversal and the analysis use optional multithreading to increase performance if the CFT is large and complex enough. The prototype only uses the number of threads that the system can run in parallel. The multithreading is controlled with thread pooling for any process that uses the database. The prototype rejects multiple user commands and always finishes a process before a new one can be started. Multiple threads are only used internally to speed up certain parts of the traversal and analysis. The user can disable the multithreading. Any use of multithreading is mentioned in the following subsections.

5.1 Prerequisites for the Analysis

In this section, the fundamentals of the traversal and analysis in the prototype are explained. These are the Neo4j Traversal Framework, traversal results, and analysis results. These fundamentals are important for understanding the different traversal and analysis approaches.

5.1.1 Neo4j Traversal Framework

The traversal of the Neo4j database is one of the key parts of this work, so all fundamentals of the Neo4j Traversal Framework are introduced in this section.
The Neo4j Traversal Framework[11] [12] is part of the Neo4j API and consists of the main interfaces that are necessary for graph traversal. Figure 26 shows the relationships between these interfaces.

[11] https://neo4j.com/docs/java-reference/current/#tutorial-traversal (accessed: 15.03.2017)

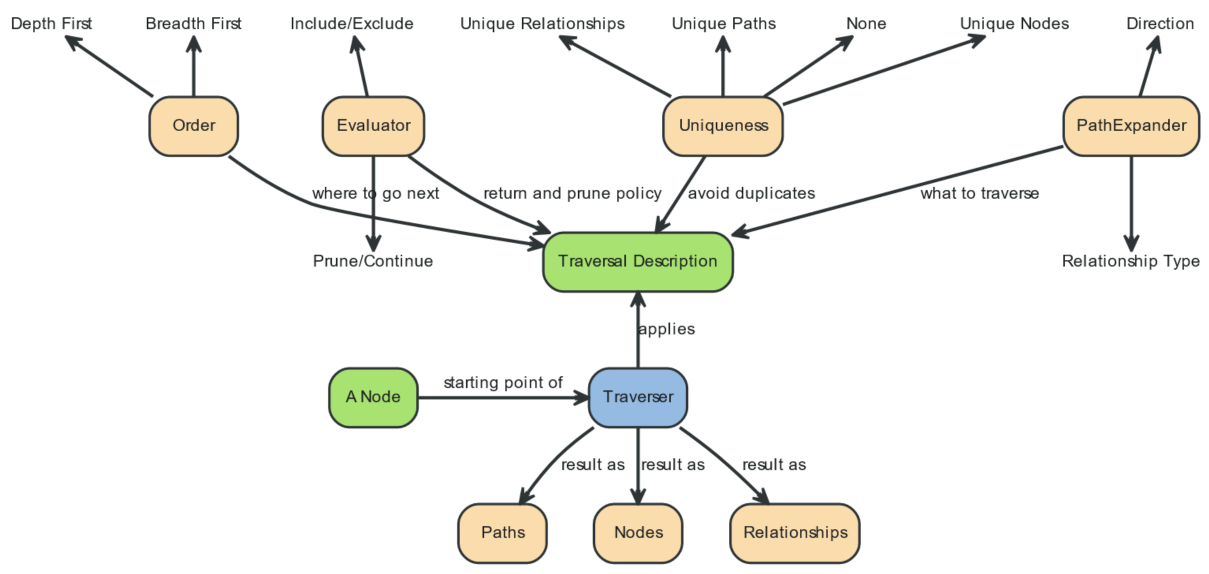


Figure 26 The Neo4j Traversal Framework

A traversal using the Neo4j Traversal Framework always starts by defining a *Traversal Description* and then using a *Traverser* to traverse the graph according to the *Traversal Description*. The traversal is performed by creating paths from the starting point of the traversal. This starting point can be a single node or multiple nodes. The paths that are created by the traversal split into multiple branches if multiple relationships are traversed. At the end of a traversal these paths may be returned and used, but the traversal can also be implemented to process data internally and not return anything. In the following, the main interfaces of the Neo4j Traversal Framework are described.

The *Traversal Description* is the main interface and is used to define the traversal. There are four main characteristics of a *Traversal Description* which decide how the graph is traversed. These are depicted above the *Traversal Description* in Figure 26 and are described in the following.

The *Order* defines the order of the traversal, and the two main orders are depth-first and breadth-first. It is possible to define a custom order, but in the prototype, the depth-first order is used exclusively because it is more memory efficient. When breath-first is used then the traversal must keep more traversal branches in memory which becomes inefficient if a CFT has gates with many incoming failure propagations.

Uniqueness defines if nodes, relationships or paths are traversed multiple times or not. This *Uniqueness* can be used to avoid traversing cycles in a graph. However, the prototype does need to traverse some nodes multiple times because it is possible that a CFT contains multiple instances of the same CFT. Therefore, the *Uniqueness* "None" is always used so that the traversal ignores repeating nodes, relationships or paths. To prevent traversing cycles, the prototype implements a cycle detection that stops traversals if a cycle is found.

The *Evaluator* controls part of the traversal and is called when a traversal path is created or extended. It returns an evaluation which decides if the traversal continues or stops and if the current path should be returned as a result of the traversal.

The *PathExpander* is the main interface for controlling the traversal. The *PathExpander* decides which relationships are traversed and is called at every step of the traversal. The prototype makes use of the *PathExpander* to create the traversal results which are used for the analysis.
The *Traverser* performs the analysis according to the *Traversal Description* and returns any paths that are returned by an *Evaluator*. It should be noted that the *Traverser* performs the analysis separately for each result and not in advance. Therefore, a full traversal is only performed when the iterator of the *Traverser* is exhausted.
An example of a *Traversal Description* and *Traverser* is shown below.

```
TraversalDescription ftTD = DBConnection.getGraphDB()
            .traversalDescription()
            .uniqueness(Uniqueness.NONE)
            .depthFirst()
            .relationships(RelTypes.Failure_Propagation, Direction.INCOMING)
            .evaluator(Evaluators.all());
Traverser ftTraverser = ftTD.traverse(outport);
ftTraverser.iterator().forEachRemaining(path ->
{
      PrintUtility.printResults(path);
});
```

This traversal starts at the outport of a CFT and traverses all incoming failure propagations of the outport. The traversal continues until a node is reached that has no incoming failure propagations (e.g. basic event, inport or outport instance). All paths of the traversal are printed as a result. In this case, all partial paths are also returned because the traversal returns the current path at every step of the traversal.
By using the method `uniqueness(Uniqueness.NONE)` any node, path or relationship may be traversed multiple times. In this case, the *Traversal Description* has no *PathExpander* that could prevent traversing cycles which would result in an indefinite traversal if the CFT contains a cycle.
The method `depthFirst()` ensures that the traversal order is depth-first so that every traversal branch is fully expanded before other branches are continued. Therefore, the printed results are ordered in a way that the progression of each path is visible because each path is traversed until it is completed.
The method `relationships(RelTypes.Failure_Propagation, Direction.INCOMING)` is a simple *PathExpander* that always expands a path if the last node of the path has a relationship with the specified relationship type and direction.

By using the method `evaluator(Evaluators.all())`, a simple *Evaluator* is created that always returns the current path and continues the traversal according to the *PathExpander*.
The prototype uses traversal in a similar way, but it implements several *PathExpanders* and an *Evaluator* to control the traversal and create the traversal results that are utilized for the analysis.
This concludes the introduction to the Neo4j Traversal Framework and serves as a basis for understanding the implementation of the traversal in the prototype. In Section 5.2, the specific implementation of the Neo4j Traversal Framework in the prototype is explained. It is also possible to traverse the graph without the Neo4j Traversal Framework by using the base Neo4j API. Such an approach is also implemented in the prototype and can be found in Section 5.2.3.

5.1.2 Traversal Results

In the prototype, the traversal results are implemented in the interface GateSet. A GateSet represents a CFT element but not all CFT elements become GateSets during the traversal. The idea is that the structure of a CFT is saved in GateSets during the traversal. These GateSets are used for the analysis.
A GateSet has a start node which is the CFT element that is represented by the GateSet. A GateSet contains a set of nodes and a set of GateSets which are also called lower GateSets. The set of nodes only contains basic events, inports, and outports. The lower GateSets are GateSets of nodes that have a failure propagation to the start node. Also, instead of having failure propagations, the start nodes of lower GateSets can be instances or classifiers of the start node. The analysis of a GateSet is implemented to analyze all lower GateSets and then continuing the analysis with the results. Depending on the implementation of the GateSet the analysis is performed differently. In the following, an example of GateSet creation and analysis is presented.

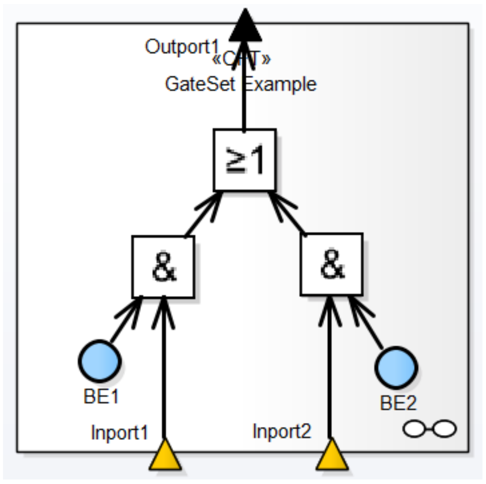


Figure 27 GateSet Example

Figure 27 shows the CFT called *GateSet Example* and the traversal starts at the outport called *Outport1* and the first GateSet is created with that outport. The traversal continues and finds an OR gate which is called *OR1*. A new GateSet is created with *OR1* and added as a lower GateSet of the *Outport1* GateSet. Next, the traversal finds the left AND gate which is called *AND1* and right AND gate which is called *AND2*. New GateSets are created with *AND1* and *AND2*, and they are added as a lower GateSet to the *OR1* GateSet. In this example, the traversal continues on the left side and finds the basic event called *BE1* and the inport called *Inport1* below the left AND gate. These nodes are added to the node set of the *AND1* GateSet. Then the traversal continues at the right AND gate and finds the basic event called *BE2* and the inport called *Inport2* below the right AND gate. These nodes are added to the node set of the *AND2* GateSet. Then, the traversal is finished. Below, the GateSets and their contents are shown.

GateSet of *Outport1*
- Lower GateSets:
  - GateSet of *OR1*
    - Lower GateSets:
      - GateSet of *AND1*
        - Nodes: { *Inport1, BE1* }
      - GateSet of *AND2*
        - Nodes: { *Inport2, BE2* }

The analysis is performed bottom-up, and the lower GateSets are always analyzed first, and the results are used for the analysis of the higher-level GateSets. The set of minimal cut sets of each GateSet is shown next.

AND1 GateSet:	{ { *Inport1*, *BE1* } }
AND1 GateSet:	{ { *Inport2*, *BE2* } }
OR1 GateSet:	{ { *Inport1*, *BE1* }, { *Inport2*, *BE2* } }
Outport1 GateSet:	{ { *Inport1*, *BE1* }, { *Inport2*, *BE2* } }

The GateSet has implementations for different CFT elements because the analysis is different for these elements. Table 5 shows the GateSet implementations for the corresponding labels.

Table 5 GateSet Implementations

GateSet	Labels
ANDSet	CFT_AND_Gate
ORSet	CFT_OR_Gate
NOTSet	CFT_NOT_Gate
XORSet	CFT_XOR_Gate
MOONSet	CFT_MOON_Gate
ConnectorGateSet	CFT_Inport, CFT_Outport_Instance
ResultGateSet	CFT_Outport, CFT_Inport_Instance

The class *ConnectorGateSet* is used to connect GateSets together which is explained in Section 5.3.3. The class *ResultGateSet* is used for traversal results, and the analysis always starts at a *ResultGateSet*. The analysis results of the GateSet are explained in the next section.

5.1.3 Analysis Results

The qualitative analysis results are stored with the class MCS. A single MCS represents a prime implicant or a minimal cut set that may contain negated events. Typically, the qualitative analysis results consist of multiple MCSs, and they stored in a set. The quantitative analysis results are not specified in detail because they are only a minor part of the prototype.

An MCS has a set of nodes that are stored as *MCSNodes*. The class *MCSNode* contains a node from the database as well as the property negated which identifies if the node is considered negated. These *MCSNodes* represent variables in a prime implicant. In the remainder of this work the *MCSNode* is called node, and if the node is negated, then it is specifically mentioned.

The qualitative analysis creates MCSs in the GateSet, and they are stored in two different MCS sets because there are two main types of MCSs. The first type is the regular MCS which represents a prime implicant for that specific GateSet. The second type is the negated MCS which represents a prime implicant for that GateSet if it is negated. For example, an AND gate with two basic events A and B would have the regular MCS {A, B}. However, there would be two negated MCSs {¬A}, {¬B}. The set of negated MCSs is the qualitative analysis result for the negated GateSet. The GateSet provides methods for calculating both types of MCSs, and they are used to improve performance. The idea behind these two types is explained in Section 5.3.

An MCS can be considered a set of nodes that are logically connected by AND. Every MCS in a set of MCSs can be considered logically connected by OR. This concept is important when multiple sets of MCSs are logically combined with AND because every MCS in each set must be combined with every MCS of the other sets. Combining MCSs means that all nodes of each MCS are added one node set. For example, take the MCS { A, B } and the MCS set { { C, D }, { E, F } }. If the MCS and the MCS set are combined the resulting MCS set would be { { A, B, C, D }, { A, B, E, F } }. A combination of MCS sets is expensive because it requires many set operations and the number of MCSs increases. Therefore, the prototype uses a technique called merging to reduce the number of MCSs initially and later use multithreading to combine MCS sets.

The two important methods of the class MCS are `addMCS(MCS mcs)` and `mergeMCSSet(HashSet<MCS> mcsSet)`. The first method adds an MCS to the MCS so that every node is added to the node set of the MCS. The second method merges a set of MCSs to the MCS where the method is called. It adds the MCS set to a list of merged MCS sets that is part of every MCS. If an MCS has any merged MCS sets in the list, then this MCS represents an MCS set that is not yet created. However, the analysis is not affected by these MCSs because all merged MCS sets are logically connected by AND so they can be treated as effectively one MCS. The prototype assures that these MCSs are correctly added and merged to other MCSs.

After the analysis is completed, the MCSs are combined with the method `combineAllMergedMCS()` in the GateSet. All MCSs that have merged MCS sets are then combined with them. Multithreading is used to combine MCS sets faster. The performance benefits of this technique are discussed in Section 6.2.1. The creation of MCSs is explained in Section 5.3. The prototype also can store analysis results in the database and reuse them when the analysis is performed. The remainder of this section describes how analysis results are stored in the database.

Multiple types of MCSs are stored in the database. These are partial MCSs, partial negated MCSs, full MCSs, and full negated MCSs. Partial and full MCSs are explained in Section 5.3.2. Besides these different MCSs, the prototype can also store quantitative analysis results in the database. Table 6 shows the labels of all analysis results.

Table 6 Labels for Analysis Results

Label	**Description**
MCS	A partial MCS.
Negated_MCS	A partial negated MCS.
Full_MCS	A full MCS.
Full_Negated_MCS	A full negated MCS.
Quant_Result	A quantitative analysis result. Has the property *Result* to store the result.

The analysis results belong to a CFT element, and they have a relationship to the node of that element. Nodes that are inside an MCS are represented in the database by relationships from the node to the MCS. Table 7 shows all relationship types for analysis results.

Table 7 Relationship Types of Analysis Results

Relationship Type	Description
Is_MCS_Of	The relationship from the partial MCS to the node of the CFT element.
Is_Negated_MCS_Of	The relationship from the partial negated MCS to the node of the CFT element.
Is_Full_MCS_Of	The relationship from the full MCS to the node of the CFT element.
Is_Full_Negated_MCS_Of	The relationship from the full negated MCS to the node of the CFT element.
Is_Quant_Result_Of	The relationship from the quantitative result to the node of the CFT element.
Is_Inside_MCS	The relationship from a node contained in an MCS to its MCS. Has the property *Negated* that shows if the node is negated inside the MCS.

Analysis results are updated when the database is modified. The maintenance of analysis results in the database is explained in Section 5.3.2.

5.2 Traversal Implementation of the Prototype

The prototype implements different graph traversals to traverse the CFT and create the GateSets. An analysis always starts at an outport of a CFT. In the following subsections, the different traversal implementations of the prototype are explained. First, the traversal state is described which is used by all traversal implementations. Second, the cycle traversal is described as an introduction to the following traversals. Lastly, the three main traversal implementations of the prototype are explained.

5.2.1 Traversal State

All traversals in the prototype use a traversal state to build the GateSets and find cycles in a CFT. The class *TraversalState* implements this traversal state. A *TraversalState* stores the last GateSet that was created by the traversal, the list of inport maps, and two sets of nodes that are used to find cycles in a CFT. The last GateSet is used to add new lower GateSets and nodes to the last GateSet that was created by the traversal. An inport map is a mapping of inports to specific inport instances. Inport maps are created at an outport instance, and they allow the traversal to continue at inports. Inports can have many inport instances, and the inport map makes sure that the traversal continues at the correct inport instance. An example of an Inport with multiple inport instances is shown in Figure 28.

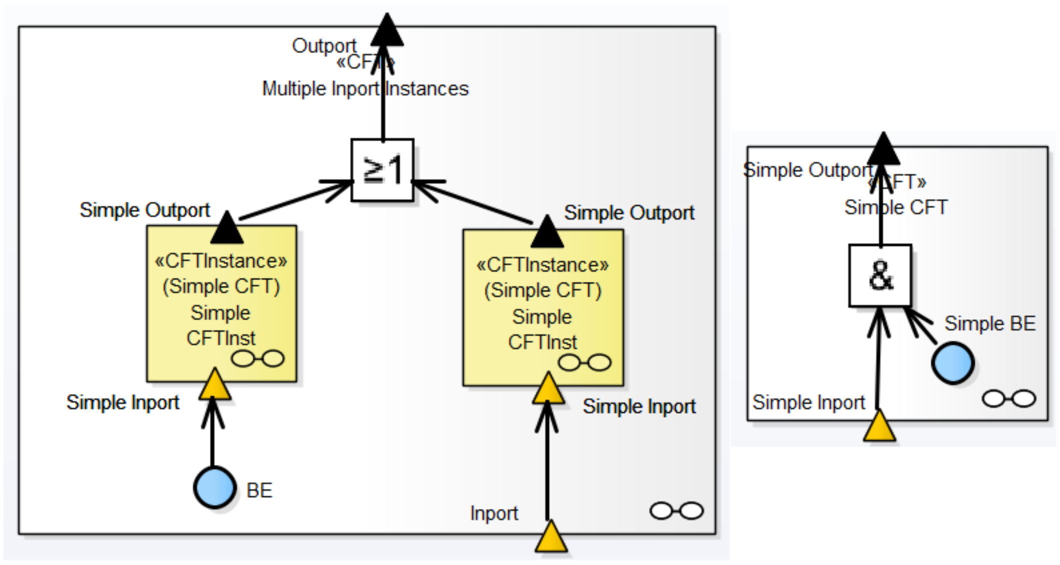


Figure 28 Inport with multiple Inport Instances

The CFT on the right called *Simple CFT* has two CFT instances in the left CFT called *Multiple Inport Instances*. These CFT instances have different inport instances and outport instances, but when the traversal enters the CFT, it finds the same inport which has two possible inport instances. Therefore, the traversal must know which inport instance must be used to continue the traversal.

When the traversal reaches an outport instance, it finds all inport instances of the specific CFT instance and then maps all inports of the underlying CFT to the inport instances of the CFT instance. If the traversal continues and reaches an inport, the last inport map is used to find the specific inport instance that is traversed next. The *TraversalState* has a list of inport maps because there can be multiple CFT abstraction levels. When an inport map is used, it is also removed from the list because it is never used again. The reason is that a traversal can only enter a CFT at an outport and only leave a CFT at an inport. Therefore, if the inport map is used, the traversal has left the CFT, and that inport map can never be used again. The last entry in the list of inport maps is also the only one that can be used because the traversal can only find inports that part of the current CFT, otherwise another CFT was entered which would add another inport map. The *TraversalState* gets copied each time the traversal continues which ensures that all traversal branches have their specific list of inport maps. If an inport map does not exist

for the inport, then the traversal ends there because the inport must belong to the same CFT as the outport where the traversal started.
A *TraversalState* contains two sets of nodes that are used to find cycles in CFT. The first set is for finding simple cycles in a CFT. Simple cycles are cycles that do not propagate through CFT instances. For example, an AND gate that has a failure propagation to itself would be a simple cycle. The set for finding simple cycles is filled with nodes that are traversed. Each time the traversal enters a CFT instance the set is cleared because simple cycles can only occur in a single CFT. The second node set of the *TraversalState* is for finding deep cycles in a CFT. Deep cycles are cycles that propagate through CFT instances. Only inport instances and outport instances are added to the set for finding deep cycles. Inport instances and outport instances are always specific to a CFT instance, so if they are traversed twice, then a deep cycle is found. With both sets, the *TraversalState* detects the shallow and deep cycles that are mentioned in [5]. A cycle is considered detected if the set already contains the node that is added. All traversals of the prototype stop traversal if a cycle is detected. All traversals use the same cycle detection, and it is built in their traversal approach. In the following subsections, the last GateSet refers to the last GateSet in the *TraversalState*.

5.2.2 Traversal for finding Cycles

This section describes the traversal for finding cycles and also serves as an introduction to the traversals described in later sections. This traversal is used to detect cycles and errors in a CFT. It is used when the prototype is used together with the EA extension to prevent background analysis of CFTs that have cycles or errors. For user analysis with the GUI, the analysis shows an error message if the analysis of such a CFT is attempted.
The class *CycleTraversal* uses the Neo4j Traversal Framework and starts the traversal at an outport. The method `hasCycleOrErrors(Node outport)` of *CycleTraversal* returns true if the traversal of the outport finds any errors or cycles. It uses a full traversal that does not return any paths, and the traversal is fully performed by the class *CyclePathExpander*. The *CyclePathExpander* is an implementation of the *PathExpander* in the Neo4j Traversal Framework. In the following, the *PathExpander* is explained, and after that, the specific implementation of the *CyclePathExpander* is described.
The method `expand(Path path, BranchState<?> branchState)` of the *PathExpander* returns a collection of relationships which are traversed further. Every relationship in the collection creates a new path by copying the current path of the traversal and adding the relationship to it. Therefore, if many relationships are returned the traversal splits into different traversal branches which are traversed according to the order of the traversal.

The *PathExpander* continues each traversal branch until it returns no relationships or if the *Evaluator* of the traversal stops the traversal. The *PathExpander* allows using the *BranchState* to keep states during the traversal. In the prototype, the *TraversalState* is used as the *BranchState*.
The *CyclePathExpander* implements the *PathExpander* and is designed to stop traversal if any cycles or errors are found. Every time the database is fully updated, the prototype checks all nodes for errors and rechecks any nodes that are added or modified. A node has errors if its relationships do not match the expected relationships for the label of the node. For example, a basic event should not have incoming failure propagations and elements that are not gates should not have multiple incoming failure propagations. The *CyclePathExpander* does not create GateSets, and the last GateSet of the *TraversalState* is always null.
The *CyclePathExpander* starts with an outport and a *TraversalState* that is null. It creates a new *TraversalState* and adds the outport to the set for finding simple cycles. Then, it adds all incoming failure propagations of the outport to the list of relationships that is returned for further traversal. If the list is not empty, the *CyclePathExpander* continues the traversal with an existing *TraversalState*. In Table 8, the behavior of the *CyclePathExpander* is described depending on the label of the last node in the current path.

Table 8 CyclePathExpander Implementation

Label of the Node	Actions for this Node
CFT_*_Gate **CFT_Outport**	All incoming failure propagations are added to the list of relationships that are returned.
CFT_Inport_Instance	The inport instance is added to the set for finding deep cycles. All incoming failure propagations are added to the list of relationships that are returned.
CFT_Outport_Instance	An inport map is created and stored it in the *TraversalState*. The set for finding simple cycles is cleared, and the outport instance is added to the set for finding deep cycles. The relationship to the classifier (outport) of the outport instance is added to the list of relationships that are returned. Also, the set for finding simple cycles is cleared.
CFT_Inport	The last created inport map is used, and the relationship to the mapped inport instance is added to the list of relationships that are returned. Also, the set for finding simple cycles is cleared.

5.2.3 Traversal without the Neo4j Traversal Framework

The prototype includes a traversal that relies on the base API of Neo4j. It is used for comparison with the Neo4j Traversal Framework and uses multithreading to increase performance. The class *ManualTraversal* implements the complete traversal and uses recursion to continue traversal.

The traversal is started with the method `startManualTraversal()` and uses the outport that was selected by the user. The *ManualTraversal* performs a full traversal and returns a single *ResultGateSet* of the outport. This *ResultGateSet* contains all traversed CFT elements as lower GateSets or nodes.

When a traversal is started, the *ResultGateSet* of the outport is created, and a new *TraversalState* is created with that *ResultGateSet*. The outport can only have a single incoming failure propagation, and the traversal continues depending on the label of the start node of that failure propagation.

If that node is one of the five gates, a GateSet that is specific to the gate is created with the node and added to the last GateSet of the *TraversalState*. Then, the last GateSet of the *TraversalState* is set to the GateSet that was created from the gate.

If the node is a basic event or inport, then the node is added to the node set of the last GateSet. In this case, the traversal would end because inports and basic events should not have incoming failure propagations.

If the node is an outport instance, a new inport map is created and added to the *TraversalState*. A *ConnectorGateSet* is created with that node and added to the last GateSet, and the last GateSet is set to the *ConnectorGateSet*. Then, the classifier (outport) of the outport instance is used to create a *ResultGateSet* which is added to the last GateSet, and the last GateSet is set to the *ResultGateSet*. The traversal continues with the current *TraversalState*.

The method `traverseState(TraversalState traversalState)` uses recursion to continue different traversal branches. First, a list of *TraversalStates* is created which is used for the next traversals after the current traversal ends. The start node of the last GateSet is retrieved. Then, for every incoming failure propagation of this start node, a new *TraversalState* created by copying the current *TraversalState*. Therefore, each failure propagation represents a traversal branch. Then, the starting node of the failure propagation is checked for its label. Table 9 shows the actions that are performed depending on the label of the node.

Table 9 ManualTraversal Implementation

Label of the Node	Actions for this Node
CFT_*_Gate CFT_Outport, CFT_Inport_Instance	The corresponding GateSet is created and added to the last GateSet. The last GateSet is set to the created GateSet.
CFT_Basic_Event	The node is added to the node set of the last GateSet, and the traversal of this traversal branch is stopped.
CFT_Outport_Instance	An inport map is created and added to the list of inport maps. A *ConnectorGateSet* is created and added to the last GateSet. The last GateSet is set to the *ConnectorGateSet*. The classifier outport is found, and a *ResultGateSet* is created from the outport and added to the last GateSet. The last GateSet is set to the *ResultGateSet*.
CFT_Inport	If the inport belongs to the CFT where the traversal started then the node is added to the node set of the last GateSet, and the traversal at this traversal branch is stopped. Otherwise, a *ConnectorGateSet* is created and added to the last GateSet. The last GateSet is set to the *ConnectorGateSet*. The last inport map is used to find the corresponding inport instance, and a *ResultGateSet* is created from the inport instance. It is added to the last GateSet. The last GateSet is set to the *ResultGateSet*. The last inport map is removed.

Each new *TraversalState* is added to the list of *TraversalStates*, and the method `traverseState(TraversalState traversalState)` is used on each *TraversalState* in the list. If multithreading is activated, the method is used concurrently on different *TraversalStates*.

5.2.4 Traversal with the Neo4j Traversal Framework

The prototype uses the Neo4j Traversal Framework and implements two versions of it. The class *Neo4jTraversal* contains the traversal methods for both versions. The first version is a full traversal like the traversal in the previous section and uses the *CFTPathExpanderFull* which is an implementation of *PathExpander*. The traversal of the

first version is started with the method `traverseOutportFull()`. The traversal of the first version works the same way as the previous traversal, and the *CFTPathExpanderFull* is implemented similarly to the *CyclePathExpander* with the exception that this traversal creates GateSets. Therefore, a full explanation is redundant, and this section focuses on the implementation of the second version which is very different to the previous traversals.
The second version is used when analysis results are stored in the database, and it performs a so-called split traversal which is different to a full traversal in the way that multiple *ResultGateSets* are created that are related but not connected to each other.
The traversal of the second version is started with the method `traverseOutportSplit(Node outport)`. The returned *ResultGateSets* are called partial CFTs or partial traversal results and can be analyzed individually, and they can be combined to a full traversal result.
The idea is that the traversed CFT is split into partial CFTs which are *ResultGateSets* that do not contain any CFT instances. Therefore, the analysis results of partial CFT are not affected by changes of elements that are not part of that partial CFT. The benefit is that these analysis results do not change with a high rate compared to the overall analysis results which are affected by almost every change in a CFT. An example of this concept is shown in Figure 29.

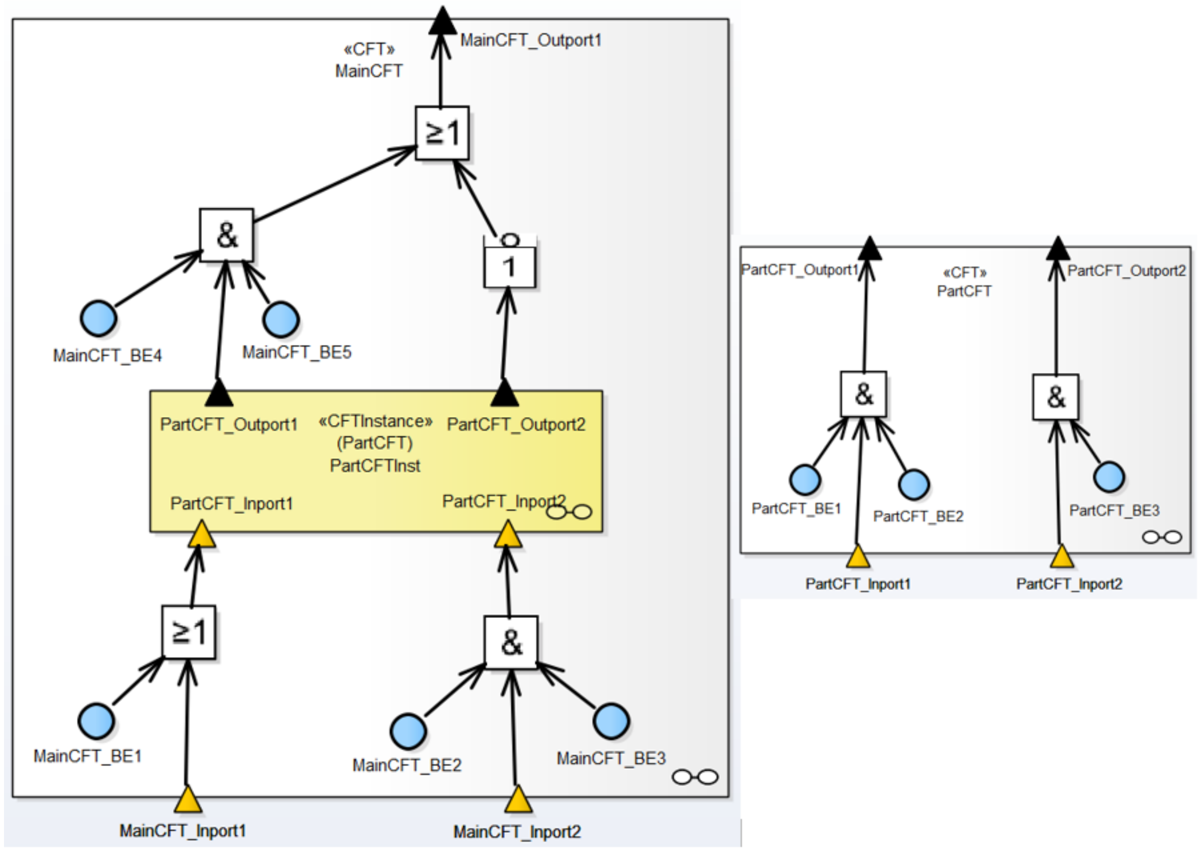


Figure 29 Partial Traversal Results Example

In this example, the analysis results of the outport *MainCFT_Outport1* are affected by any changes in the *MainCFT* and the *PartCFT* on the right side. For example, if any basic event is added or removed it has a direct effect on the analysis result of the *MainCFT_Outport1*. Therefore, storing only the analysis results of that outport would not be very efficient because it would become invalid with every modification of the CFT or its CFT instances. Therefore, single analysis results of large CFTs are very volatile to changes and the prototype splits CFTs into partial CFTs and stores the analysis results of these partial CFTs.

For this example, the split traversal creates 5 partial CFT if the traversal starts at the outport *MainCFT_Outport1*. These 5 partial CFTs are displayed in Figure 30.

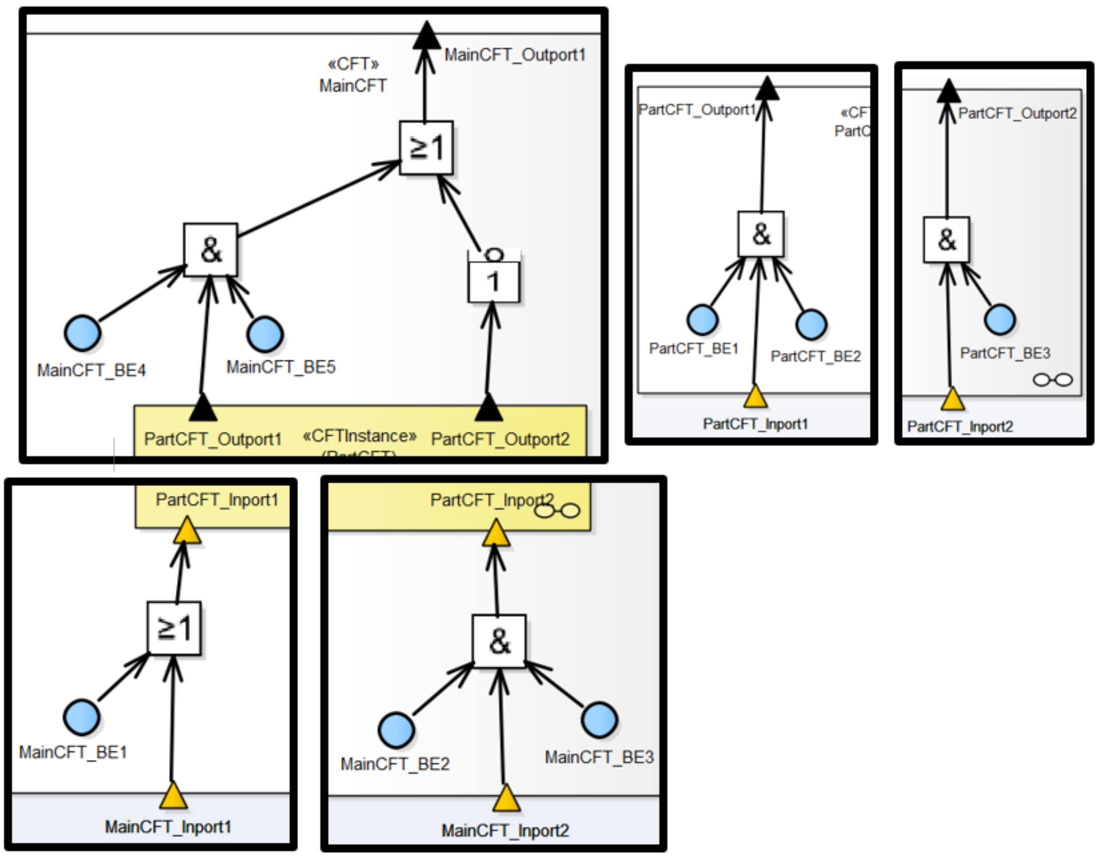


Figure 30 Partial CFT Example

The first partial CFT starts at the *MainCFT_Outport1* and ends at the outport instances *PartCFT_Outport1* and *PartCFT_Outport2*. Then, two partial CFTs are created that start at the inport instances *PartCFT_Inport1* and *PartCFT_Inport2*. The last two partial CFT are created from the traversal of both outports of the CFT called *PartCFT* seen on the right side of Figure 29. The traversal returns all these partial CFTs in a separate *ResultGateSet*. The combination of these *ResultGateSets* to a single *ResultGateSet* is explained in Section 5.3.3. In the remainder of this section, the technical details of the traversal are discussed.

The first step of the traversal is to create a list of nodes that are traversed next and the second list of nodes that are traversed now. The outport is added to the second list, and all inport instances in the CFT are also added to that list. The traversal stops at any inports

so the inport instances must be traversed separately. The *Traversal Description* uses the *PathExpander* called *CFTPathExpanderSplit* and the *Evaluator* called *CFTEvaluatorSplit*. Instead of traversing a single node, all nodes from the second list are traversed. The traversal returns paths with an outport as the end node. These outports are traversed again using recursion.
The *CFTPathExpanderSplit* works similar to the *CFTPathExpanderFull*, but some labels are handled differently. Firstly, the traversal can start at outports and inport instances, and whenever a traversal is started, the *CFTPathExpanderSplit* creates a *ResultGateSet* with the starting node and adds it to the set of *ResultGateSets* that are returned. Then, all failure propagations to the starting node are added to the relationships that are returned by the *CFTPathExpanderSplit* for further traversal.
In Table 10, the actions by the *CFTPathExpanderSplit* for nodes after the starting node are shown.

Table 10 CFTPathExpanderSplit Actions

Label of the Node	**Actions for this Node**
CFT_*_Gate	The corresponding GateSet is created and added to the last GateSet. The last GateSet is set to the created GateSet. All incoming failure propagations are added to the list of relationships that are returned.
CFT_Basic_Event **CFT_Inport**	The node is added to the node set of the last GateSet and traversal of this traversal branch is stopped.
CFT_Outport	This node is ignored by the CFTPathExpanderSplit. The CFTEvaluatorSplit returns the path with the outport and stops the traversal.
CFT_Outport_Instance	An inport map is created and added to the list of inport maps. A ConnectorGateSet is created with the inport map and added to the last GateSet. The last GateSet is set to the ConnectorGateSet. The relationship to the classifier (outport) of the outport instance is added to the list of relationships that are returned.

Inport instances are only traversed at the start of a traversal because the traversal stops at every inport. If the traversal finds an outport instance, it creates an inport map, but the *TraversalState* is not used to save the inport map. Instead, a *ConnectorGateSet* is created from the outport instance and the inport map. This *ConnectorGateSet* can later be used to connect other GateSets to it. Also, the node of the classifier outport is added to the node set of the *ConnectorGateSet*. However, the traversal does not stop, and the relationship to the classifier outport is added to the list of relationships returned by the *CFTPathExpanderSplit*. If the *CFTPathExpanderSplit* finds that outport continuing the traversal, it ignores it because it is handled by the *CFTEvaluatorSplit*. If an inport is found, the traversal adds it to the node set of the last GateSet. Otherwise, the *CFTPathExpanderSplit* works like the *CFTPathExpanderFull*. The *CFTEvaluatorSplit* returns a path and stops the traversal whenever an outport is the end node of the evaluated path, and the path is not zero-length. The zero-length path contains only the start node which can be an outport or inport instance. Otherwise, the *CFTEvaluatorSplit* returns nothing and continues the traversal. The end nodes of the paths that are returned by the traversal are put into the list of nodes that are traversed next. The method `traverseOutportSplit(Node outport)` is used on every node of that list. The split traversal can use multithreading to start the next traversals. The idea is that every outport and every inport instance creates their *ResultGateSet* which can be analyzed independently and combined later. The combination and analysis of these split *ResultGateSets* are explained in Section 5.3.3.

5.3 Analysis Implementation of the Prototype

The analysis starts at a *ResultGateSet* and calculates the analysis result. The prototype implements a qualitative analysis using prime implicants and a simple quantitative analysis using a basic failure probability instead of failure rates.
The main scope of this work is the qualitative analysis. Nevertheless, Section 6.3.1 summarizes the basic concept of how a quantitative analysis using the graph database could be implemented. In the following subsections, various aspects of the analysis are explained. Section 5.3.1 explains the analysis implementation of individual GateSets and Section 5.3.2 describes how analysis results are stored in the database. In Section 5.3.3, the connection of *ResultGateSets* and analysis results is explained which occurs when the traversal is split which was described in the previous section. Section 5.3.4 provides an overview of the analysis process and gives an example.

5.3.1 Analysis Implementation for the Traversal Results

The GateSet has two different results for qualitative analysis and can calculate a simple quantitative analysis result. The first qualitative analysis result is the set of MCSs and the second result is the set of negated MCSs. The core of the qualitative analysis implemented in two methods which create the MCSs for the respective sets. The first

method is the `createMCSSet()` which creates the set of MCSs and the second method is `createNegatedMCSSet()` which creates the set of negated MCSs. These methods are called depending on if the higher-level GateSet needs MCSs or negated MCSs. The benefits of using two methods instead of one is shown in the following example.
Imagine a large CFT where the first gate below the analyzed outport is a NOT gate. In a normal bottom-up calculation of MCSs, the last operation at the top NOT gate would require a negation of the complete set of MCSs. For large sets of MCSs, this requires exponential set operations because every MCS has to be negated and combined with every MCS. For example, take the three MCSs shown below.
{ A, B, C }, { D, E, F }, { G, H, I }
Part of the negated version of these MCSs is shown below.
{ ¬A, ¬D, ¬G }, { ¬A, ¬D, ¬H }, { ¬A, ¬D, ¬I }, { ¬A, ¬E, ¬G }, { ¬A, ¬E, ¬H }, { ¬A, ¬E, ¬I }, { ¬A, ¬F, ¬G }, { ¬A, ¬F, ¬H }, { ¬A, ¬F, ¬I }, { ¬B, ¬D, ¬G }, …
There are 3 * 3 * 3 = 27 set operations to negate the three MCSs. A higher number of MCSs always causes a loss in performance.
The *AbstractGateSet* which is an implementation of GateSet implements two methods to call the respective qualitative analysis methods on all lower GateSets. All implementations of GateSet in the prototype extend the *AbstractGateSet* so they can use these methods. The method `createMCSSetInLowerGateSets()` creates the set of MCSs in all lower GateSets. The method `createNegatedMCSSetInLowerGateSets()` creates the set of negated MCSs in all lower GateSets. These two methods can use multithreading to create MCSs in multiple lower GateSets.
The quantitative analysis can be performed by calling the method `calculateProbability()`, and it returns the basic failure probability of the GateSet. The quantitative analysis does not consider repeated events and produces inaccurate results if they exist. The quantitative analysis always creates a list of failure probabilities. The failure probability of each node is added to that list, and then the failure probability of all lower GateSets is calculated and added to the list.
Every implementation of the GateSet creates the set of (negated) MCSs of all lower GateSets before it creates its set of (negated) MCSs. Depending on which method is called, the set of MCSs or the set of negated MCSs is created.
In the following tables, the analysis of each GateSet implementation is explained. First, the logical function of the GateSet is described, and then the two methods for qualitative analysis are explained. After that, the quantitative analysis of the GateSet is explained. The result of the quantitative analysis is represented by a mathematical formula whenever possible. Any running variables represent positions in the list of failure probabilities that is created for the quantitative analysis.

Table 11 NOTSet Analysis

NOTSet	Description
Logical Function	The *NOTSet* represents the NOT gate, but it is implemented as a NAND gate to compensate for multiple inputs. It evaluates to true if any input is false. The negated *NOTSet* evaluates to true if at least one input is true.
Creating MCSs	The set of negated MCSs is created in all lower GateSets, and they are all added to the set of MCSs. Each node is negated and put into an MCS and added to the set of MCSs.
Creating negated MCSs	The set of MCSs is created in all lower GateSets, and they are all added to the set of negated MCSs. Each node is put into an MCS and added to the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is $\prod i\ (1 - p_i)$.

Table 12 ResultGateSet & ConnectorGateSet Analysis

ResultGateSet & ConnectorGateSet	Description
Logical Function	The *ResultGateSet* and *ConnectorGateSet* represent outports, inports, and their instances and only one input is allowed. They evaluate to true if the input is true and the negated version evaluates to true if the input is false.
Creating MCSs	The set of MCSs is created in the lower GateSet and added to the set of MCSs. If no lower GateSet exists, then the node is put into an MCS and added to the set of MCSs.
Creating negated MCSs	The set of negated MCSs is created in the lower GateSet and added to the set of negated MCSs. If no lower GateSet exists, then the node is negated and put into an MCS and added to the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is p_1.

Table 13 ANDSet Analysis

ANDSet	Description
Logical Function	The *ANDSet* represents an AND gate, so it evaluates to true if all inputs are true. The negated AND gate evaluates to true if any input is false.
Creating MCSs	The set of MCSs is created in all lower GateSets. All nodes are put into one MCS. This MCS is added to the set of MCSs. If the *ANDSet* has no nodes, then it adds the set of MCSs of the first lower GateSet to the set of MCSs. Then, the set of MCSs of each lower GateSet is merged with the existing MCSs in the set of MCSs.
Creating negated MCSs	The set of negated MCSs is created in all lower GateSets, and they are all added to the set of negated MCSs. Each node is negated and put into a separate MCS. These MCSs are added to the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is $\prod i$ (p_i).

Table 14 ORSet Analysis

ORSet	Description
Logical Function	The *ORSet* represents an OR gate, so it evaluates to true if any input is true. The negated OR gate evaluates to true if all inputs are false.
Creating MCSs	The set of MCSs is created in all lower GateSets, and they are all added to the set of MCSs. Each node is put into a separate MCS. These MCSs are added to the set of MCSs.
Creating negated MCSs	The set of negated MCSs is created in all lower GateSets. All nodes are negated and put into one MCS. This MCS is added to the set of negated MCSs. If the *ORSet* has no nodes, then it adds the set of negated MCSs of the first lower GateSet to the set of negated MCSs. Then, the set of negated MCSs of each lower GateSet is merged with the existing MCSs in the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is $1 - \prod i$ ($1 - p_i$).

Table 15 XORSet Analysis

XORSet	**Description**
Logical Function	The *XORSet* represents a XOR gate, so it evaluates to true if a single input is true and all other inputs are false. The negated XOR gate evaluates to true if any two inputs are true or if all inputs are false.
Creating MCSs	The set of MCSs and the set of negated MCSs are both created in all lower GateSets. Two lists of MCS sets are created. Each node is put into a separate MCS which is then put into a separate MCS set. Therefore, an MCS set is created from every node, and these are added to the first list. Then, the same is done using negated nodes and the resulting negated MCS sets are added to the second list. The set of MCSs of each lower GateSet is added to the first list, and the set of negated MCSs of each lower GateSet is added to the second list. These two lists are ordered the same way meaning the MCS set of each node and lower GateSet is at the same position in both lists. The set of MCSs of the *XORSet* is created by merging each MCS set in the first list with all negated MCS sets in the second list except if the negated MCS set is at the same position in the list. Otherwise, the MCS set would be merged with its negated MCS set which would cause a contradiction.
Creating negated MCSs	The set of MCSs and the set of negated MCSs are both created in all lower GateSets. The same two lists as above are created. Then, every possible pair of MCS sets in the first list is merged together and added to the set of negated MCSs. Lastly, all negated MCS sets in the second list are merged and added to the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is $\sum i$ (p_i * ($\prod_{i \neq j} j$ (1 - p_j))).

Table 16 MOONSet Analysis

MOONSet	**Description**
Logical Function	The *MOONSet* represents a Combination (M out of N) gate, so it evaluates to true if at least M inputs are true. The negated Combination gate evaluates true if at least (N + 1) - M inputs are false.
Creating MCSs	The set of MCSs is created in all lower GateSets. A list of MCS sets is created the same way as the first list of the *XORSet* is created. Then, every combination of M MCS sets in the list is merged and added to the set of MCSs.
Creating negated MCSs	The set of negated MCSs is created in all lower GateSets. A list of negated MCS sets is created the same way as the second list of the *XORSet* is created. Then, every combination of (N + 1) - M negated MCS sets in the list are merged and added to the set of negated MCSs.
Quantitative Analysis	The result of the quantitative analysis is the sum of the following products. Multiplying every possible combination of M probabilities in the list with each other and with the negated probabilities of the other N - M probabilities in the list. For example, the list of probabilities { p1, p2, p3, p4 } and M = 3. The result of the quantitative analysis would be p1 * p2 * p3 * (1 – p4) + p1 * p2 * p4 * (1 – p3) + p1 * p3 * p4 * (1 – p2) + p2 * p3 * p4 * (1 – p1).

The analysis of partial CFT which were introduced in Section 5.2.4 works exactly like the analysis of CFT that are completely stored in one *ResultGateSet*. However, both sets of MCSs are always created for partial CFT. The analysis of partial CFT produces partial MCSs, and they are stored in the database. Additional analysis steps are required to obtain the full analysis result of the outport where the traversal started. The following two sections explain the storage and the additional analysis steps.

5.3.2 Storing and Reusing Analysis Results in the Neo4j Database

The prototype is partially designed to accomplish continuous analysis so that the analysis results are instantly available when the CFT model changes. The goal is to reduce waiting times for analysis and to reduce the effort for recalculation by using previous results. In Section 5.1.3, the storage format of the analysis results in the database was explained, and this section provides some more detail on the process of storage, maintenance and reuse of analysis results.

In the prototype, analysis results are reused only if the traversal is split which is described in Section 5.2.4. Splitting the traversal and returning multiple *ResultGateSets* allows more comprehensive reuse of analysis results because the results are stored at more locations in the graph and modification is less destructive to analysis results. The idea is to store analysis results at every inport instance and at every outport that was found during the traversal.

During the analysis, four types of MCSs are stored which are partial MCSs, full MCSs, and their respective negated forms. The full MCSs are only stored at outports, and they represent the full qualitative analysis results for that outport which can be directly used for analyzing higher-level CFTs. They are more volatile and often destroyed when the CFT of the outport changes. The partial MCSs are more robust to changes, but they are also less valuable because they store only partial analysis results and must be combined with other partial MCSs to create a full MCSs. The analysis after a split traversal always creates the set of MCSs and the set of negated MCSs in every GateSet, and they are always stored together. The effort for creating the additional MCS set is minimal. The benefit is more effective reuse because the analysis results are available for any negation in higher-level CFTs.

The following paragraphs describe how analysis results are stored when the prototype is used together with the EA extension that was explained in Section 4.4. In this case, the continuous analysis is activated in the EA extension, so the prototype analyzes every outport of every CFT and stores every analysis result so that they are always available. If a CFT changes then the analysis results are updated accordingly.

Before the split traversal is started on an outport a separate traversal for finding cycles which described in Section 5.2.2 is performed. If the CFT of the outport has no errors and no cycles, then the split traversal is started. After the split traversal, the qualitative analysis is performed on every returned *ResultGateSet* which produces the so-called partial MCSs. These partial MCSs are stored in the database on their respective outport or inport instance. The next step is to connect all *ResultGateSets* to one complete *ResultGateSet* for the outport where the traversal started. Then, the partial MCSs are connected to full MCSs at every outport that can be found in the connected *ResultGateSet*. The connection of *ResultGateSets* and the connection of partial MCSs to full MCSs is explained in the next section, and in the following, the assumption is that every *ResultGateSet* with an outport contains the full MCSs and the full negated MCSs for that outport. The next step is to store these full MCSs in the database at their respective outports. Then, the analysis that started at single outport is complete, and a split traversal and analysis on all remaining outports is performed by the prototype. If the outport already has stored full MCSs, then traversal and analysis are skipped because this outport was analyzed previously as part of the analysis from another outport. In the case that no full MCSs are available, but the database contains partial MCSs then these are reused when they are needed.

When a CFT is modified, the MCSs need to be adjusted depending on the structural changes in the CFT. The prototype handles changes conservatively by deleting any MCS that are directly affected by the change. The performance of this approach is lower, but there is no chance for errors. The method `deleteMCSAndResults(Node node, boolean onlyFullMCS)` removes all MCSs from the specified node, and the method is called on every node that can be reached with an outgoing failure propagation. If the parameter `onlyFullMCS` is true, then only full MCSs and full negated MCSs are removed from the node. If the node is an outport, then the method is called on any outport instance but only deleting the full MCSs, and full negated MCSs because outport does not have local effects on other CFTs. If the node is an inport instance, then the method is called on every outport instance of the same CFT instance. Again, these inport instances do not have local effects, and only full MCSs and full negated MCSs are deleted.

Often more MCSs are deleted than necessary, but they are recalculated as soon as possible. The benefits of reuse can be seen when the prototype is used together with the EA extension because then the prototype keeps recalculating the analysis results whenever the CFT model changes. Therefore, deleting too many MCSs is a smaller issue because the process runs in the background and the user is not bothered by waiting times.

5.3.3 Connecting Traversal and Analysis Results

If the traversal was split, then there are multiple *ResultGateSets*. The analysis is performed on each *ResultGateSet*, but each analysis result only represents part of the complete analysis that a full traversal and analysis would produce. Therefore, to get a complete analysis result, the GateSets must be connected. Then, their MCSs can be connected to return the complete analysis result.

The method `connectGateSets(LinkedList<HashMap<Node, Node>> inportMapList)` is used to connect the *ResultGateSets*. *AbstractGateSet* implements this method, and *ConnectorGateSet* overrides this implementation. The method is first called on the *ResultGateSet* that has the starting outport of the traversal as a start node. The parameter is initially null. The idea is that the method is called on lower GateSets and if a GateSet contains nodes that are outports or inports these nodes are transformed into the *ResultGateSets* that were created by the split traversal. The inports can be transformed into GateSets of inport instances by using the inport map list from the parameter of the method. The *ConnectorGateSet* is the only GateSet that can have an outport as a part of its node set. The *ConnectorGateSet* also stores an inport map that was created during the split traversal. The *ConnectorGateSet* replaces its outport with the *ResultGateSet* for that outport and adds its inport map to the list of inport maps. The method to connect GateSets is then called on the *ResultGateSet* with the extended list of inport maps. After the connection method finishes, all *ResultGateSets* are connected. The next step is to connect the MCSs and store the connected MCSs of each *ResultGateSet* that represents an outport. The method `connectMCS()` is used to create the connected MCSs. The method can only be used on *ResultGateSets* and *ConnectorGateSets*. It connects MCSs and also negated MCSs. The idea of the method is to replace MCSs in the MCS sets that already exist from the analysis on the separated *ResultGateSets*. If the method is used on a *ResultGateSet*, then it finds all *ConnectorGateSets* that represent outport instances in the lower GateSets. It calls `connectMCS()` on all these *ConnectorGateSets* and replaces all outport instances that are in any MCSs with the set of MCSs from the corresponding *ConnectorGateSet*. Once all outport instances have been replaced in the MCSs and negated MCSs, the *ResultGateSet* has calculated the full MCSs and the full negated MCSs. Therefore, a full analysis result is calculated for the *ResultGateSet*, and if it represents an outport, then the full MCSs and the full negated MCSs are stored in the Neo4j database. If the method `connectMCS()` is used on the *ConnectorGateSet*, it replaces all inports in all MCSs with the MCS sets of the corresponding *ResultGateSets* which represent the inport instances. The method `connectMCS()` is called on all these *ResultGateSets*, so their MCSs are connected.

5.3.4 Performing the Analysis

The class *MainTraversal* starts the analysis and depending on the settings different traversals are used which decide the approach of the analysis.
The full traversal of the class *ManualTraversal* and the full traversal of the class *Neo4jTraversal* return a single *ResultGateSet*. The analysis, whether it is qualitative or quantitative, is started at that *ResultGateSet* and the results are stored in *MainTraversal*. In that case, the analysis results are not stored in the Neo4j database.
If the split traversal of the class *Neo4jTraversal* is used on an outport, then the database is checked to see if the analysis result already exists and if it does then analysis finishes, and the result is available in *MainTraversal*. In the case, that analysis results are not available the analysis continues by starting an analysis on all *ResultGateSets* that were created by the split traversal. If any results are available in the database, they are used. Then, the *ResultGateSets* are connected, and after that, the MCSs are connected.
In the following figure, an example is shown of a CFT with all five gates. An analysis is performed on this CFT to show how the prototype executes the analysis.

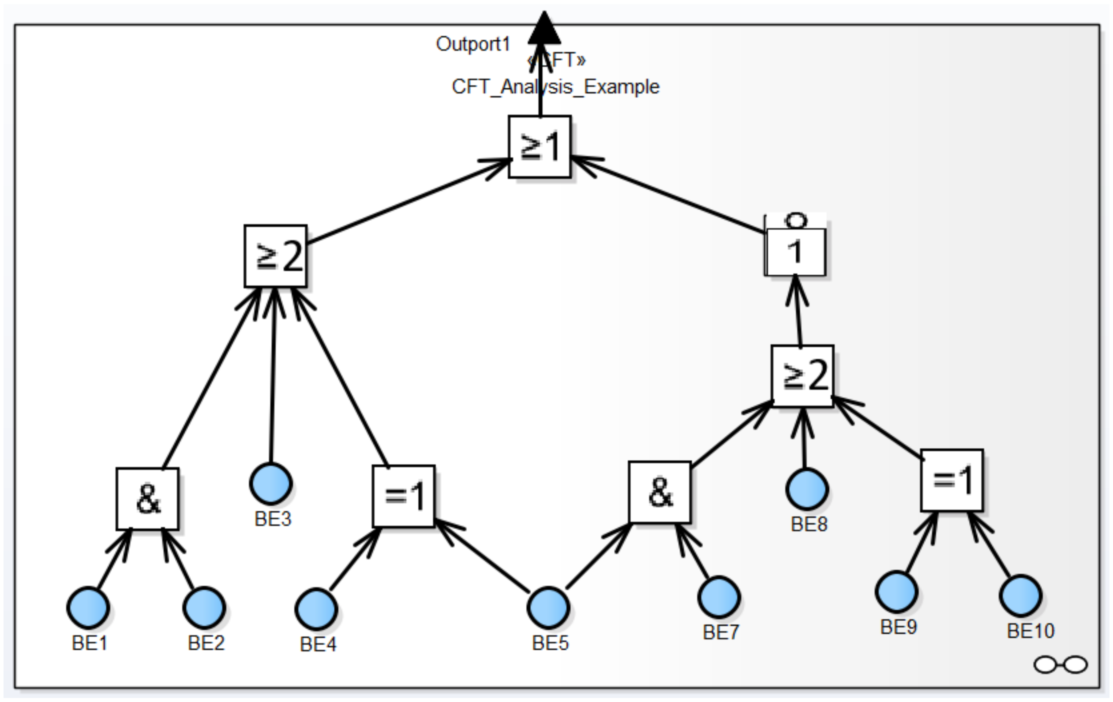


Figure 31 CFT Analysis Example

The first step of the analysis is the traversal which creates the GateSets. Below a representation of the GateSets in the prototype after the traversal can be seen.

```
CFT_Outport GateSet | Outport1
GateSets:
  CFT_OR_Gate GateSet | OR1
   GateSets:
    CFT_MOON_Gate GateSet | MOON1
     NodeSet: {BE3}
     GateSets:
      CFT_XOR_Gate GateSet | XOR1
       NodeSet: {BE4, BE5}
      CFT_AND_Gate GateSet | AND1
       NodeSet: {BE2, BE1}
    CFT_NOT_Gate GateSet | NOT1
     GateSets:
      CFT_MOON_Gate GateSet | MOON2
       NodeSet: {BE8}
       GateSets:
        CFT_AND_Gate GateSet | AND2
         NodeSet: {BE5, BE7}
        CFT_XOR_Gate GateSet | XOR2
         NodeSet: {BE9, BE10}
```

Now the qualitative analysis can be performed. The following results are ordered from lowest to highest GateSet to see how the MCS sets evolve. Each MCS is represented as a set in the set of MCS.

```
Set of MCS (AND1): { {BE1, BE2} }
Set of MCS (XOR1): { {BE4, ¬BE5}, {¬BE4, BE5} }

Set of negated MCS (AND2): { {¬BE5}, {¬BE7} }
Set of negated MCS (XOR2): { {BE9, BE10}, {¬BE9, ¬BE10} }

Set of MCS (MOON1):
{ {BE1, BE2, BE3}, {BE1, BE2, B4, ¬BE5}, {BE1, BE2, ¬B4, BE5}, {BE3, BE4, ¬BE5},
{BE3, ¬BE4, BE5} }

Set of negated MCS (MOON2) / Set of MCS (NOT1):
{ {¬BE5, ¬BE8}, {¬BE7, ¬BE8}, {¬BE5, BE9, BE10}, {¬BE5, ¬BE9, ¬BE10}, {¬BE7, BE9,
BE10}, {¬BE7, ¬BE9, ¬BE10}, {¬BE8, BE9, BE10}, {¬BE8, ¬BE9, ¬BE10} }

Set of MCS (OR1) / Set of MCS (Outport1):
All MCS from MOON1 and NOT1 in one MCS set.
```

The example has hidden prime implicants which can be calculated by the prototype. The calculation of prime implicants in the prototype is basic and not very efficient. It finds all nodes that are negated in one MCS and unnegated in another MCS. Any such pair of MCSs is copied, and the respective node is removed in both sets. Then one MCS is added to the other MCS, and the resulting MCS is checked if it is always false. An MCS is considered to be always false if it contains the same node in negated and unnegated form. If a resulting MCS is always true, then it is discarded. Otherwise, it is added to the set of MCSs. Any resulting MCSs that are not minimal are also discarded. Hidden prime implicants are searched until the no additional prime implicants can be found. That means that hidden prime implicants that are created with previously found hidden prime implicants are also found. The hidden prime implicants of the example are shown below.

```
Hidden Prime Implicants:
{ {BE1, BE2, ¬BE4, ¬BE8}, {BE3, ¬BE4, ¬BE8}, {BE1, BE2, ¬BE4, BE9, BE10}, {BE1,
BE2, ¬BE4, ¬BE9, ¬BE10}, {BE3, ¬BE4, BE9, BE10}, {BE3, ¬BE4, ¬BE9, ¬BE10} }
```

These prime implicants are added to the MCS set of the outport. Then all necessary prime implicants are calculated, and the qualitative analysis is complete.

6 Evaluation of the Prototype

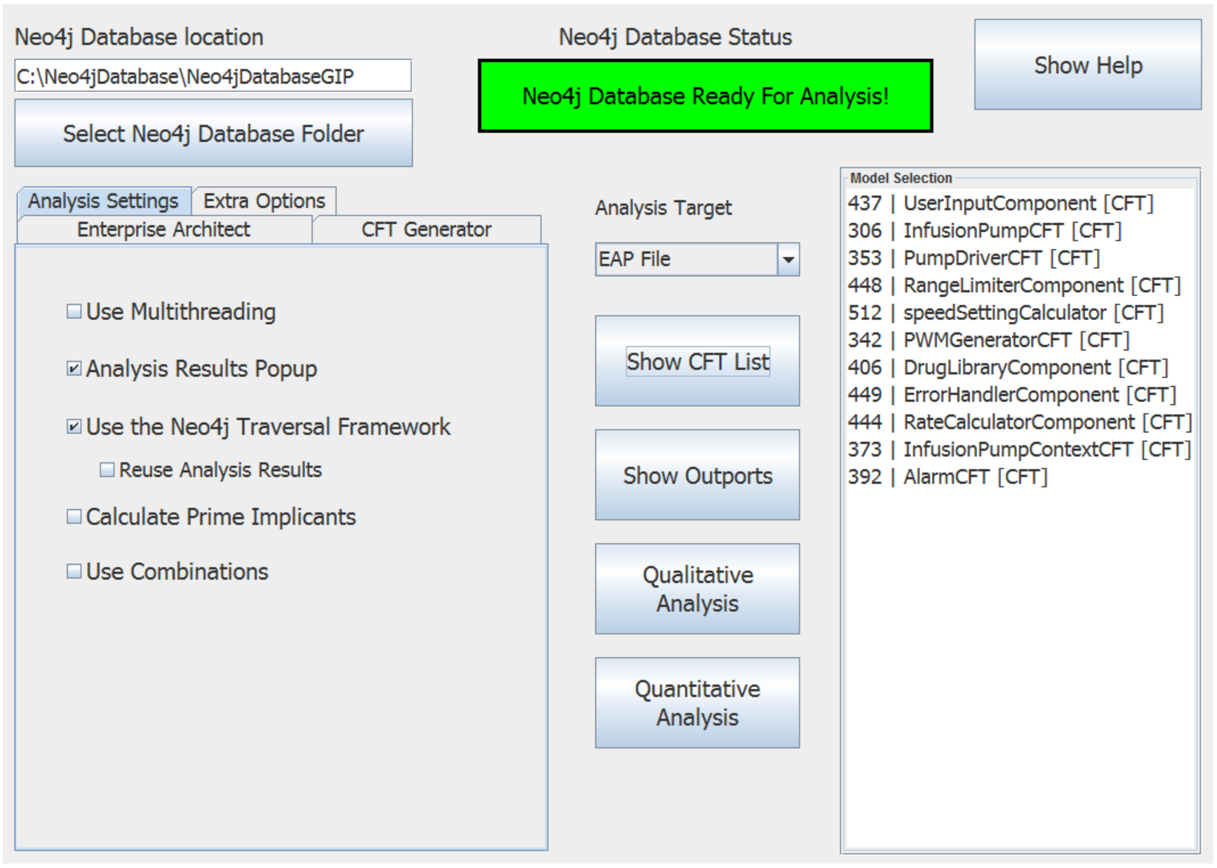
In the previous section, the implementation of the analysis was explained, and in this section, the prototype is evaluated. In Section 6.1, the GUI of the prototype is presented, and the main functionality is described. Then, in Section 6.2, the analysis of the prototype is evaluated according to the goals of this work. Section 6.3 discusses some limitations of the prototype and how to overcome them.

6.1 Using the final Prototype

The final prototype has a GUI to perform the analysis. In Section 4, the update process of the prototype was already shown. The following subsections assume that the Neo4j database is ready for analysis. This section presents the analysis process and explains all different analysis options. In the end, some additional functionality of the prototype which available in the GUI is presented.

6.1.1 Using the GUI for Analysis

The GUI of the prototype standalone application is designed similar to the Analysis Window for the EA Extension which was presented in Section 4.4.3. It has the same analysis options. In Figure 32, the relevant parts of the GUI are shown. The figure shows the GUI in a state where it is connected to a Neo4j database that contains CFTs. In addition to that the button “Show CFT List” was clicked to display all CFTs in the Neo4j database in the right-hand text window “Model Selection”.



Neo4j Database location: C:\Neo4jDatabase\Neo4jDatabaseGIP

Neo4j Database Status: Neo4j Database Ready For Analysis!

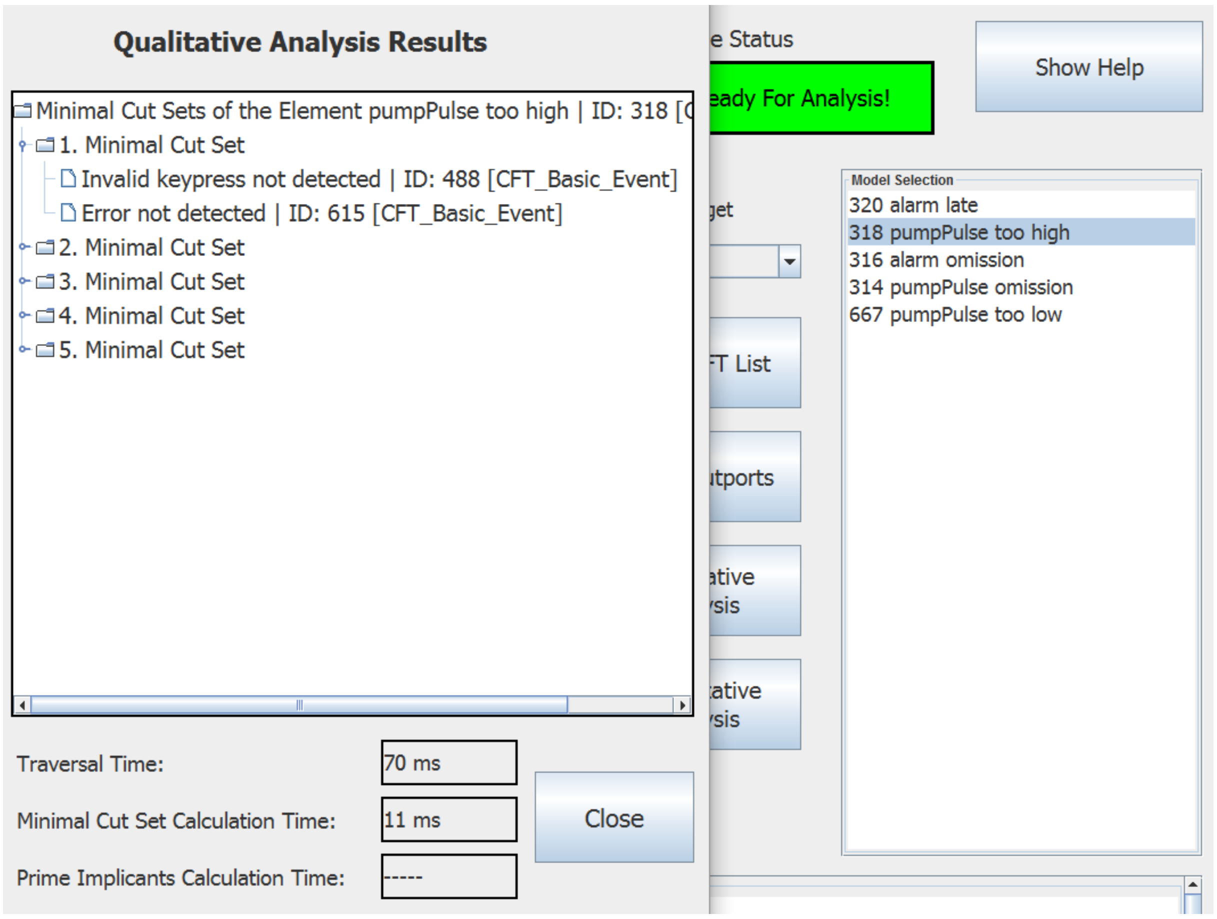
☐ Use Multithreading
☑ Analysis Results Popup
☑ Use the Neo4j Traversal Framework
  ☐ Reuse Analysis Results
☐ Calculate Prime Implicants
☐ Use Combinations

Analysis Target: EAP File

Model Selection
437 | UserInputComponent [CFT]
306 | InfusionPumpCFT [CFT]
353 | PumpDriverCFT [CFT]
448 | RangeLimiterComponent [CFT]
512 | speedSettingCalculator [CFT]
342 | PWMGeneratorCFT [CFT]
406 | DrugLibraryComponent [CFT]
449 | ErrorHandlerComponent [CFT]
444 | RateCalculatorComponent [CFT]
373 | InfusionPumpContextCFT [CFT]
392 | AlarmCFT [CFT]

Figure 32 Prototype GUI showing CFTs

The CFTs in the “Model Selection” can be selected by clicking on them and when a CFT is selected the button “Show Outports” can be used to display all outports of the selected CFT. Then, the analysis can be performed on a selected outport by clicking “Qualitative Analysis” or “Quantitative Analysis” depending on which analysis should be performed. If the checkbox for “Analysis Results Popup” is selected, then the completed analysis opens a separated window that displays the results. Figure 33 shows this result windows for qualitative analysis.



Qualitative Analysis Results

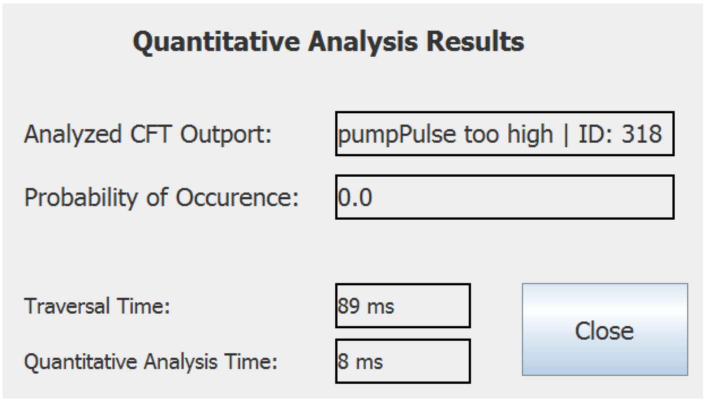
Minimal Cut Sets of the Element pumpPulse too high | ID: 318 [C
1. Minimal Cut Set
Invalid keypress not detected | ID: 488 [CFT_Basic_Event]
Error not detected | ID: 615 [CFT_Basic_Event]
2. Minimal Cut Set
3. Minimal Cut Set
4. Minimal Cut Set
5. Minimal Cut Set

Traversal Time: 70 ms
Minimal Cut Set Calculation Time: 11 ms
Prime Implicants Calculation Time: -----

Model Selection
320 alarm late
318 pumpPulse too high
316 alarm omission
314 pumpPulse omission
667 pumpPulse too low

Figure 33 Qualitative Analysis Result Window

The results window shows the calculation times for different parts of the analysis. Figure 34 shows the results window for a quantitative analysis of the same element. The quantitative analysis result is 0.0 because not all elements in the CFT had defined failure probabilities.



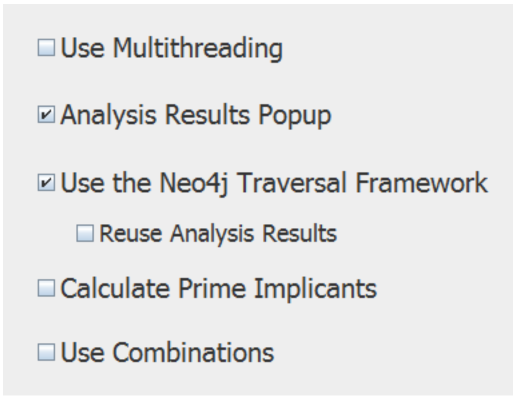
Quantitative Analysis Results

Analyzed CFT Outport: pumpPulse too high | ID: 318
Probability of Occurence: 0.0

Traversal Time: 89 ms
Quantitative Analysis Time: 8 ms

Figure 34 Quantitative Analysis Results Window

All analysis options are shown in Figure 35 and are explained in the following.



☐ Use Multithreading

☑ Analysis Results Popup

☑ Use the Neo4j Traversal Framework

☐ Reuse Analysis Results

☐ Calculate Prime Implicants

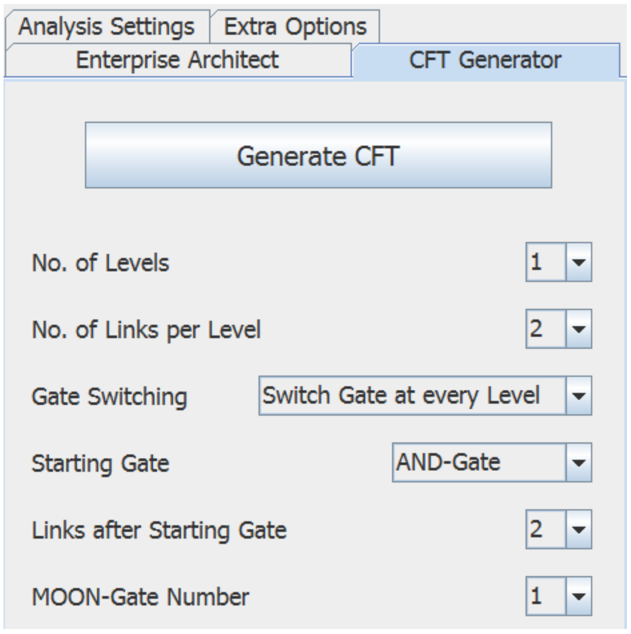
☐ Use Combinations

Figure 35 Analysis Options in the Prototype

If the “Use Multithreading” checkbox is selected, then the prototype uses multithreading whenever possible. However, the benefits of multithreading are only measurable in large CFTs. If the complete analysis takes only a few hundred milliseconds, then the option for multithreading may be even slightly slower because the setup for multithreading requires some time. If the “Analysis Results Popup” is selected, then the Analysis results are presented in a separate window. However, analysis results along with some other information are always displayed in the “Outport Console” at the bottom of the GUI. If “Use the Neo4j Traversal Framework” is selected the traversal is handled by the class *Neo4jTraversal* instead of the class *ManualTraversal*. If “Reuse Analysis Results” is selected then the split traversal in the class *Neo4jTraversal* is used, and analysis results are stored in the Neo4j database as well as reused if they are available. Otherwise, the full traversal of the class *Neo4jTraversal* is used to traverse the CFT. If the “Calculate Prime Implicants” is selected then the prototype will find any hidden prime implicants in the qualitative analysis results and add them to the results. Otherwise, hidden prime implicants are ignored. If “Use Combinations” is selected then the prototype combines merged MCSs at the end of the analysis. Otherwise, merged MCSs are combined whenever a GateSet finishes its analysis.

6.1.2 Additional Functionality of the GUI

The prototype has some additional functionality besides the analysis that can be accessed by the GUI. These are used for testing purposes or convenience operations that assist in using the prototype. The first functionality is the CFT generator that generates simple CFTs in the Neo4j database. These can be used for testing the performance of the prototype as well as how much memory is used for traversals and analysis. The options of the CFT generator are displayed in Figure 36.



No. of Levels: 1
No. of Links per Level: 2
Gate Switching: Switch Gate at every Level
Starting Gate: AND-Gate
Links after Starting Gate: 2
MOON-Gate Number: 1

Figure 36 CFT Generator in the GUI

The CFTs that can be generated are only a single CFT with one outport and only basic events at the end of each branch. Therefore, it is essentially a fault tree (FT) instead of a CFT but it uses the notation of a CFT, and all FTs are stored as CFTs in the database. The first gate after the outport is variable, but all other gates are AND and OR gates. The number of levels is the depth of the CFT which is the number of failure propagations between the first gate and the basic events and the bottom of the CFT. The number of links per level is the number of incoming failure propagations that each gate has in a level. The gate switching determines how AND and OR gates are switched. They can be switched at every level or switched at every link. The starting gate can be set to AND, OR, NOT, MOON, and XOR gate. The links after the starting gate can be set differently from the number of links per level. The “MOON-Gate Number” is only relevant if the starting gate is a MOON gate and this option sets the number of incoming links that must be true for the MOON gate to evaluate to true. The idea of the CFT generator is to produce a CFT that has a lot of minimal cut sets so that the analysis has to work with many minimal cut sets. Therefore, the qualitative analysis cannot calculate results in a reasonable time for generated CFT that have too many levels. The generated CFT can be analyzed by changing the “Analysis Target” seen in Figure 32. Other functionality that is worth mentioning is displayed in Figure 37.

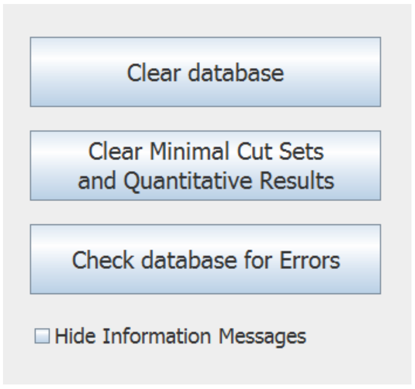

Clear database
Clear Minimal Cut Sets and Quantitative Results
Check database for Errors
☐ Hide Information Messages

Figure 37 Extra functionality

The button "Clear database" clears the entire database. The button "Clear Minimal Cut Sets and Quantitative Results" removes all analysis results that were previously stored in the database. The button "Check database for Errors" checks if there are any errors in the CFT model or if the analysis results are stored incorrectly. These buttons are very useful if the code is changed to check if the prototype is still working correctly.

6.2 Evaluation of the Prototype

The prototype was developed to show how the Neo4j database can be used for qualitative analysis of CFTs. The idea was to use the Neo4j Traversal Framework to traverse CFTs and use the traversal results for the analysis. The prototype can store analysis results and reuse them in later analysis. The benefits and drawbacks of the prototype are described in the following subsections. The prototype implements multiple approaches, and they are compared to each other in terms of performance and other benefits.

6.2.1 Evaluation of the Analysis without Reuse

The prototype implements two traversals that do not store analysis results in the Neo4j database. These two traversals are full traversals and were implemented for comparison with the split traversal that reuses analysis results. The idea was to traverse the CFT completely and to return a result that could be used directly for a full analysis. The goal of the two implementations was to find out if the Neo4j Traversal Framework was fast enough to compete with the performance of the manual traversal which only used basic methods of the Neo4j API. In various tests, the Neo4j Traversal Framework was compared with the performance of the manual traversal. The traversal performance was tested on a simple CFT with one outport and no inports. The CFT was generated by the CFT generator of the prototype. The generated CFT has 10 gate levels, and each gate has 3 incoming failure propagations. Therefore, the CFT has a total of 88575 nodes. There is one node for the CFT and one node for the outport. Then there are $3^0 + 3^1 + 3^2 + 3^3 + 3^4 + 3^5 + 3^6 + 3^7 + 3^8 + 3^9 = 29523$ gates and $3^{10} = 59049$ basic events. The first two gate levels are AND gates, and the following gate levels switch between AND and OR gates starting with OR gates at level 3. In the following, the average time for traversals is presented and discussed. Each traversal was performed 100 times, and the average was calculated. The traversal time includes the creation of all GateSets.

The ManualTraversal without multithreading was the slowest with an average traversal time of 350ms. The single threaded full Neo4jTraversal had an average traversal time of 290ms. The multithreaded ManualTraversal reached the best average traversal time with 250ms. These results were expected because the Neo4j Traversal Framework is very efficient for single threaded traversal. However, it cannot reach the performance of multithreaded manual traversal. When the size of the CFT is significantly reduced, the average traversal time is so low that the differences between the traversals are insignificant. Therefore, the Neo4j traversal is the best overall option because it does not require multithreading or recursion which can increase memory usage.

The analysis is the same after both traversals because they both produce the same GateSets. Since the prototype does not reuse results for these traversals, the CFT is analyzed completely from scratch. However, the *ResultGateSet* is complete for the outport and can be analyzed directly. The analysis performance varies depending on which analysis options are activated in the prototype. The analysis performance tests were conducted with a generated CFT that had 5 levels and 2 links per level resulting in 32 basic events. The analysis for this CFT returns 4096 minimal cut sets. The test was repeated 20 times with and without multithreading. Without multithreading, the average execution time of the qualitative analysis was 7312ms. With multithreading, the average execution time of the qualitative analysis was 4376ms. Therefore, the performance is about 40% better when multithreading was activated on the tested system. The combination of MCSs which was explained in Section 5.1.2 had only a minor average performance benefit of 100ms regardless if multithreading was active or not. The combination of MCSs improves performance significantly if the number of minimal cut sets is above 10000 but the overall analysis time becomes unreasonable if the number of prime implicants is that high. Overall, the prototype showed that intelligent use of multithreading improves performance significantly, but the single-threaded traversal with the Neo4j Traversal Framework is fast enough in practical applications.

6.2.2 Evaluation of the Analysis with Reuse

The part of the prototype that implements qualitative analysis with reuse has more practical uses than the other parts of the prototype. The Neo4j database was used to efficiently store analysis results at the CFT elements. During the analysis, these results are reused which improves the performance. This approach uses a different traversal compared to the previous approaches because the analysis is performed on different parts of the CFT. The idea is to analyze as many parts of the CFT as possible and store their results. These results are then combined to full analysis results for every outport that was found during the traversal. The full analysis results are also saved because they can be reused more efficiently. If the full analysis result is already available for an outport, then the traversal and analysis can be skipped which improves performance. However, the performance is not a primary goal for this approach because the intent is that this analysis is performed continuously in the background of another application like EA. The EA extension communicates with the prototype, and the analysis is performed whenever the CFT is modified in EA. Therefore, the user is not affected by analysis performance. In normal use cases where the CFT is not extremely large and complex, the analysis is completed before the user opens the analysis window to check the analysis results for the CFT. Tests showed that the initial analysis with reuse is about 50% slower than the analysis without reuse because it requires many more steps to perform the analysis results and store them in the database. However, once the database contains the analysis results, the analysis is consistently faster than the analysis without reuse. If the CFT is modified slightly, the analysis with reuse is still faster in most cases, but with more extensive changes the performance benefit is non-existent, or the analysis with reuse is slower. The reasons for that are the small CFTs that were tested. A real performance benefit can only be achieved if the CFT contains many CFT instances which are very complex. In smaller CFTs, the benefit of saving the results is minimal because the analysis is so fast that it does not matter if the results are directly available. In addition to that, the analysis with reuse is not optimized in the prototype. Therefore, the analysis results are not stored intelligently at specific outports but at every outport and inport instance. The traversal is also slower because it has to be restarted at every outport that is found. Therefore, the set-up time for each traversal affects the performance more significantly. For practical use of the prototype, the storage of analysis results requires some rework, so that results are stored more intelligently and that the traversal performance is better.

Overall, the prototype showed how the Neo4j database and the Neo4j Traversal Framework could be used in unison with fault tree analysis and how analysis results can be reused for the analysis of different parts of a CFT. However, the prototype has a few limitations that need to be reviewed before it is evaluated for practical use. The next section describes some of these limitations and provides possible solutions to them.

6.3 Limitations of the Prototype

The goal of the prototype was to show the benefits of using a Neo4j database for qualitative analysis. A simple quantitative analysis was also implemented, but in practice, more advanced quantitative analysis is required. The first subsection explains how a more advanced quantitative analysis can be realized in the prototype and how failure rates could be reused. The second subsection describes possible improvements to the prototype with regards to performance.

6.3.1 Practical Quantitative Analysis

The prototype does not implement quantitative analysis in a practical form and only calculates a basic failure probability. Also, repeated events are not considered in the quantitative analysis of the prototype. In practice, the quantitative analysis uses failure rates over time to get a realistic approximation of the reliability of the component that is modeled with the CFT. This form of analysis is very complex and modern tools generate BDDs and use failure rate sampling to calculate the reliability of the outport with a high level of accuracy. The prototype was not implemented to perform this task, but if a Neo4j database is considered for further application in safety analysis, a practical quantitative analysis is essential. However, then reuse of analysis results becomes much more complex and new approaches might be necessary to get any benefit from reuse.
One of the most efficient techniques for quantitative analysis of fault trees (FTs) is the use of binary decision diagrams (BDDs) [3]. Since the algorithms for the quantitative analysis on BDDs are already available, the idea would be to implement a prototype that stores the BDD of the FT or CFT and updates it when it changes. There are many aspects that would have to be considered for such an approach because changes in the BDD would affect the reduced ordered BDD (ROBDD) which is used for the analysis. The idea would be to find efficient ways to update the ROBDD instead of completely recreating the BDD and reducing it afterward. Such an approach can be implemented in the Neo4j database by connecting the basic events to a BDD with special relationships. The ROBDD can be found by using the Neo4j Traversal Framework to reduce the BDD and find efficient variable orders. The ROBDD could be maintained with intelligent traversal that can react to modification in the CFT. However, such an implementation is beyond the scope of this work.

6.3.2 Performance Considerations

The prototype uses various techniques to improve the performance of the analysis, but the main focus was not on maximizing the performance. The multithreading can be extended to other parts of the prototype that can be run in parallel. The prototype only uses multithreading in places where the benefits outweigh the drawbacks, but there are some areas where multithreading can achieve better performance. Multithreading was avoided when the implementation was too complex and error prone. The connection of GateSets and especially the connection of MCSs can be performed much faster if multithreading is used. The problem is that it is hard to determine where different threads might come into conflict.

Next, the concept of the MCS implementation in the prototype could be improved to allow better multithreading and improve the speed of set operations. The idea would be to create an MCS where it is simple to replace elements with other MCSs and allow concurrent modifications on different elements of the MCS so that different threads can modify the MCS at the same time. A more specialized implementation of the set interface could achieve this goal. Besides that, the concept of MCS could be completely discarded, and instead, a representation of a logical formula could be used to represent the analysis results. These could be handled and stored more easily, but the disadvantage would be that the prime implicants would not be clearly visible and would have to be calculated every time.

Another way to improve the performance would be to implement a better storage of analysis results. The prototype deletes a lot of analysis results every time the CFT is modified. Instead of removal the analysis results could be maintained and updated by checking how the changes in the CFT affect the analysis results. For example, adding a NOT gate somewhere in the CFT would only switch the set of MCSs with the set of negated MCSs in higher-level CFT elements.

There might be other ways to improve the performance by rethinking the reuse of analysis results in the prototype. For example, the analysis results of smaller CFT do not need to be stored because their analysis is fast and there is no reason to store their results. In case the CFT is very large then it might improve performance to store analysis results at more parts of the CFT. In Section 7 some additional ideas for performance improvement are described. Overall, the performance of the prototype has much room for improvement, and the Neo4j database has more potential if it is used efficiently.

7 Conclusion and further work

In this work, the fundamentals of fault tree analysis were presented, and the most common analysis techniques were explained. The component fault tree (CFT) and the component-integrated component fault tree (C^2FT) were introduced, and the benefits over conventional fault trees were described. The basics of the Neo4j graph database were presented, and the main concepts were discussed. The transfer of CFTs from a relational database to the Neo4j database was explained in detail. All concepts of the traversal and analysis in the prototype were presented and illustrated with various examples. Lastly, the prototype was evaluated, and some of the limitations were discussed.

The goal of this work was the implementation of a prototype that utilizes the Neo4j Traversal Framework to improve the performance of fault tree analysis. This work describes the steps that are necessary to transfer CFTs from a relational database to the graph database Neo4j and how the CFTs can be structured efficiently for analysis. The analysis described in this work uses different approaches, and the benefits and drawbacks of these approaches were discussed. The prototype demonstrated how the Neo4j database could be used to store analysis results efficiently and reuse them when the CFT is modified. The implementation of a practical quantitative analysis was discussed as well as some ideas for further performance improvement of the prototype. The Neo4j database is well suited for graph-based safety analysis, and the traversal implementation is very intuitive.

Further work should be focused on implementing a more practical quantitative analysis in the prototype as well as testing the viability of the Neo4j database for more complex safety analysis like Markov models or Petri Nets [1]. These models can be easily stored in a Neo4j database, and special traversal algorithms could be developed for the analysis of such models. The prototype could be extended to include more analysis techniques such as binary decision diagrams (BDDs) which can improve the performance further. Additionally, the generator for CFTs should be extended, so that more practical CFTs can be created that can be used to optimize the prototype for different scenarios. The continuous analysis of the prototype requires some optimization to be competitive in practical applications. For example, the storage of analysis results should be limited to places where they provide performance benefits. The concept of the split traversal should be reworked to offer more flexibility and performance. This could be achieved by performing traversals that determine the structure of a CFT and then decide which form of traversal is best suited for the analysis of that CFT.

Overall, this work showed that the Neo4j Traversal Framework offers great potential for graph-oriented safety analysis. The Neo4j database as a backend provides many benefits over a relational database and is easy to integrate into an application that performs safety analysis.

References

[1] Safety and Reliability of Embedded Systems Lecture, TU Kaiserslautern, 2013 http://seda.informatik.uni-kl.de/teaching/suze/ws2013/ (accessed: 15.03.2017)

[2] Fault Tree Handbook with Aerospace Applications, NASA, 2002 https://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf (accessed: 15.03.2017)

[3] Sinnamon, Roslyn M., “Binary decision diagrams for fault tree analysis”, Doctoral Thesis, Roslyn Mary Sinnamon, 1996 https://dspace.lboro.ac.uk/2134/7424 (accessed: 15.03.2017)

[4] B. Bollig, I. Wegener, “Improving the variable ordering of OBDDs is NP-complete” in IEEE Transactions on Computers, Volume: 45, Issue: 9, Sep 1996, pp. 993 – 1002 http://dx.doi.org/10.1109/12.537122 (accessed: 15.03.2017)

[5] B. Kaiser, P. Liggesmeyer and O. Mäckel, “A new component concept for fault trees”, in 8th Australian Workshop on Industrial Experience with Safety Critical Systems and Software – SCS 2003.

[6] D. Domis, M. Trapp, "Integrating Safety Analyses and Component-Based Design", in *Computer Safety, Reliability, and Security*, M. Harrison, M. Sujan, Eds., Berlin / Heidelberg, Germany: Springer, 2008, pp. 58-71.

[7] D. Domis, M. Trapp, "Component-Based Abstraction in Fault Tree Analysis", in *Computer Safety, Reliability, and Security*, B. Buth, G. Rabe, T. Seyfarth, Eds., Berlin: Springer, 2009, pp. 297-310.

[8] Enterprise Architect 12 A Reviewer’s Guide, Sparx Systems, 2015 http://www.sparxsystems.com/downloads/whitepapers/EAReviewersGuide.pdf (accessed: 15.03.2017)

[9] R. Adler et al., "Integration of Component Fault Trees into the UML", in Models in Software Engineering, J. Dingel, A. Solberg, Eds., Berlin / Heidelberg, Germany: Springer, 2011, pp. 312-327.

[10] Michael Hunger, E-Book Neo4j 2.0 Eine Graphendatenbank für alle https://neo4j.com/neo4j-graphdatenbank-book/ (accessed: 15.03.2017)

[11] The Neo4j Developer Manual v3.1, Neo4j Team, 2017
http://neo4j.com/docs/developer-manual/current/ (accessed: 15.03.2017)

[12] Enterprise Architect User Guide Series “Enterprise Architect Object Model”
http://www.sparxsystems.com/resources/user-guides/automation/enterprise-architect-object-model.pdf (accessed: 15.03.2017)

List of Figures

Figure 1 Top Event, Event, Basic Event, AND gate and OR gate....................................11
Figure 2 Fault tree example 12
Figure 3 BDDs of Part A failure | Component A failure | Part E failure 16
Figure 4 BDD and ROBDD of the top event System failure...17
Figure 5 Cause Effect Graph 19
Figure 6 CFT example 20
Figure 7 CFTs of Component A and B 20
Figure 8 CFTs of Battery and Part E 21
Figure 9 Realization and Specification..22
Figure 10 Enterprise Architect 12.1 UI 24
Figure 11 Logical Component Infusion Pump modeled with I-SafE 25
Figure 12 CFT of DrugLibraryComponent...26
Figure 13 CFT of PumpDriverComponent 26
Figure 14 A simple graph database...29
Figure 15 Neo4j Browser 33
Figure 16 Three EA Elements 40
Figure 17 Three EA Elements in Neo4j 40
Figure 18 Allowed diagram elements in EA 44
Figure 19 Alarm CFT in EA 45
Figure 20 Alarm CFT in Neo4j...45
Figure 21 Neo4j database not ready in the GUI 46
Figure 22 Neo4j database ready for update in the GUI..47
Figure 23 Neo4j database ready for analysis in the GUI 47
Figure 24 EA Extension Settings 51
Figure 25 Analysis Window for the EA Extension...52
Figure 26 The Neo4j Traversal Framework...54
Figure 27 GateSet Example 57
Figure 28 Inport with multiple Inport Instances...62
Figure 29 Partial Traversal Results Example 67
Figure 30 Partial CFT Example 68
Figure 31 CFT Analysis Example 79
Figure 32 Prototype GUI showing CFTs..82
Figure 33 Qualitative Analysis Result Window...83
Figure 34 Quantitative Analysis Results Window..83
Figure 35 Analysis Options in the Prototype...84
Figure 36 CFT Generator in the GUI...85
Figure 37 Extra functionality...86

List of Tables

Table 1 Neo4j Labels.....41
Table 2 Neo4j Labels.....42
Table 3 Neo4j Relationship Types.....42
Table 4 List of commands.....49
Table 5 GateSet Implementations.....58
Table 6 Labels for Analysis Results.....60
Table 7 Relationship Types of Analysis Results.....61
Table 8 CyclePathExpander Implementation.....64
Table 9 ManualTraversal Implementation.....66
Table 10 CFTPathExpanderSplit Actions.....69
Table 11 NOTSet Analysis.....72
Table 12 ResultGateSet & ConnectorGateSet Analysis.....72
Table 13 ANDSet Analysis.....73
Table 14 ORSet Analysis.....73
Table 15 XORSet Analysis.....74
Table 16 MOONSet Analysis.....75

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability (transaction properties)
API	Application Programming Interface
BDD	Binary Decision Diagram
BPMN	Business Process Model and Notation
CEG	Cause Effect Graph
CFT	Component Fault Tree
C^2FT	Component-Integrated Component Fault Tree
Cypher	Cypher Query Language
EA	I-SafE application and its platform Enterprise Architect
FT	Fault Tree
FTA	Fault Tree Analysis
GateSet	Traversal Result
GPLv3	General Public License Version 3
GUI	Graphical User Interface
IESE	Fraunhofer-Institut für Experimentelles Software Engineering
Inport	CFT Input Failure Mode
MCS	Prime Implicant (Minimal Cut Set that includes negated events)
Neo4j	Neo4j Graph database
Neo4j CE	Neo4j Community Edition
Outport	CFT Output Failure Mode
ROBDD	Reduced Ordered Binary Decision Diagram
SQL	Structured Query Language
SysML	Systems Modeling Language
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator (web address)