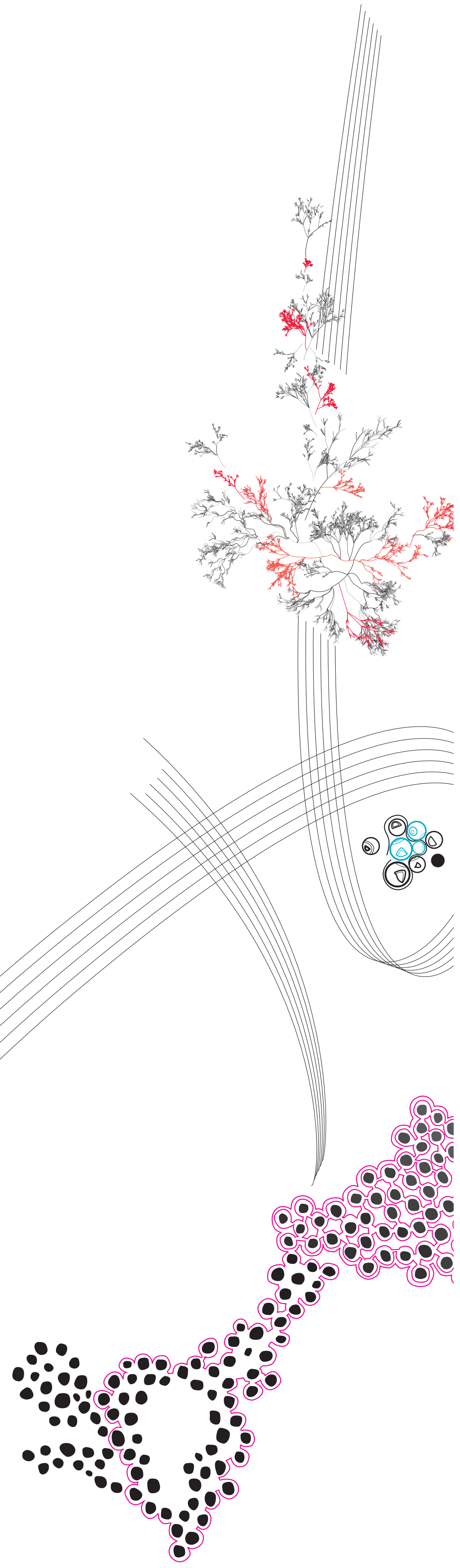


MSc Computer Science
Final Project

Automatic evaluation of fault-tree-based student exercises

Nele Budde

Committee Members:

dr. ing. Ernst Moritz Hahn (1st Supervisor)
dr. ir. Andrea Continella
prof. dr. Anne Remke

August, 2024

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

UNIVERSITY OF TWENTE.

Contents

Abstract

Fault tree analysis is a crucial concept in reliability engineering, and it plays a role in the "Software Testing and Risk Assessment" (STAR) course at the University of Twente. To enhance the teaching of fault tree analysis, students are tasked with creating fault trees based on given system descriptions. However, manually evaluating these exercises is tedious and can be challenging. This thesis aims to explore and analyze various approaches to automate the evaluation of students' fault trees by comparing them to sample solutions. The objective is to provide means of the manual assessment of fault tree exercises, thereby providing students with more opportunities for practice and feedback. This thesis focuses on three algorithmic approaches: Boolean equation comparison, edit distance comparison, and pattern matching comparison.

Chapter 1

Introduction

Fault trees present an important concept in the field of reliability engineering. To ensure that future engineers are familiar with the concept of fault trees, courses related to software engineering are taught at many universities. For example, fault tree analysis is part of the course "Software testing and Risk assessment" (STAR) [11], which is a mandatory course for the Master Program Software Technology [10] at the University of Twente.
To successfully teach the concept of fault tree analysis, one important part is that students create fault trees based on a given system description. For example, in a previous STAR tutorial, one exercise was to depict the system in Figure 1.1 as a fault tree: For the system to be functional, at least one of the Servers 1 to 5 had to function. To guarantee that a server is functioning, at least one of its cable and switch combinations has to be intact. A possible solution for this exercise, with a detailed description of intermediate events, can be seen in Figure 1.2.
However, evaluating the correctness of these types of exercises can be cumbersome and time-consuming, since there is not just a right and a wrong solution. Instead, students may submit solutions that have all the correct basic events and hierarchy, but use the wrong type of gate. Others could misinterpret the description of the given exercise and only come up with a partial solution. A third student might hand in solutions where the fault tree is correct, but not minimal, which could mean that the solution can still be im-

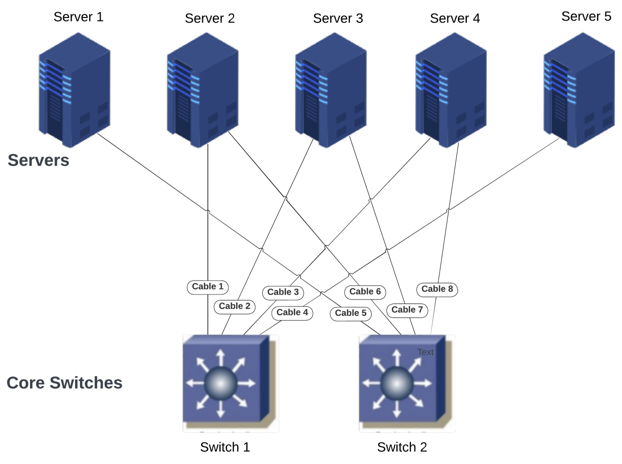


FIGURE 1.1: Example exercise from a STAR tutorial where the shown systems should be depicted into a fault tree.

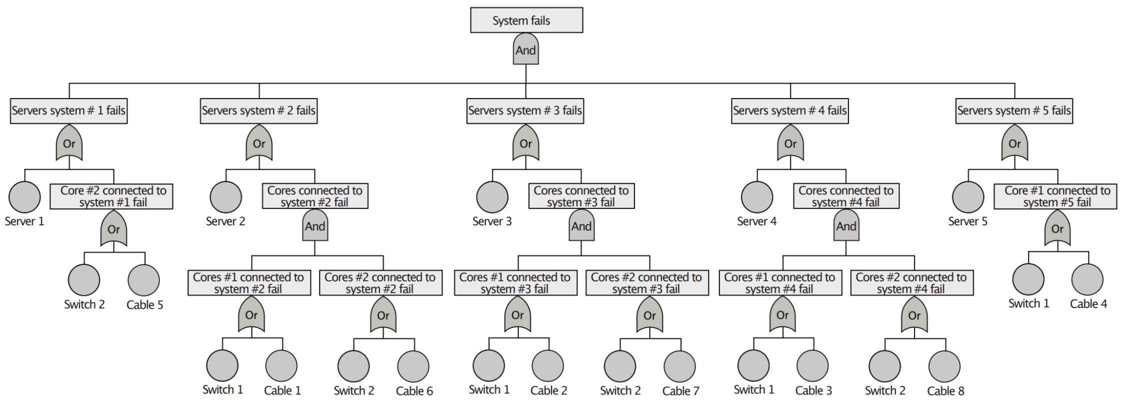

System fails
And
Servers system # 1 fails
Servers system # 2 fails
Servers system # 3 fails
Servers system # 4 fails
Servers system # 5 fails
Or
Or
Or
Or
Or
Server 1
Core #2 connected to system #1 fail
Or
Switch 2
Cable 5
Server 2
Cores connected to system #2 fail
And
Cores #1 connected to system #2 fail
Or
Switch 1
Cable 1
Cores #2 connected to system #2 fail
Or
Switch 2
Cable 6
Server 3
Cores connected to system #3 fail
And
Cores #1 connected to system #3 fail
Or
Switch 1
Cable 2
Cores #2 connected to system #3 fail
Or
Switch 2
Cable 7
Server 4
Cores connected to system #4 fail
And
Cores #1 connected to system #4 fail
Or
Switch 1
Cable 3
Cores #2 connected to system #4 fail
Or
Switch 2
Cable 8
Server 5
Core #1 connected to system #5 fail
Or
Switch 1
Cable 4

FIGURE 1.2: Solution for exercise in Figure 1.1.

proved. All of these scenarios, and more, have to be taken into account during the process of evaluation.
In addition, the manual and time-consuming evaluation process of these types of exercises means that students get only limited opportunities to test their abilities in creating fault trees. By automating this process and providing textual feedback on how their fault tree can be improved, students could use it during their studies to get even more practice. Therefore, we also want to evaluate what kind of textual feedback can be gained from comparing two fault trees, and how useful it might be to help the student to understand their mistakes.
Based on an interview with the organiser of the STAR tutorials we learned that those evaluation processes are currently performed completely manually. Usually, it is more common for students to submit a supplementary text with their solution, to argue why their fault tree is correct and to explain their approach, instead of just submitting the fault tree. Then, the evaluation is made by different teaching assistants based on the provided description rather than on the fault tree itself. Regarding the feedback for such exercises, students receive some comments for their overall assignment. If this is not sufficient, they can actively ask for feedback during the tutorials by raising questions. Regarding the most important requirement, so that such a tool will benefit the course, the interviewee mentioned that it would be important that the tool supports static and dynamic fault trees: Static fault trees represent system failures using only fixed, logical relationships between events, while dynamic fault trees incorporate time-dependent behaviours as well. Additionally, it should provide feedback so that students can use it as an additional chance to practise their skills.
Hence, the main purpose of this thesis is to research how fault tree-based student exercises could be evaluated automatically, in a form where a student's drawing of a fault tree is compared to a given sample solution. The tool should support dynamic fault trees and provide textual feedback so that the students can understand which mistakes they made and how their solutions can be improved.
Algorithmic approaches to automate the evaluation of other graphical-based exercises already exist. For example, Aiouni et al. [12] proposed a graph-matching algorithm to evaluate flowchart exercises. And Alur et al. [14] created a tool for automatic grading of the construction of deterministic finite automatons (DFA). However, there is currently no approach that covers the comparison of fault tree-based exercises.
The rest of the thesis is structured as follows: Chapter 2 introduces the concept of fault trees in general, as well as the web application UTML used to create and export the fault

trees into a JSON format. Chapter 3 contains an overview of (commercial) fault tree analysis tools, an overview of quantitative analysis methods which have been considered as a foundation to compare two fault trees with each other, and provides information about automated grading in general. Currently, no tool or implemented algorithm allows for a fault tree comparison that suits our use case. Therefore, we will analyze how well a boolean equation comparison, an edit distance metric comparison and a pattern-matching comparison can be applied. Then, Chapter 4 describes in detail how those three chosen algorithms will compare two fault trees with each other, whereas Chapter 5 explains how they have been implemented. Chapter 6 introduces the test cases that will serve as the foundation of evaluating all three algorithms in Chapter 7. Our evaluation shows that creating a fair ranking of difference scores between two fault trees remains challenging. However, combining the feedback from the boolean equation comparsion and the edit distance metric yield promising results. By utilizing the boolean equation comparsion, we can identify most logical equivalent submissions correctly. Then, we can use the edit distance metric approach to provide detailed description how the submission needs to be corrected or can be improved. Therefore, we conclude that the final comparison strategy will be combining those two methods. Lastly, Chapter 8 concludes the major findings.

Chapter 2

Preliminaries

The following chapter serves as an introduction to fault tree analysis in general, including an overview of typically used events and gates and explains the difference between static and dynamic fault trees as well as the difference between quantitative and qualitative fault tree analysis in Section 2.1. The second part of the chapter introduces a web application called UTML, which can be used to draw fault trees and export their structure into a JSON format in Section 2.2.

2.1 Fault Trees

Fault tree analysis (FTA) is one of the most widely used techniques for system reliability and safety study, among others in the fields of nuclear power plants, aerospace and defence [41]. The technique was first developed in 1962 at Bell Telephone Laboratories with the aim to analyse the launch control system of the intercontinental Minuteman missle [40]. In general, FTA is an analytic technique where an undesired event like a system failure is defined, and then the system is analyzed in the context of its environment and operation to find all combinations of basic events that will lead to the occurrence of the predefined undesired event. These basic events represent the cause of the undesired event and can be

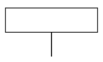
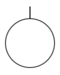
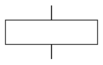
Name	Symbol	Description
Top-level Event		Represents the undesired event whose occurrence is supposed to be analysed. Usually a system failure.
Basic Event		Represents basic causes for the top level event, e.g. failure of single components or human error
Intermediate Event		Located between the basic events and the top level event. Intermediate events are caused by other events in the fault tree and, in turn, cause other events. They represent higher-level events that can be analyzed further.

Table 2.1: Overview of Fault Tree Events

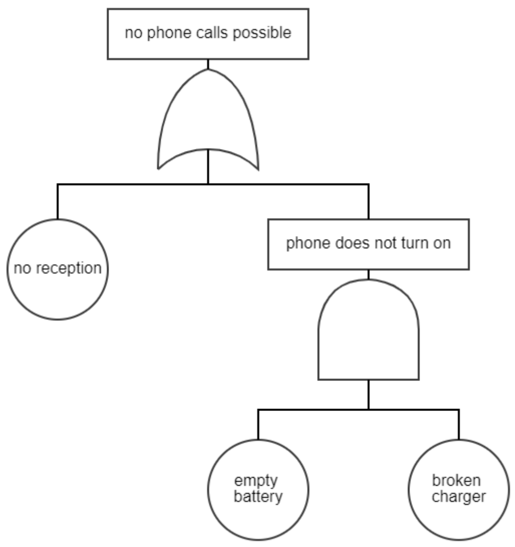


FIGURE 2.1: Fault tree example representing the undesired event of not being able to make a phone call

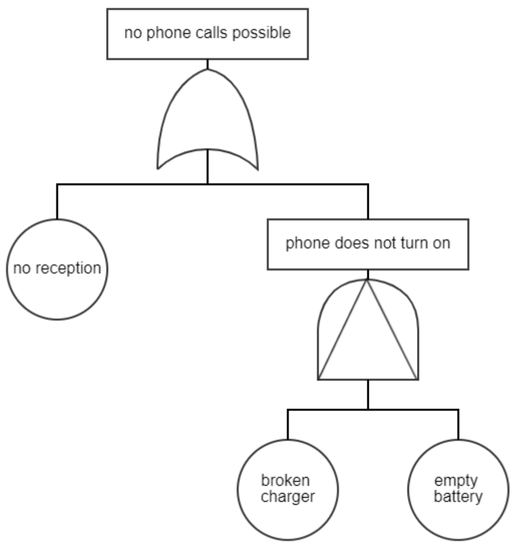


FIGURE 2.2: Dynamic Fault tree example representing the undesired event of not being able to make a phone call

associated among others with component hardware failures, human errors or environmental conditions. To represent this in the fault tree, three different types of events have been defined, which are listed and explained in Table 2.1: The top-level event, the basic event and the intermediate event.
A small example of a fault tree is given in Figure 2.1, modelling the undesired event "no phone calls possible", which is a top-level event. In Figure 2.1 we can also see an intermediate event "phone does not turn on" and three basic events "no reception", "empty battery" and "broken charger".
The second component of fault trees is logical gates. Logical gates represent the relationships between different events and conditions that can lead to a particular undesired or top-level event outcome. In static fault tree analysis, we generally differentiate between the gates AND, OR and VOTING. An illustration and description of these gates is provided in Table 2.2. This set of gates can be extended by a NOT gate and an XOR gate, which can also be seen in Table 2.2. However, fault trees using the NOT or XOR gate usually have disadvantages compared to fault trees consisting solely of the standard gates, for example, because they are generally more difficult to analyze [16].
For static fault trees (SFT), the system failure is insensitive to the order of occurrence of component fault events, since all of these gates are static coherent gates. The fault tree in Figure 2.1 contains two different gates. First, an AND gate combining the basic events "empty battery" and "broken charger", indicating that both events have to happen for the intermediate event "phone does not turn on" to occur. Second, there is an OR gate between the basic event "no reception" and the intermediate event, indicating that at least one of those events has to occur for the top-level event to happen.

2.1.1 Dynamic Fault Trees

Dynamic fault trees (DFT) extend the capabilities of static fault trees by introducing a temporal dimension, enabling a more comprehensive analysis of system failures over time. This means, that in contrast to static fault trees, the failure criteria of a system expressed

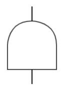
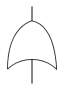
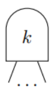
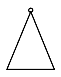
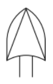
Name	Symbol	Description
AND Gate		Logic gate, whose output fault occurs if all of its input faults occur.
OR Gate		Logic gate, whose output fault occurs if at least one of the input faults occure.
VOTING Gate		Logic gate, whose output fault occurs if a specified minimum number k of input gates fail.
NOT Gate		Logic gate whose output fault occurs, if its input fault does not occur.
XOR Gate		Logic gate, whose output fault occurs, if exactly one of its input faults occurs.

Table 2.2: Overview of Static Fault Tree Gates

in a dynamic fault tree may depend on the combination of fault events and the sequence of occurrence of input events [26].
An illustration and description of the gates limited to dynamic fault trees are provided in Table 2.3, including the priority AND gate (PAND), the SPARE gate and the Functional dependency (FDEP).
Figure 2.2 contains a modified version of the "no phone calls possible" fault tree in the form of a dynamic fault tree. Instead of an AND gate as in Figure 2.1, the fault tree contains a priority AND gate. This indicates, that the event "broken charger" has to happen before "empty battery" for the intermediate event to occur. Logically, if the event "empty battery" happens first, the still functioning charger could be used to charge the battery again.

2.1.2 Fault Tree Analysis

Depending on the objectives of the analysis, fault tree analysis (FTA) can be qualitative or quantitative. Qualitative analysis is often used as an initial phase of the analysis process that focuses on understanding the structure of a fault tree and assessing the potential failure modes, without assigning specific quantitative probabilities to events.
The qualitative analysis could consist of studying minimal cut sets, studying minimal path

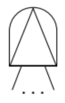
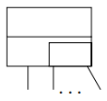
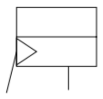
Name	Symbol	Description
Priority AND Gate (PAND)		AND Gate where the output fault occurs if the input faults occur in a specified order (from left to right).
SPARE Gate		Gate that indicates primary inputs (big/outer rectangle) and a pool of spare gates (small/inner rectangle). Output fault only occurs if both spare and primary input fail.
Functional Dependancy (FDEP)		Input trigger (input on the left side at the arrow) and dependant events (inputs at the bottom). Events fault occurs if a trigger has been raised.

Table 2.3: Overview of Dynamic Tree Events

sets or common cause failures for standard fault trees. For dynamic fault trees, one could apply methods such as minimal cut sequences, which are minimal sets of events that, if they occur together, will lead to the occurrence of the top event. [33]
On the other hand, quantitative analysis describes the part of the analysis process that assigns specific quantitative probabilities to events and conditions within a fault tree to assess the overall system reliability and risk [33]. For dynamic fault trees, such analysis methods could include the usage of Markov chains or dynamic Bayesian networks.

2.2 UTML

UTML [15] is a free web application designed by students of the University of Twente that allows one to visualise different types of diagrams, like UML diagrams and fault trees. The example fault tree in Figure 2.1 has been created using UTML. The frontend of UTML is written in TypeScript and is using the Angular framework, and the backend is designed using Spring Boot.
Next to visualizing diagrams, one of the features is to export the drawn diagram into a JSON file. A JSON export from Figure 2.1 can be seen in Listing 2.1. To highlight what is most important in this context, Listing 2.1 only contains the information related to the fault tree elements and their structure. Information UTML uses for their visation, e.g. the exact position and format of elements have been omitted. The JSON export contains two types of main categories: the "edges" and the "nodes". Each element in "edges" represents one connection between two fault tree elements. Each element in "nodes" represents either an event or a gate, which is specified in the property "type". E.g., the type "RectangleNode" represents a top-level or intermediate event and the "EllipseNode" represents a basic event.
As of now, UTML only supports the gate types AND and OR, but an extension for the missing static gates and the dynamic gates has been requested. To add the logic necessary

to use the dynamic gates, the general idea is to add new properties to the gate elements, that e.g. contain an ordering of all its child nodes which could be used to express their priority for a PAND gate. Until UTML gets extended, the generated JSON files will be extended manually to support the missing gates.

```json
{
  "edges": [
    {
      "startNodeId": 0,
      "endNodeId": 1
    },
    {
      "startNodeId": 3,
      "endNodeId": 4
    },
    {
      "startNodeId": 3,
      "endNodeId": 5
    },
    {
      "startNodeId": 1,
      "endNodeId": 2
    },
      "startNodeId": 1,
      "endNodeId": 6
    },
    {
      "startNodeId": 6,
      "endNodeId": 3
    }
  ],
  "nodes": [
    {
      "type": "RectangleNode",
      "text": "no phone calls possible",
    },
    {
      "type": "OR Gate",
      "text": "",
    },
    {
      "type": "EllipseNode",
      "text": "no reception",
    },
    {
      "type": "AND Gate",
      "text": "",
    },
    {
      "type": "EllipseNode",
      "text": "empty battery",
    },
    {
      "type": "EllipseNode",
      "text": "broken charger",
    },
    {
      "type": "RectangleNode",
      "text": "phone does not turn on",
    }
  ]
}
```

Listing 2.1: UTML JSON export representing the fault tree in Figure 2.1

Chapter 3

Related Work

So far, no direct approaches have been proposed to automate the evaluation of fault tree exercises. Therefore, we want to examine if the current landscape of (commercial) tools offers any features in regards to comparing two fault trees with each other in Section 3.1. Then, multiple qualitative analysis methods are presented and scrutinized for their applicability in comparing fault trees, both regarding static and dynamic fault trees in Section 3.2. Lastly, in Section 3.3 an exploration of automated evaluation standards is undertaken.

3.1 Existing Tools

Many software tools already exist that allow their users to perform fault tree analysis, both in the form of quantitative and qualitative analysis. Most of those tools are commercial and are not limited to analysing fault trees, but instead offer complex reliability, safety and availability problems and analysis of many different features.
In this section, we want to analyse what different software tools have to offer specifically to compare two or more fault trees with each other to analyze how suitable and effective these tools are in solving our specific use case. The tools to analyze have been chosen based on internet research and extended by an overview of existing tools provided by Ruijters and Stoelinga [33]. An overview of the tools that have been tested and if they support fault tree comparison can be seen in Tabel 3.1.
For those tools where Table 3.1 indicates that they do not support FT comparison, this explicitly means that they do not contain any features that find similarities and differences between two fault trees. However, though it is not suitable for our use case, it should be mentioned that most of them still support a multi-window view: This means that it would

(Commercial) tool	supports FT comparison
ALD Reliability and Safety Software [5]	No
Computer-Aided Fault Tree Analysis (CAFTA) [4]	Yes
Isograph Faulttree+ [7]	Yes
ITEM ToolKit [8]	No
Free Web Fault Tree Analysis (FTA) Software Tool [1]	No
ReliaSoft [9]	No

Table 3.1: Overview of analyzed commercial tools and whether or not they support a fault tree comparison feature

e.g. be possible to manually compare the minimal cut sets of two fault trees with each other.
The tools whose analysis yielded a positive result regarding the support of FT comparison, which are Isogepah Faulttree+ and CAFTA, will be described in more detail in the following subsections.

3.1.1 Isograph Faulttree+

Isograph FaultTree+ [7] is a software tool used for conducting fault tree analysis to assess system reliability, safety, and risk by modelling potential failure modes and their impacts. Isograph has a feature that allows its users to compare two projects, e.g. two fault trees, with each other. They call this feature "Differencing". To use this, it is possible to choose the projects one wants to compare, which can also be more than two, and specify which parts of the project should be assessed during this comparison.
For the use-case of comparing fault trees with each other, we started by comparing multiple fault trees containing small differences. Here, the most important observation is that the differencing feature only compares the overall structure of the fault trees with each other: Elements are identified based on an ID, which is assigned manually by the user, and then compared among the fault trees e.g. regarding their type or their outputs. Among others, this results in the following scenarios:

1. If two fault trees have the exact same structure, but one gate has a different ID than the other gate, it simply is detected that there are not identical gates in the fault trees, even if they are both AND gates.

2. If two fault trees have the same structure except for one gate type (Type AND in FT1 and type OR in FT2, but their ID is identical), then it is detected that those gates have different types.

3. However, if the fault trees are as described in 2 but have different IDs, then it only detects that there are not-identical gates, but does not detect that their difference is in regards to their type.

4. No feature to compare (minimal) cut sets has been included.

5. Equivalences have not been detected: For example, a fault tree with a top-level event, an AND gate and 4 basic events attached to the AND gate and a second fault tree with two AND gates which each have 2 basic events attached, have not been identified as equivalent.

Since equivalences cannot be detected, and matching between nodes is only possible using an identical ID, Isographes' "Differencing" feature would not be sufficient for our use case of evaluating student-based exercises, and therefore more complex comparison examples have not been created.

3.1.2 CAFTA

CAFTA (Computer Aided Fault Tree Analysis) [4] is another software application used for constructing, analyzing, and evaluating fault trees to assess system reliability and identify potential failure points. Related to fault trees, CAFTA offers a direct comparison of the gate structure of two fault trees and a direct comparison of two fault trees' cut sets.
For the cut set comparison, the tool calculates the minimal cut sets (MCS) of each fault

tree and highlights in each list, which cut set does not appear in the other list based on the name of each event. It can also identify whether there are only small differences in a cut set, like one element being named differently, or one cut set having an additional element in one fault tree. Next to the MCS, this comparison also contains a section for the basic events and their failure rate. Again, differences between those lists are highlighted.
The structural comparison of logical gates works as follows: The matching between gates happens based on the name, and the type of gate and the direct children are getting compared. Similar to the cut set comparison, the output is an individual list for each fault tree including the gate name, type and children, and highlighted are the parts that do not match the other fault tree. We can make the following observations:

1. If two fault trees have the exact same structure, but one gate has a different type than the other gate, the CAFTA's structural comparison detects that there are two different gates. However, the highlighting makes it clear that only the type differs.

2. Equivalences are not detected: For example, a fault tree with a top-level event, an AND gate and 4 basic events attached to the AND gate and a second fault tree with two AND gates which each have 2 basic events attached, have not been identified as equivalent.

Since equivalences cannot be detected, and matching between nodes relies on a common name, we conclude that CAFTA's comparison feature is also not a suitable tool for our use case. It provides a few more insights than Isograph from Section 3.1.1 in the form of a highlighting of the differences between the two files, but its comparison is still restricted to names and is solely based on the given structure and not the logic behind the structure.

3.2 Qualitative Analysis Methods

If a student hands in a solution that is identical to the given solution, it is easy to validate this and award the student all points without further explanation. The more complicated case is to find out how much a solution diverges from the given solution: Is the submission correct but not minimal? Is one gate mixed up in the solution? Are there more or less basic events in the submission? Is there one error that repeats itself throughout multiple subtrees? Those are only some scenarios that can happen when comparing one submission with the given solution. Therefore, this section is dedicated to researching existing qualitative analysis techniques in the field of FTA and evaluating how they could be applied to finding similarities and differences between two fault trees.
Table 3.2 contains an overview of which existing qualitative analysis techniques have been considered, and which have been explored further in this thesis: Boolean equation comparison, edit distance metrics, and a pattern-matching approach. These techniques were chosen because they represent three distinct approaches for comparing fault trees. Additionally, we anticipate they will offer valuable insights into the differences between the two compared fault trees, especially when the solution is incorrect. A more detailed discussion of this decision, including the potential benefits and drawbacks of each approach for our use case, is provided at the end of each method introduction.

3.2.1 Basic Structural Comparison

Fault trees can be analysed simply by identifying their structural elements, e.g. the number of events and gates and their underlying dependency, or the fault trees' depth and

Method	Further exploration?
Basic structural comparison	No
Cut set comparison	No
Boolean equations comparison	Yes
Modularisation	No
Graph Com Alg: Edit distance metrics	Yes
Graph Com Alg: Subgraph isomorphism	No
Graph Com Alg: Pattern Matching	Yes

Table 3.2: Overview of which analysis techniques will be explored further in this thesis.

complexity. Other aspects could be their common cause failure, which occurs when multiple events fail due to a single underlying cause, or their critical paths [33]. Among others, these structural comparisons are used to identify the most vulnerable part of the fault tree. We can use such structural elements to compare two fault trees with each other: We can identify common events that appear in both fault trees, which can provide insight into similarities in failure mechanisms. We can also identify events that only occur in one fault tree, which may represent a failure mode that has not been integrated into the other system. Next, we can analyse the type of logical gates that have been used, to determine if both fault trees use similar gate configurations or if there are variations in how events are combined to reach the top event. By analyzing the depth and complexity, we can evaluate whether one fault tree is more detailed or complex. Common cause failures can be compared to see if there are shared vulnerabilities between both systems and a critical path analysis can be used to check whether the systems have similar critical paths or if they differ significantly between the two systems. Then, it could be analysed if there are dependencies between events that are consistent or different between the systems. Finally, by analyzing gate combinations, we can e.g. evaluate if both fault trees use complex gate combinations, or if one fault tree uses simpler logic than the other.
Overall, a comparison of two fault trees based on structural elements can give a first indicator of the similarities and differences between the systems. However, it might provide a challenge to find a quantification for these basic structural comparisons: For example, how similar are two fault trees, just because they share a common cause of failure? Therefore, we decided not to include the basic structural comparison in our further exploration.

3.2.2 Cut Set Comparison

A cut set represents a set of basic events that cause the system modelled by the fault tree to fail. Minimal cut sets (MCS) are a subset of those, that have the smallest number of events which, if they all occur, cause the system to fail. [38]
In the qualitative analysis of standard fault trees, a cut set analysis is used to determine the most vulnerable elements of the fault tree, e.g. by identifying elements that appear more often than others in a minimal cut set or by identifying elements in cut sets with a small total number of elements. To calculate a fault tree's minimal cut set, different approaches have been proposed. Among others, it is possible to use boolean manipulation [38] or to derive the MCS by creating a Binary Decision Diagram for the fault tree [13].
For DFTs, the concept of cut sets is insufficient due to their time dependency but can be expanded to so-called cut sequences [27]. A cut sequence describes an order in which basic

events have to happen to result in a system failure.
In the use-case of comparing two fault trees with each other, it could be interesting to see which cut sets are identical, to identify failure pathways that are shared between both solutions. In addition, we could also determine which minimal cut set is unique to one system and whose failure pathway therefore contributes to failure in one system but not the other.
However, we decided not to further explore this method. While it might give a student some insights into which basic events need to be corrected, it might be difficult to gather more valuable feedback from this method. In addition, we might also run into issues that it is for example not guaranteed that two fault trees represent the same system structure just because their MCSs are similar.

3.2.3 Boolean Equations Comparison

Standard fault trees can be expressed in boolean equations, by describing the logical relationship between events based on the gates used in the fault tree. With the help of this boolean equation representation, it is possible to draw a conclusion about the fault trees' similarities and differences. To achieve this, the boolean equations of both fault trees are needed.
Boolean equations can be compared in different ways. For example, we could construct a truth table for both equations and analyze which and how many input combinations produce the same output for both truth tables and which do not.
Another approach is to make use of algebraic comparison. To make the comparison easier, we can normalize both equations, e.g. by using the Sum-of-Products form [18]. After the normalization, boolean algebraic operations can be used to find similarities and differences [36]: Logical equivalence can be used to determine if two fault trees are equivalent, by evaluating whether those equations are equal. However, this could still mean that one or both given fault trees are not minimal. Intersections can be used to find common elements that appear in both fault trees and union operations can be used to find the combined set of events that appear in either of the fault trees. Lastly, a complement can be used to identify events that are unique to one fault tree.
DFT cannot simply be represented in a standard boolean equation, since we still need to represent the dependent properties of events. Otherwise, an AND gate and a PAND gate would be seen as identical elements. However, we can create workarounds to fix that issue. For the algebraic approach for example, Ni et al. [28] propose an algebraic framework that represents each event, gate and logical union as variable arrays that allow modelling DFTs with OR, AND, PAND, FDEP and SPARE gates. Then, we could compare DFTs with the use of algebraic operations as we already established for static fault trees.
Boolean equation comparison is the first method that will be explored further in this thesis, specifically the truth table comparison approach. It allows for the simplification of complex logical expressions. By applying this method, we want to evaluate whether or not two fault trees are logically equivalent. The perks of this approach are, that equivalence can be detected, even if the fault trees do not have an identical structure. For example, if the fault tree is not minimal, but still a correct representation of the system description, the comparison should be able to return detected equivalence. Therefore, we want to include this method to see if we can extract all submissions that are logically equivalent to the solution. On the downside, this also indicates that we cannot use this approach to offer feedback in the form of improvements, but only for error handling.

3.2.4 Modularisation

Another method to compare the structure of two fault trees with each other is to find similarities in reduced parts of the fault trees via modularization. To apply this, we need to explore how we can find so-called modules in the fault tree and how we can use them to compare two fault trees.
A module describes a subtree of the fault tree whose basic events do not occur anywhere else in the fault tree [19]. Generally, modules are used to reduce the computational cost of basic operations on fault trees, like the computation of the minimal cut set. A linear-timed algorithm, to find modules has been proposed by Dutuit et al.[19], where they make use of depth-first left-most traversals of the fault tree.
Once the modules in both fault trees have been identified, we could e.g. compare how many of them overlap, and which ones are a subset of each other, or we could make use of previously discussed techniques.
In addition, modules can also be used for the analysis of DTFs [20]. When finding modules for DFTs, we can pay attention to differentiating between static modules and dynamic modules, indicating that static modules only contain gates allowed in SFTs. By doing so, we can apply comparison techniques for static fault trees to the static modules and only have to use the techniques developed for dynamic fault trees where it is necessary.
However, we decided not to further explore this idea, mainly because we anticipate that it will not be able to handle common mistakes that students might make very well. For example, if a student has all the right events, gates and dependencies, but has one extra connection between an already existing event and a gate, the module generation for both solutions will have huge differences. Also, by applying the approach to differentiate between static and dynamic models, we might run into issues that negatively influence the comparison. For example, if a student uses an AND gate instead of a PAND gate, but their solution is otherwise identical, creating modules that differentiate between static and dynamic will return vastly different results for those two fault trees, even though the only difference is one gate type. Finally, not all events may be part of one model, therefore we cannot guarantee that all elements are part of the comparison process by just using modularization.

3.2.5 Graph comparison algorithms

Dynamic and static Fault trees are direct acyclic graphs (DAG) [25] and therefore we can also make use of graph theory-based algorithms to compare the structural differences between fault trees. Overall, it can be noted that most graph comparison algorithms, like subgraph isomorphism and tree pattern matching, usually require a clear ordering of nodes. At first glance, this seems to be a very restrictive requirement for our use case, because we do not want to differentiate between different orders of subtrees in a fault tree. However, this restriction should be manageable by using preprocessing methods. This could be an algorithm that orders all nodes of the fault tree according to a given set of rules.
The following subsections will present three graph comparison techniques we can also use to analyze differences between two fault trees.

Edit distance metrics

One graph theory-based technique we can use to quantify the dissimilarity between fault trees is the edit distance metric [39]. The concept behind the edit distance metric is a number of operations required to transform one fault tree into another, like insertions, deletions and relabeling. Each of those operations gets a respective cost function. Then,

the edit distance problem is to create a so-called transformation script, which describes a sequence of operations that have to be executed to transform one tree into the other. Finally, the overall cost can be calculated based on the transformation script and the operations' cost function [17].
Various algorithms are already established for calculating edit distance. For instance, Zhang et al. [42] have presented an effective algorithm that determines the edit distance between trees. Such algorithms work as follows: Each fault tree will be converted into sequences of nodes and use these sequences to initialize an edit distance matrix. Then, the matrix will be iteratively filled based on the edit operations, to calculate the minimum edit distance between all pairs of substructures. The final value in the matrix finally represents the minimal edit distance value.
The edit distance metrics method is one of the algorithms we want to implement to evaluate its efficiency for our use case. Based on the cost function, we can automatically assess how similar the student's solution has been to the actual solution. In addition, we anticipate that the transformation script can serve as a guide for the student to understand how their solution needs to be modified. With the help of the edit distance metric, we hope to get detailed information about how the two fault trees differ, and which steps have to be performed to turn the student's submission into a replica of the correct solution. The downside to this approach is, that the difference is calculated on the exact structure of the fault trees, including the order of child nodes. Therefore, we do not expect this approach to be able to differentiate between errors and logically equivalent variants of the solution.

Subgraph isomorphisim

Two graphs are called isomorphic if there is a one-to-one correspondence between their nodes and edges while maintaining the connections between nodes [32]. This means, that the two graphs are considered identical in structure, even if the nodes are labelled differently.
Furthermore, subgraph isomorphism is a concept in graph theory that deals with determining whether one graph is isomorphic to a part in the larger graph [37]. In the area of fault trees, this would allow us to check if one fault tree is a subtree of the other. This could also be useful in the context of modularisation from Section 3.2.4, to see which modules of one fault tree also appear in the other.
However, just using subgraph isomorphism will not be sufficient enough for our use case, since it will only cover a very specific set of mistakes students can make, like forgetting to include a basic event. Since the calculation of isomorphism also relies solely on the structure of the nodes and edges, it will provide no useful information about the labelling of the nodes. For example, the usage of an AND gate instead of an OR gate will not be detected. Therefore it will not be included in the set of methods we want to explore further.

Pattern matching

Pattern matching describes the process of identifying the presence of a specific pattern or structure within a fault tree. In comparison to the modularization approach from Section 3.2.4, where predefined modules within the fault trees form the baseline for the comparison, pattern matching examines the entire fault tree structure. Multiple algorithms to perform pattern matching specialised for tree structures have been e.g. proposed by Hoffmann et al. [23].
For our use case, pattern matching could take on various forms. For example, we could

compute a similarity measure based on common substructures between two fault trees, or we could analyze which nodes from FT1 also appear in FT2.
Pattern matching is the third method we want to explore further. The previously chosen boolean equation comparison focuses on the logic behind the tree structure and the edit distance metric focuses on the structure of the fault tree as a whole unit. By incorporating a pattern-matching comparison that emphasizes the analysis of individual nodes within the tree, we introduce an additional perspective to the overall evaluation of fault trees.

3.3 Automated evaluation

This thesis aims to analyse how fault trees can be compared to each other with the overlaying goal of creating a tool which can automatically evaluate how accurate a student's fault tree solution is. The concept of automated assessment of exercises to reduce the workload of teachers is already widely established. Especially with exercise types that require textual answers, such techniques are widely in use [21].
Some advances have been made to automate the evaluation of graph-based or diagram-based exercise types. Usually, such tools work as follows: First, the teacher has to feed the evaluation tool with at least one correct solution. Then, the students can submit their solutions to the given exercise. The student's solution will be compared to the correct solution via a defined set of similarity criteria. Finally, the student will receive feedback on the correctness of their solution, which they can use for improvement.
An example of such a tool is described by Soler et al. [34], which has been developed for UML class diagrams or by Hoggarth et al. [24], which has been extended to be used for several domains, like Entity-Relationship Diagrams or Sequence Diagrams. The second example tool operates as follows: Students have a graphical interface, where they can use predefined elements to draw their solution. After a student submits a solution and it has been compared to the previously defined correct solution, the student gets visual feedback on which components are correct and which are incorrect. For those incorrect components, the student can also access additional textual feedback that reasons why it is not marked as correct. It has been observed, that next to reducing a teacher's workload, the repetitive exercises of such a tool can be useful in strengthening a student's knowledge, and the provided feedback supports the student to understand the subject material even better [24].
Overall, the critical part of these tools for automated evaluation is how the similarity criteria are defined and how feedback will be generated. To define the similarity criteria, we already established that we will implement several methods that have been proposed in Section 3.2.
Once we analyse which comparison method is the most promising, we want to extend its implementation to not just provide a similarity score, but also textual feedback the student can learn from. How exactly we can generate textual feedback heavily depends on the used comparison methodology. To do this, several principles to formulate the output could be taken into consideration:

- Consider two types of feedback: The first type specifies errors in the student's solution, whereas the second type lists additional differences between the student's solution and the professor's solution, even though they do not lead to an error, to show how the student could improve their fault tree. [24]

- Classify different types of errors that might appear and assign a message to each error type [35], e.g. the number of basic events is incorrect, less basic events are required, incorrect gate type.

- Consider the seven principles of formative feedback practises from Macfarlane-Dick and Nicol [29]: These principles include e.g. that feedback should encourage positive motivational beliefs or that effective feedback also provides information to teachers that can be used to set the focus of the next lesson.
- Consider a general priority ranking of the feedback: If a solution contains several errors, highlight e.g. the two most critical of them, so that the student knows which shortcomings to focus on first [22]

Chapter 4

Algorithm Description

The following chapter provides a detailed description of how the chosen methods to compare two fault trees work in theory. To illustrate this, we explain the methods individually and demonstrate them using one static and one dynamic example.
Figure 4.1 shows a static fault tree of a stranded road trip. For this road trip to fail, the phone has to fail (so that help cannot be called) and the bike itself has to fail. For the bike to fail, either the engine has to fail or two out of three tires, which includes two regular tires and one spare tire. This correct solution will be compared to a student's submission, which can be seen in Figure 4.2. This submission is nearly identical to the correct solution, except for one gate, which is highlighted in red in the Figure, that has been switched from an OR gate to an AND gate.
Figure 4.3 shows a dynamic fault tree version of the stranded road trip including all three dynamic fault tree gates. The first difference is an additional PAND gate to take into account that the phone could be charged as long as the charger is functional. The second difference is that the VOTING gate in the static version has been replaced by a SPARE gate, indicating that only the two regular tires are in use. Finally, this variant uses a

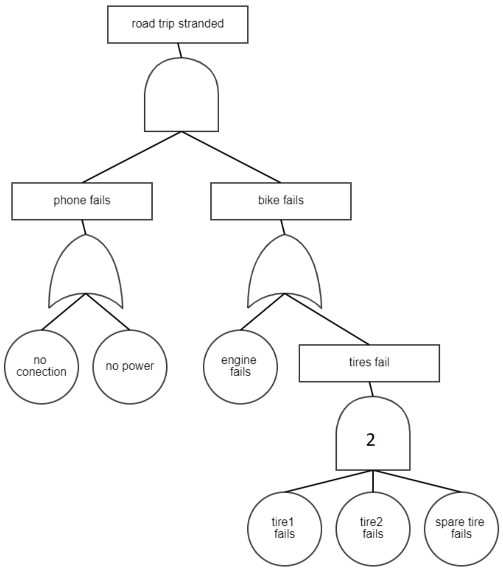


FIGURE 4.1: Static fault tree example: Correct solution of a stranded road trip

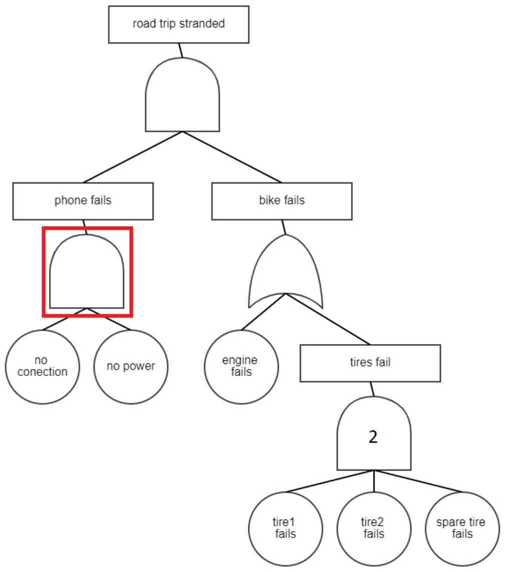


FIGURE 4.2: Static fault tree example: Incorrect solution of a stranded road trip

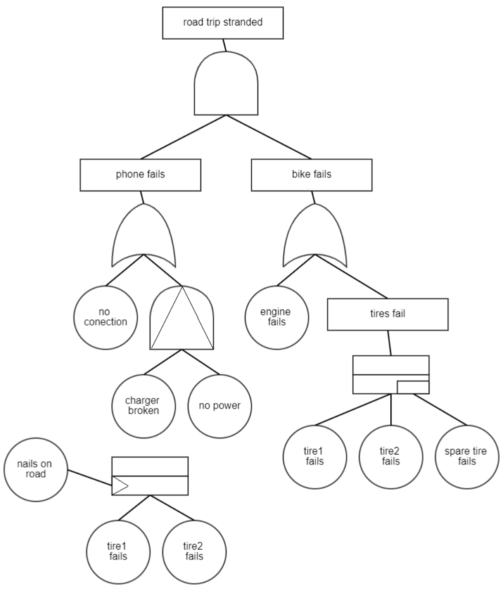


FIGURE 4.3: Dynamic fault tree example: Correct solution of a stranded road trip

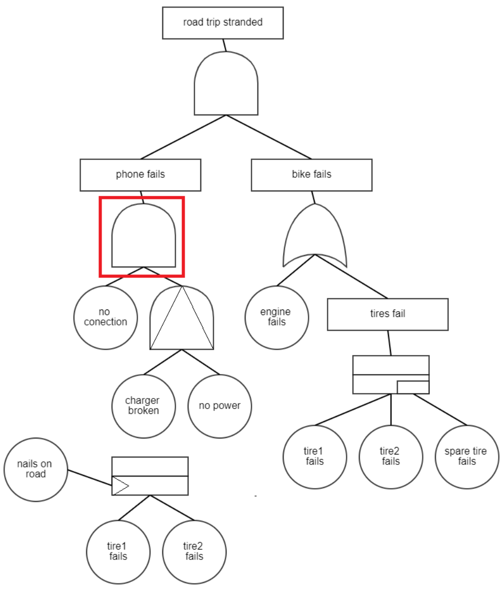


FIGURE 4.4: Dynamic fault tree example: Incorrect solution of a stranded road trip

functional dependency to showcase that both tires will get destroyed if there are nails on the road. We will compare this dynamic solution to a student's attempt shown in Figure 4.4. Similar to the static example, the difference between the two fault trees is a gate switch from an OR gate to an AND gate, which is highlighted in red in the student's submission.

4.1 Boolean Equation Algorithm

For the boolean equation approach, we perform a comparison of the truth tables that can be generated based on the boolean equation of a fault tree. To achieve this, the following steps have to be executed:

1. Preprocess fault trees.
2. Generate a boolean equation for both fault trees.
3. Generate a truth table for both boolean equations.
4. Compare how many variable allocations of the truth table yield the same output.
5. Postprocess fault trees.

Steps 2 to 4 are identical between static and dynamic fault trees. Step 1, the preprocessing of fault trees differs between static and dynamic fault trees, with the preprocessing of dynamic fault trees being an extension of the static preprocess. Step 5 is only required for dynamic fault trees. The details of the pre- and postprocessing will be explained in the following subsections.

4.1.1 Static Fault Tree Specifics

Generally, static fault trees can be represented as a boolean equation: OR, AND, NOT and XOR gates all have their respective boolean operator. The only static gate that cannot be expressed directly is the VOTING gate. However, every VOTING gate can also be expressed as a combination of AND and OR gates. Therefore, this transformation can be applied as a pre-processing step, so that no more voting gates are used in the fault tree before it will be transformed into a boolean equation. Additionally, all intermediate events will be removed from the fault tree, since they cannot be part of the boolean equation.

Example

Figure 4.1 shows us an example of a fault tree that illustrates which events need to occur for a road trip to fail. Figure 4.2 represents a student's attempt to recreate this fault tree based on a textual description of the context.
To compare these two solutions, we will follow the steps presented in Section 4.1.
Step 1. First, preprocessing needs to be executed to ensure that all gates used in the fault tree can be used in a boolean equation, which concerns all VOTING gates and the removal of intermediate events. The result of the preprocessing step of the teacher's solution can be seen in Figure 4.5: The voting gate has been split into three separate AND Gates, each containing two of the three tires. If one of these AND Gates fails, the intermediate event "tires fail" will occur. The same preprocessing has also been executed on the student's solution, whose results are shown in Figure 4.6.
Step 2. Once all voting gates have been transformed and intermediate events removed, we can move on to step two and generate boolean equations for both fault trees. For the teachers solution in Figure 4.5, this results in the following boolean equation:

$$\begin{aligned}\text{Roadtrip fails\_teacher} = {} & (\text{no\_connection} \lor \text{no\_battery}) \land \\ & (\text{engine\_fails} \lor (\text{tire1} \land \text{tire2}) \lor (\text{tire1} \land \text{spare\_tire}) \lor (\text{tire2} \land \text{spare\_tire})))\end{aligned}$$

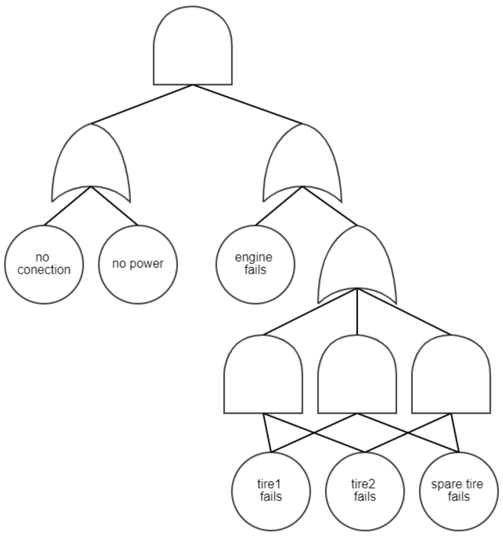


FIGURE 4.5: static fault tree example: Correct solution after the preprocess

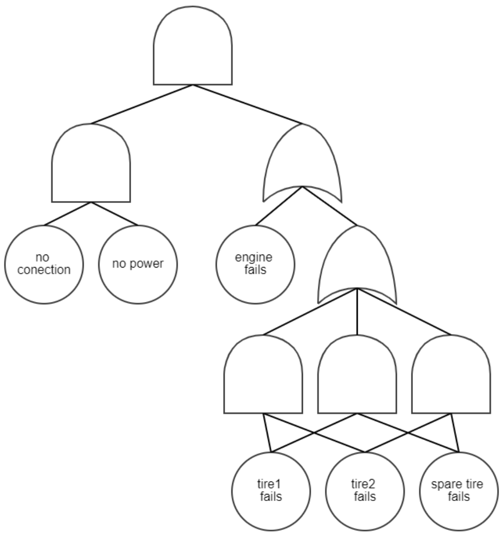


FIGURE 4.6: static fault tree example: Incorrect solution after the preprocess

Similar, the students solution results in the following boolean equation, with the difference that there is an AND relation, which is marked in red, between the events no_connection and no_battery, instead of an OR relation:

$$\begin{aligned}\text{Roadtrip fails\_student} = &(\text{no\_connection} \wedge \text{no\_battery}) \wedge \\ &(\text{engine\_fails} \vee (\text{tire1} \wedge \text{tire2}) \vee (\text{tire1} \wedge \text{spare\_tire}) \vee (\text{tire2} \wedge \text{spare\_tire})))\end{aligned}$$

Step 3. Then, truth tables for both boolean equations can be generated. An excerpt of this truth table showing the first 15 variable allocations and their output can be viewed in Listing 4.1.
Step 4. Based on these truth tables, we can generate a similarity score by counting how many variable allocations result in the same outcome. Here, 6 different basic events yield 64 variable allocations. Out of these 64 allocations, 23 allocations yield different results for road trip teachers and road trip students. In Listing 4.1, these occurrences are highlighted in orange. Overall, this results in a difference score of $\frac{23}{64} = 35.938\%$.
Step 5. Since there are no post-processing steps for static fault trees, this concludes the boolean equation comparison example.

4.1.2 Dynamic Fault Trees Specifics

Due to the types of gates a dynamic fault tree can contain, it is not necessarily possible to make a direct translation between a dynamic fault tree and a boolean equation. In addition to static gates, dynamic fault trees can contain FDEPs, SPARE and PAND gates. FDEPs can be handled similarly to VOTING gates, where we translate them into OR gates in respective places. PAND and SPARE gates on the other hand do not have an equivalent representation we can use. Instead, we have used a different workaround: First, PAND gates will be replaced by AND gates and SPARE gates will be replaced by VOTING gates. However, we will tag them to indicate that they used to be a PAND or a VOTING gate. Then, we will execute steps 1 to 4 mentioned above. Finally, we add a comparison in the post-process to check if gates tagged as previous PAND or previous SPARE gates in the student's solution are also tagged as such in the teacher's solution. To achieve

	engine fails	no battery	no connection	spare tire	tire1	tire2	**Road trip teacher**	**Road trip student**
1	F	F	F	F	F	F	**F**	**F**
2	F	F	F	F	F	T	**F**	**F**
3	F	F	F	F	T	F	**F**	**F**
4	F	F	F	F	T	T	**F**	**F**
5	F	F	F	T	F	F	**F**	**F**
6	F	F	F	T	F	T	**F**	**F**
7	F	F	F	T	T	F	**F**	**F**
8	F	F	F	T	T	T	**F**	**F**
9	F	F	T	F	F	F	**F**	**F**
10	F	F	T	F	F	T	**F**	**F**
11	F	F	T	F	T	F	**F**	**F**
12	F	F	T	F	T	T	**T**	**F**
13	F	F	T	T	F	F	**F**	**F**
14	F	F	T	T	F	T	**T**	**F**
15	F	F	T	T	T	F	**T**	**F**

Table 4.1: Excerpt of the teacher's and student's truth table for the static fault tree solution of the stranded road trip

this, a two-step comparison will be executed. In the first step, we will only check if the number of occurrences of PANDs and SPAREs is identical between the two solutions. In a second step, we compare if the tagged nodes themself are identical, by comparing all of its children. If the second step yields positive results, we can be very certain that PANDs and SPAREs have been used as intended. If the second step yields negative results, we can at least conclude that the usage of the correct gate has been attempted and that a difference between the two fault trees is related to the usage of the SPARE or the PAND gates.

Example

Figure 4.3 shows us an example of a fault tree that illustrates a failed road trip. Figure 4.4 represents a student's attempt to recreate this fault tree based on a textual description of the context.

Step 1. First, the fault tree gates PAND, SPARE and FDEP will be preprocessed as described above. The results of the transformation of these three gates can be seen in Figure 4.7 for the teacher's solution and in Figure 4.8 for the student's attempt. For both fault trees, the executed steps are identical: First, the PAND gate has been switched out with an AND gate and tagged as a previous PAND gate, which is highlighted in orange. Then, the SPARE Gate was replaced by a VOTING gate with a 2 out of 3 capacity and tagged as a previous SPARE gate, highlighted in blue. Lastly, each triggered event in the FDEP has been extended with an OR gate and the triggering event, highlighted in green. After these dynamic gates have been transformed, we need to perform the preprocessing required for static fault trees as well, which will transform the newly created VOTING gate and remove the intermediate events.

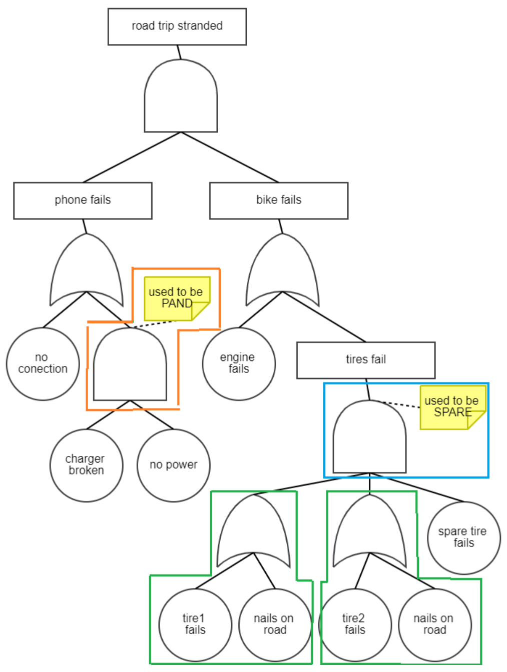


FIGURE 4.7: dynamic fault tree example: Correct solution after preprocessing dynamic gates

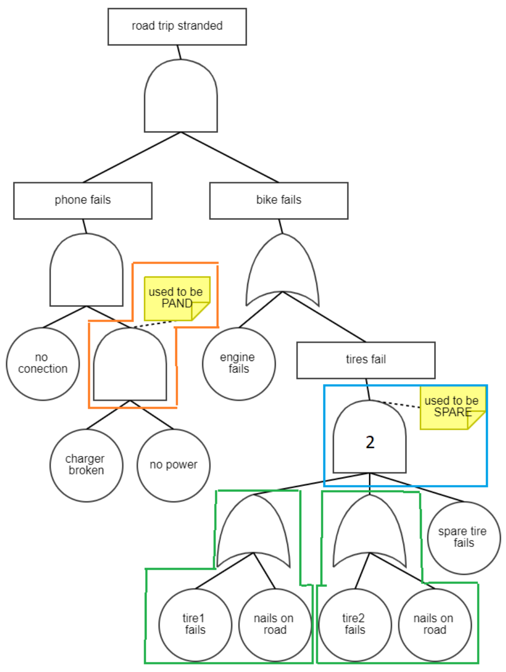


FIGURE 4.8: dynamic fault tree example: Incorrect solution after preprocessing dynamic gates

Step 2. Next, we have to generate the boolean equations for both fault trees. For the teacher's solution, the boolean equation is

$$\begin{aligned}&\text{Roadtrip fails\_teacher} = (\text{no\_connection} \lor (\text{broken\_charger} \land \text{no\_battery}) \land \\ &(\text{engine\_fails} \lor ((\text{nails\_Road} \lor \text{tire1}) \land (\text{nails\_Road} \lor \text{tire2})) \lor \\ &\qquad ((\text{nails\_Road} \lor \text{tire1}) \land \text{spare\_tire}) \lor ((\text{nails\_Road} \lor \text{tire2}) \land \text{spare\_tire})))\end{aligned}$$

For the student's attempt, the boolean equation looks as follows:

$$\begin{aligned}&\text{Roadtrip fails\_student} = (\text{no\_connection} \land (\text{broken\_charger} \land \text{no\_battery}) \land \\ &(\text{engine\_fails} \lor ((\text{nails\_Road} \lor \text{tire1}) \land (\text{nails\_Road} \lor \text{tire2})) \lor \\ &\qquad ((\text{nails\_Road} \lor \text{tire1}) \land \text{spare\_tire}) \lor ((\text{nails\_Road} \lor \text{tire2}) \land \text{spare\_tire})))\end{aligned}$$

Similar to the static example, the difference between these equations is the equation operator marked in red, which has been switched from an OR to an AND.
Step 3. Step 4. Next, a truth table will be generated for both equations to compare the outcome for each variable allocation. In this example, there are 8 basic events, therefore there are 256 different variable allocations we have to consider. Based on the truth table comparison 128 variable allocations yield different results for the teacher's and the student's solution. Therefore we have a difference score of $\frac{128}{256} = 50.393\%$.
Step 5. Lastly, the post-processing has to be executed. Here, we check whether all gates that have a PAND tag and all gates that have a SPARE tag of the teacher's solution also exist in the student's solution. Based on the teacher's solution in Figure 4.7, there is one PAND tag and one SPARE tag, which also occur in the student's solution in Figure 4.8. Therefore we can assume that the handling of these gates has also been correct in the submitted version of the student.

4.1.3 Feedback

For the boolean equation comparison, we can generate different kinds of feedback. First of all, we can check whether the two fault trees are logically equivalent or not: If all variable allocations of the truth table yield the same boolean value for both fault trees, and the postprocess of SPARE and PAND gates returns positive results, the fault trees are logical equivalents. If the postprocess detects a difference but overall the variable allocations still yield the same results, we can assume that the solutions are generally very similar except for one or more switched gates from AND to PAND or VOTING to SPARE.
If the truth table comparison returns differences between the variable allocations, the fault trees are not logically equivalent. In addition, a difference score of the number of different variable allocations divided by the number of possible variable allocations will be returned. Lastly, we want to see if we can gather information about each basic event, by counting how often a gate variable is allocated with FALSE in all the variable allocations that yield different results. While testing the BEC, we observed that gate variables responsible for differences in the fault tree often occur more frequently or infrequently in their allocations compared to other gates. To validate this observation and assess its potential for providing useful feedback, we assume that gates with a balanced division between TRUE and FALSE allocations are less likely to be part of the reason the two fault trees are different, whereas gates with many TRUE or many FALSE allocations might be the reason for those differences.

4.2 Edit Distance Metrics

To calculate the edit distance between two fault trees, one of the two fault trees will be transformed into the other fault tree, with the help of INSERT, UPDATE and DELETE operations. For example, to transform the static student's solution for the failed road trip shown in Figure 4.2 into the teacher's solution from Figure 4.1, only the student's AND gate at the intermediate event "phone fails" needs to be transformed into an OR gate using one UPDATE operation. To retrieve a final score for such a transformation, each of the operations INSERT, UPDATE and DELETE gets a cost function assigned. For our example, we give each operation a score of 1. Therefore, the complete transformation results in an edit distance score of 1.
However, while updating this single gate in the example might be the obvious answer, it is not the only answer. Instead, the transformation script, which is a listing of all operations that have to be executed for one fault tree to be translated into the other fault tree, can also look as follows: First, from the intermediate event, "phone fails", DELETE the basic event "no_connection", DELETE the basic event "no_battery" and DELETE the AND gate. Then, INSERT an OR gate, and INSERT as children to this OR Gate the basic events "no_connection" and "no_battery". This transformation script results in an edit distance of 6.
Therefore, the main purpose of the edit distance metrics algorithm is to find the transformation script with the minimal edit distance out of all possible transformations. Multiple approaches have been proposed to achieve this, and we describe the APTED approach in more detail in Subsection 4.2.2.
The edit distance metrics algorithms have the limitation that the nodes have to be in order. This means, that if one node has the children "no_connection" and "no_battery", and another node has the children "no_battery" and no_connection", it will not be viewed as equal. Therefore both fault trees need to go through a preprocessing step where all child nodes are ordered in alphabetical order.

4.2.1 Dynamic Fault Tree Extension

To enable the edit distance approach to support dynamic fault trees, additional information must be provided for the comparison. For the SPARE gate and the FDEP, we have to know which elements of the gate's children are spare nodes for the SPARE gate, and which children are triggers for the FDEP. To do this, all spares of a SPARE gate will get the prefix "SPARE_" and all triggers of an FDEP gate will get the prefix "TRIGGER_". For example, this means that the calculated edit distance metric will contain an UPDATE operation if one node is incorrectly marked as a spare because the node text has to be updated from its original text "Basicevent1" to "SPARE_Basicevent1".
To ensure that the logic behind PAND gates will be handled correctly, we prohibit the alphabetical reordering of PAND's children nodes in the preprocessing step of static fault trees.
Otherwise, the edit distance metric for dynamic fault trees will follow the same execution steps as the edit distance metric for static fault trees.

4.2.2 APTED

The APTED [31] [30] (All Path Tree Edit Distance) algorithm is a modern and efficient method for calculating the edit distance between two tree structures. APTED is designed to be more efficient than other tree edit distance algorithms, such as the one proposed by

Zhang and Shasha [42]. While Zhang and Shasha's algorithm has a cubic time complexity in the worst case, APTED includes several optimizations that result in a significantly improved average-case performance. This is achieved for example by breaking down the trees into smaller subtrees and processing them in an order which reduces redundant calculations.
APTED starts by linearizing the trees through a post-order traversal, which assigns unique IDs to each node based on its position in the traversal order. It then uses a dynamic programming table to keep track of the minimum edit costs for transforming subtrees. The algorithm recursively computes the costs for each node, considering the defined cost functions and the structure of the trees.
Therefore, the implementation of APTED involves several important steps, including preprocessing the trees to compute post-order IDs, defining the cost functions, and using the dynamic programming table to calculate the tree edit distance. To simplify the application for developers, libraries such as "apted" in Python provide a ready-to-use implementation of the APTED algorithm.

4.2.3 Feedback

We can retrieve two different kinds of feedback from the edit distance metric comparison. First of all, we can gather information from the edit distance score. If the score is zero, the two fault trees are identical. If it is greater than zero, then they are not identical. In this case, more information can be gathered from the second kind of feedback, which is the transformation script. This transformation script contains information about which nodes have to be inserted, updated or deleted so that one fault tree will be identical to the other.

4.3 Pattern Matching

To apply pattern matching as a comparison technique between two fault trees, a greedy algorithm has been developed. Greedy algorithms are algorithms that do not necessarily find the optimal solution for a problem, but which need significantly less time and memory to execute.
In our specific case, the developed approach is to iterate through all gate nodes of the student's solution and calculate a difference score to all unmatched nodes in the teachers's solution. Then, the nodes with the smallest difference score will be matched to the students's node. In the end, all minimal difference scores can be added together to get an overall difference score between the fault trees.
How to calculate the difference score between two nodes is interchangeable, though the elements we want to consider are the node type, the node text, the number of children and the level of the nodes in the fault tree.

Example

In this example, the difference score will be calculated using the following equation:

$$\begin{aligned}\text{DiffScore\_Static} = {} & AreEqual(S.Name, T.Name) + AreEqual(S.Type, T.Type) + \\ & |S.Level - T.Level| + MatchingChildren(S.Children, T.Children)\end{aligned}$$

The function AreEqual(x,y) returns 0 if both parameters are equal, and 1 otherwise. It is used to check if e.g. the minimum for the VOTING gate is equal, and if the type of the nodes is identical. The function MatchingChildren(x, y) verifies how many child nodes are

teacher gates	names score	types score	level score	child score	**total**
AND - top level	0	0	0	0	**0**
OR - phone fails	0	1	1	2	**4**
OR - bike fauls	0	1	1	2	**4**
VOTING 2	1	1	2	3	**7**

Table 4.2: Comparison of the student's top-level event with all unmatched events of the teacher's solution

teacher gates	names score	types score	level score	child score	**total**
AND - top level	-	-	-	-	-
OR - phone fails	0	1	0	0	**1**
OR - bike fauls	0	1	0	2	**3**
VOTING 2	1	1	1	3	**6**

Table 4.3: Comparison of the student's second level AND gate with all unmatched events of the teacher's solution

student gates	matched teacher gates	difference score
AND - top level	AND - top level	0
AND - phone fails	OR - phone fails	1
OR - bike fauls	OR - bike fauls	0
VOTING 2	VOTING 2	0
	Total	**1**

Table 4.4: Final matching between student's and teacher's solution with a total difference score of 1.

identical in both child lists. For each child node of x missing in y and for each child node of y missing in x, the score will be increased by 1.
To compare the failed road trip fault tree modelled by the student shown in Figure 4.2 to the correct solution of the teacher in Figure 4.1, the pattern matching algorithm will execute as follows:
The first node to match is the top-level gate AND from the students's fault tree. For this current node AND we calculate the difference score of all gates in the teachers's solution, which can be seen in Table 4.2. According to the equation to calculate the difference score for static fault trees, the Table shows a listing of each component of the equation for each gate in the teacher's solution. As a result, we can match the top-level AND gate of the teacher's solution, which is highlighted in green in the table, with the current node from the student's solution. In total, they have a difference score of 0.
Next, we repeat this step for the next node in the student's solution, which is the AND gate following the intermediate event "phone fails". Table 4.3 shows the difference scores for each teacher's gate in comparison to this current node. The gates that are highlighted in orange have already been matched to another node from the student's solution and are therefore not considered for all following comparisons. As it can be observed in the Table, the match with the smallest difference score of 1 is the teacher's OR gate that follows the teacher's intermediate event "phone fails" and will therefore be matched to the current node.
This process will be repeated until all student's nodes have been matched. A final matching

for this example can be seen in Table 4.4 with a total difference score of 1. The first row of the table is the matching from Table 4.2, and the second row is the matching from Table 4.3. For all other gates and events in the student's solution matches with a difference score of 0 could be found. Therefore, it can be concluded that the only difference between the two fault trees is the node type of the gate after the intermediate event "phone fails".

4.3.1 Dynamic Fault Tree Extension

To allow this method to support dynamic fault trees, the difference score function can be extended. Here, we can add another check to evaluate the additional specifications for the dynamic gates.

$$\text{DiffScore\_Dynamic} = \text{DiffScore\_Static} + \text{DynamicSpecifications(S.Infos, T.Infos)}$$

For SPARE gates, the method DynamicSpecification(x, y) evaluates, if the child nodes that function as spare nodes are identical or how many nodes differentiate them. Similarly, for FDEPs, this will be evaluated for all nodes functioning as triggers. Lastly, for PAND gates it will be evaluated how many child nodes are out of order.
Otherwise, pattern matching for dynamic fault trees will follow the same execution steps as pattern matching for static fault trees.

4.3.2 Feedback

In the pattern-matching comparison, the feedback includes a difference score and the mapping of each TreeNode to its counterpart in the other fault tree. If the difference score is greater than 0, we can identify which TreeNode matches contribute to the increase, highlighting the specific parts in the student's submission that might require modification.

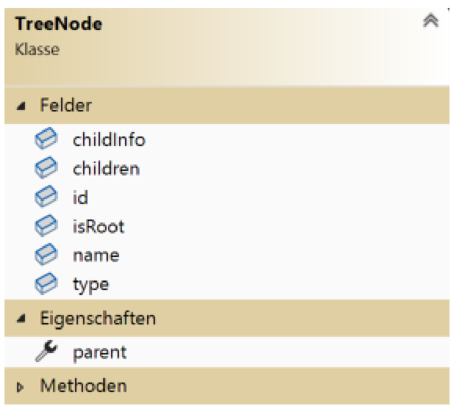
Chapter 5

Implementation

The algorithms described in this thesis have been implemented as a console application using C# and the .NET 8.0 SDK.
The following chapter includes a description of the data structure to represent the fault tree in the C# application in Section 5.1as well as an explanation of how the three different algorithms have been implemented: First, Section 5.2 introduces the interface used to implement the algorithms. Then, Section 5.3 contains the description of the boolean equation comparison, Section 5.4 of the edit distance metrics comparison and Section 5.5 of the pattern matching comparison.

5.1 Tree representation

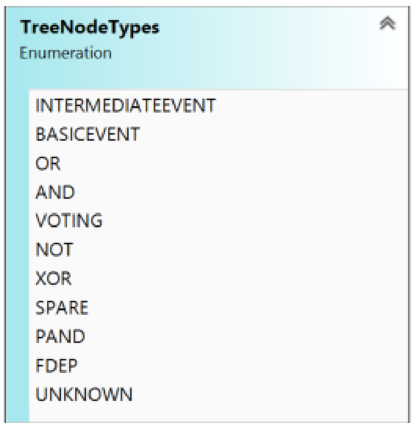
A new data structure called *TreeNode* has been implemented to represent the fault trees. The class TreeNode with its attributes can be viewed in Figure 5.1. Each TreeNode gets a unique *ID* in the form of a GUID. The attribute *type* stores which kind of fault tree element the node represents. This attribute is of the datatype *TreeNodeTypes*, which is an enumeration shown in Figure 5.2, containing all possible gate types in static and dynamic fault trees, the event types basic event and intermediate event and an Unknown type. The attribute *name* represents the text of a fault tree element, and will e.g. store the name of an event or the number of a VOTING gate.
To model the structure of a tree, each TreeNode has an attribute *children*, which is a list



TreeNode
Klasse

- Felder
  - childInfo
  - children
  - id
  - isRoot
  - name
  - type
- Eigenschaften
  - parent
- Methoden

FIGURE 5.1: Class TreeNode with its attributes



TreeNodeTypes
Enumeration

INTERMEDIATEEVENT
BASICEVENT
OR
AND
VOTING
NOT
XOR
SPARE
PAND
FDEP
UNKNOWN

FIGURE 5.2: Enumeration TreeNodeType

of other TreeNodes. In addition, to allow for an easier iteration through the tree structure, a TreeNode also contains a reference to its *parent* TreeNode. Therefore, each TreeNode element can only have one parent.
ChildInfo is a list of GUIDs which is used to store additional information for the dynamic gates. In the case of a PAND gate, it contains all IDs of its child nodes in their correct ordering. For SPARE gates, it includes the IDs of all child nodes that function as the spare nodes, and for FDEPs those that function as the trigger nodes. For all other node types, this list will remain empty. Lastly, the attribute *isRoot* will indicate if this TreeNode element is the root of the modelled fault tree.

5.1.1 UTML To TreeNode Transformation

Several steps need to be executed to transform the JSON exported from UTML to the TreeNode representation. First, we need to read the JSON data using the C# JSON serializer. This allows us to create a temporary representation in a data structure that directly captures all the important information from the JSON file. Similar to the JSON representation, this intermediate structure contains two separate lists: one list that includes all the individual nodes, and one that represents the edges between nodes.
Before we create the TreeNode representation, we want to ensure that the fault tree has no occurrences of one node having two parents, because the implementation of the TreeNode only allows for one parent node. Therefore, we have to create duplicates of each node that has more than one parent. To do so, we iterate over the list of edges. Each edge contains information about the index of its start node and its end node. For each edge, we check if the index of the end node occurs more than once. For each occurrence, we append a duplicate of this end node to the node list and update the end node reference of the matching node to point to this new node instead. Then, we also have to ensure that all child nodes of this duplicated node remain children of both node occurrences. To achieve this, we duplicate all elements in the edge list, where the start node references the node we duplicated, and assign the start node to the newly created node. By duplicating the edges, it is possible that we created more nodes with two parents. Therefore, we repeat this process until we iterate the edge list without two occurrences of the same end node references.
Once we have this intermediate representation, we can use it to generate our TreeNode representation. First, we iterate over the list of nodes and create a new TreeNode for each of these list items. During this iteration, the new TreeNode receives information about its attribute name and type and gets assigned a newly generated ID. These new TreeNodes are stored in a new list so that their indexing is identical to the original nodes list.
Then, we iterate over the original list of edges. Each edge contains information about the index of its start node and its end node. Based on these indexes, we can find the corresponding nodes in the TreeNode list and add the end TreeNode as a child of the start TreeNode. Simultaneously, the start TreeNode will be added as the parent of the end TreeNode.

Property ChildInfo

So far, the hierarchical structure and the information about the text and the type are represented in the TreeNode structure. To support the usage of dynamic gates as well, we need to translate the information related to the attribute childInfo. Since we need to ensure that all TreeNodes have been created and have been getting their unique ID, the childInfo mapping has to happen in a second iteration of the original node list.

In the JSON representation, this extra information about the children is given by a list of indexes, which refers to the index of the node in the original extended nodes list.

Functional Dependancies

Generally, the parent-children hierarchy will be identical between the tree representation in the JSON format and the TreeNode representation. However, it is typical for functional dependencies to be displayed independently from the rest of the fault tree, which can be observed in Figure 4.3. However, in the TreeNode representation, only one root node is allowed. Otherwise, it cannot be ensured that all TreeNodes are processed when iterating through the fault tree. Therefore we require a workaround to integrate the FDEPs into the tree structure.
To do this, we iterate through the list of nodes after all TreeNodes have been initialized the first time. If we come across a TreeNode of the type FDEP that does not have another NodeTree set as its parent node, we will add it as a child node to the actual root of the fault tree. This way, we can guarantee that whenever we iterate through the complete TreeNode representation of the fault tree, these FDEPs will not be skipped.

5.1.2 Precheck tree

To ensure that the JSON files used to generate the fault trees contain a description of a valid fault tree, a correctness validation will be executed once the NodeTree representation has been created. The correctness validation checks that the following rules apply to the generated fault tree.

- Check that basic events do not have children.
- Check that all nodes referenced in the ChildInfo list are also children of that node.
- For PAND gates, check that the number of ChildInfo elements is equal to the number of children.
- Check that each gate of the type AND, PAND, OR, XOR, SPARE, VOTING and FEDP has at least two children nodes.
- Check that each gate of the type NOT has exactly one child.
- Check, that there is only one node that qualifies as the root node. If this is an intermediate event, ensure that it only has one child node.
- For FDEPs, check that all triggered nodes are basic events.

While this list is not a complete list of all rules a fault tree might have to be coherent with, it covers many scenarios that might otherwise lead to errors during the execution of the different comparison algorithms.

5.2 IAlgorithm Interface

To implement the different algorithms, we implemented an interface called IAlgorihtm. This interface contains three methods:

- *TreeNode preprocessing(TreeNode root):* Used to preprocess the fault tree, including all necessary pre-process steps for static fault trees, before the specific comparison can be executed.

- *string staticMethod(TreeNode x, TreeNode y):* Method used to implement the comparison between two static fault trees with each other. Returns comparison results as a string.
- *string dynamicMethod(TreeNode x, TreeNode y):* Method used to implement an extension of the static comparison, to suppoert the comparison between dynamic fault trees. Returns comparison results as a string.

This interface allows us to easily switch between different comparison algorithms. However, it also limits us to executing only one algorithm at a time. To use multiple algorithms together, we need to run them sequentially and then merge their feedback.

5.3 Boolean Equation Algorithm

To compare two fault trees based on their boolean equation, several things need to be handled: First, we need to implement all required fault tree transformations, which will be covered in Section 5.3.1. Then, to compare the two fault trees, we need to generate truth tables for both fault trees, which is explained in Section 5.3.2. Lastly, we want to explore what kind of feedback we can generate from this method, which will be covered in Section 5.3.3.

5.3.1 Fault Tree Transformations

To ensure that the fault tree can be translated into a boolean equation, several transformation steps have to be executed: The removal of intermediate events, as well as the transformation of the VOTING gates, the FDEPs, the SPARE gates and the PAND gates. Each gate transformation will be executed one after the other, and therefore the current fault tree will be iterated multiple times. This has been done deliberately because some of the transformations will be reused in the other algorithms as well.
Unit tests have been added to ensure the correctness of these fault tree transformations.

Remove Intermediate Events

To remove all intermediate events, we must traverse the complete fault tree and search for all nodes with the TreeNodeType INTERMEDIATEEVENT. If an intermediate event has been found, we can delete it from the tree by removing the node itself from its parent's list of children and adding all the intermediate events children to the parent's list of children instead. It needs to be ensured that the parent property of the intermediate events' children will be set to the parent they have recently been added to.
In addition to this general case, there are two special cases for removing intermediate events that need to be handled differently:

1. If the root is an intermediate event, then we can make its single child the new root by deleting the child nodes' parent reference.
2. If the intermediate event is the child of a PAND gate, it needs to be ensured that the order of the children is not disrupted. To do this we ensure that all child nodes of the intermediate event will be inserted at the parent children's lists index where the intermediate event has been removed from.

Transform VOTING Gates

Each VOTING gate of the fault tree needs to be transformed into a combination of one OR and multiple AND gates.
First, we need to generate the AND gates with their respective child nodes. The amount of minimal nodes k that need to fail is still stored as the text in the VOTING Gate. Therefore, we need to calculate all combinations consisting of k out of all child nodes. This can be achieved by iterating through all possible index combinations of size k from an array of the size of all child nodes. For each combination of indices, it constructs the corresponding combination of elements from the input array and yields the result.
For each of these generated input combinations, a new AND gate will be created. Each AND gate will be added as a child node to the VOTING gate
Lastly, the type of the VOTING gate can be changed to the TreeNodeType OR.
In addition to this general transformation of VOTING gates, there are two special cases that will be handled differently:

1. If the number of minimal children that need to fail equals 1, the type of this gate can be changed to an AND gate with no further adjustments.

2. If the number of minimal children that need to fail equals the number of child nodes, the type of this gate can be changed to an OR gate with no further adjustments.

Transform FDEPs

To transform the FDEPs, the fault tree has to be iterated two times. The aim of the first ieration is to create a list of TreeNode pairs, where the first TreeNode represents the triggering node from the FDEP relationship, and the second TreeNode is the one that should be triggered. Therefore, if an FDEP gate has two triggering events A and B and three events that are triggered, X, Y and Z, six new pairs will be in total added to the list: (A, X), (A, Y), (A, Z), (B, X), (B, Y) and (B, Z). Therefore, we are looking for FDEPs in the fault tree during the first iteration to ensure that all triggering and triggered node pairs get added to the list. In addition, the FDEPs will also be deleted during this iteration, by removing the TreeNode from its parent's childlist.
The aim of the second iteration is to add each triggering node as an OR relationship to their respective triggered node. Therefore, we are looking for all nodes that are basic events during the second iteration. Once a basic event has been found, we iterate over the list of pairs of triggering and triggered nodes to check, if the triggered node's text matches the current node's text. If there is a match, a new OR gate node will be created, and both nodes of the current pair will be added as child nodes. Then we continue to iterate over the list of pairs to see if the current node has other trigger nodes, and if so, we add these trigger nodes as children to the new OR gate as well. Finally, the current node will be removed from its parents' list of children, and the new OR node will be added instead. Once the second iteration ran through, the FDEP transformation has been completed.

Transform PAND Gates

To transform a PAND gate, the type of the TreeNode needs to be changed to the TreeNodeType AND and to ensure that we can still recall that this used to be a PAND gate, the TreeNode text will be changed to "Original_PAND".
Afterwards, as a first step to compare the usage of PAND gates, we can count how many occurrences of the "Original_PAND" tag exist in each fault tree. In a second step, to

determine if two NodeTrees with the text "Original_PAND" are identical, we compare the type and text of the parent of both TreeNode and the amount, type and text of the child nodes.

Transform SPARE Gates

To transform a SPARE gate, the type needs to be changed to the TreeNodeType VOTING. Then, it needs to be calculated how many gates need to fail, so that the voting gate fails. This number equals the number of children marked as a spare child + 1. Both this number and the tag "Original_SPARE" will be added to the NodeTrees text.
Since the VOTING gates also need to be transformed before a boolean equation can be generated, this needs to be executed before the VOTING gate transformation.
Afterwards, we can follow the same instructions as the comparison of PAND gates: In the first step, the occurrences of the tag "Original_SPARE" are compared, and in the second step, we can evaluate if the TreeNodes with this tag are identical in both fault trees.

5.3.2 Truth Table

To generate a truth table for a fault tree, we use the Boolean Expression Parser [3], which is a C# Library that allows parsing, evaluating and generating truth tables for boolean expressions.
The first step is to generate a boolean expression string that the boolean expression parser can parse. The parser currently supports 8 logical operators, which are "and", "or", "not", "xor", "nand", "nor", "xnor" and "implies". Since we assume that all gate transformations have been executed beforehand, the fault tree will only contain TreeNodes with the TreeNodeTypes OR, AND, NOT, XOR and BASICEVENT, which can all be translated directly into elements of the boolean expression. For example, for the static failed road trip fault tree from Figure 4.5, the generated boolean equation will be

$$\begin{aligned}&(\text{No\_Connection OR No\_Battery) AND (Engine\_Fails OR}\\&((\text{Tire1 AND Tire2) OR (Tire1 AND Spare\_Tire) OR (Tire2 AND Spare\_Tire)))}\end{aligned} \tag{5.1}$$

The process of converting this logical expression into a truth table involves several steps that are included in the Boolean Expression Parser Library: tokenisation, parsing, constructing an Abstract Syntax Tree (AST), and evaluation.
First, the input expression undergoes tokenisation. In this step, the input is split into tokens, which are internal representations of the elements in the expression. Here, tokens can be logical operators, variable names, or brackets. For example, the input expression "A AND NOT B" would be tokenised into the list [A, AND, NOT, B].
Next, these tokens are parsed into a postfix notation. Postfix notation places operators after their operands, removing the need for brackets and simplifying the evaluation process. For our tokenised list [A, AND, NOT, B], the parsing step would produce [A, B, NOT, AND].
Then, we can transform the parsed tokens into an AST. An AST is a tree representation of the parsed expression, designed to enable easy traversal and evaluation. Each node in the AST represents either an operator or a variable and can only have zero, one, or two child nodes, depending on the operator type.
Finally, the AST is evaluated, which involves traversing the tree and assessing each node recursively, based on the values of its child nodes. During this evaluation, the expression

is assessed for every possible set of inputs, and the results are compiled into a truth table. Once both truth tables have been created, they can be compared. For each possible set of inputs, we count how many of the results are not identical. If all outputs are identical, the two boolean expressions are logically equivalent.

5.3.3 Feedback

If all variable allocations in the truth tables are equivalent and we did not occure any dissimilarities regarding the usage of SPARE or PAND gates, we can return feedback indicating that the students submission is logically equivalent to the teachers solution. If we detected any dissimilarities, between the usage of SPARE or PAND gate, we can output to which degree they differ in both fault trees.
When we encounter inequivelnce, calculate the difference score and the variable occurences in all not-matching variable allocations. Therefore, we iterate through the truth table again and increase an occurence counter for each variable allocated with FALSE if we encounter an inequivalenve. We can then create a variable ranking based on the difference between their values and half the total number of truth table entries.

5.4 Edit Distance Metrics

To implement the edit distance metric comparison, we made use of the C# APTED Library, which is a library that implements the APTED algorithm to calculate the edit distance metric between two trees. How this library has been used will be explained in Section 5.4.1. Since the edit distance metric calculation depends on the order of nodes, some additional node tree transformations have been added, which are explained in Section 5.4.2. Lastly, we want to explore what kind of feedback we can generate from this method, which will be covered in Section 5.4.3

5.4.1 APTED Library

The AptedSharp [2] library is a C#.NET implementation of the APTED algorithm introduced in Section 4.2.2, which is used to calculate the tree edit distance. It computes the minimum cost required to transform one tree into another and provides a mapping between tree nodes. The library includes customizable cost models and a parser for tree structures in bracket notation. The code is open-source and licensed under the MIT license.

Create Input String

To perform the tree edit distance, the APTED library has its own Node representation. Therefore, we need to translate our TreeNode into the APTED Node data structure. To do so, the APTED library implements an additional parser called *BrackerNotationParser*, which can parse trees represented in a bracket notation. To create such a bracket notation from the TreeNode datatype can be handled easily. For example, for the failed road trip fault tree shown in Figure 4.1, the bracket notation is

$$\begin{aligned}&\{\text{AND}\{\text{OR}\{\text{no\_connection}\}\{\text{no\_power}\}\}\{\text{OR}\{\text{engine\_fails}\}\\ &\qquad\{\text{VOTING\_2}\{\text{tire1\_fails}\}\{\text{tire2\_fails}\}\{\text{spare\_fails}\}\}\}\}\end{aligned} \tag{5.2}$$

To create this bracket notation string, the tree will be iterated in inorder. If the current TreeNode is a basic or intermediate event, the text of the TreeNode will be added to the

string. If the current TreeNode is a gate, the type of the gate will be added. The only exception is the VOTING gate, where both the type and the text are added.
To accommodate dynamic fault trees, as previously explained in Section 4.2.1, a "SPARE_" prefix is added to all child nodes marked as a spare for SPARE gates. Simultaneously, for nodes marked as a trigger for FDEPs, a "TRIGGER_" prefix is added to those child nodes. Following these extra rules, the bracket notation for the dynamic failed road trip is

$$\begin{gathered}\{\text{AND}\{\text{OR}\{\text{no\_connection}\}\{\text{PAND}\{\text{broken\_charger}\}\{\text{no\_power}\}\}\} \\ \{\text{OR}\{\text{engine\_fails}\}\{\text{SPARE}\{\text{tire1\_fails}\}\{\text{tire2\_fails}\}\{\text{SPARE\_spare\_fails}\}\}\} \\ \{\text{FDEP}\{\text{TRIGGER\_nails\_road}\}\{\text{tire1\_fails}\}\{\text{tire2\_fails}\}\}\}\end{gathered} \tag{5.3}$$

Calculate Edit Distance

Once the input string has been created, it can be parsed into the APTED Node representation using its BracketNotatoionParser.
Then, to calculate a distance, a cost model needs to be specified. The APTED Library allows for customization of such a cost model by implementing its *ICostModel* interface. To implement this interface, a cost for deleting nodes, a cost for inserting nodes and a cost for updating nodes need to be specified. In our implementation of the edit distance metric comparison, this ICostModel is interchangeable. This allows for experimentation of different cost functions during the evaluation of the edit distance metrics algorithm. However, if no cost model gets specified, a default ICostModel with a deletion, insertion and updating cost of 1 will be used.
Finally, we can use the APTED Library's function *ComputeEditDistance* to calculate the edit distance metric between two fault trees.

5.4.2 NodeTree Transformation

In contrast to the boolean equation comparison, the edit distance metric does not compare the logic equivalence of two fault trees. Instead, the single elements of the fault tree are compared with each other. To ensure a similar ordering of the nodes and to minimize simple logical equivalences, the NodeTree transformation steps reordering of children, minimizing of subsequent gates and minimizing according to the distributive law have been added.

Reordering Of Children

During the calculation of the edit distance, the order of the child nodes will be considered. Therefore, if a node has a list of children {A, B}, and another node has a list of children {B, A}, they will not be viewed as identical. Since the order of child nodes in fault trees is generally irrelevant, we need to ensure that we order the child nodes by specific rules:

- All gate types will be ordered before basic events.
- Basic events will be ordered alphabetically based on their name.
- Gate types will be ordered in the following order: AND, PAND, OR, XOR, NOT, VOTING, SPARE, FDEP.
- If a child list contains two gates of the same type, we create a string representation of their children. Then, these gates can be ordered alphabetically based on their string representation. For Example, if one AND node has the two children "EventA" and "EventB", its string representation would be "EventAEventB".

The only exceptions to this rule are PAND gates, where no reordering of the child nodes will occur.
To implement the reordering the C# interface IComparer [6] has been implemented.

Subsequent Gates

If two subsequent gates of type OR or AND have the same gate type, they can be combined into one gate. An example of this transformation can be seen in Figure 5.3: The fault tree on the left contains subsequent OR gates, which will be transformed in the fault tree on the right side, which only contains one OR gate.
To achieve this transformation, the complete fault tree needs to be iterated. For each node, we check if one or more of its child nodes have the same gate type as the current node. If so, the children of this child node will be added as children to the current node, whereas the affected child itself will be removed. By applying this check recursively to the child nodes first, we can ensure that this minimization also works on three subsequent gates of the same type.

Distributive Law

Another fault tree minimization we want to perform is based on the distributive law regarding the gates AND and OR. An example can be seen in Figure 5.4: The fault tree on the left side has two AND nodes that have the basic event A as a child. According to the distributive law, this can be transformed into the fault tree on the right.
To implement this minimization, the whole fault tree needs to be iterated. For each OR gate, we check if all child nodes are AND gates. If that is the case, we check if all AND gates have at least one common identical subtree. If that is not the case, the distributive law cannot be applied and we can move on. If there is however at least one common subtree, we can reorder the fault tree as follows: The type of each child AND gate becomes an OR gate, and all common subtrees will be deleted. Instead, these will be added as children to the original OR gate, and the type of this gate will be changed to an AND gate.
The same process can be applied to search for AND gates with OR gates as the child nodes.

5.4.3 Feedback

Based on the output of the APTED Library, we can generate a guide of which nodes need to be deleted, updated or inserted for the students solution to match the teachers solution. To do this, we can use APTEDs class *LoggingOperationExecuter* to get an overview of all necessary operations. The final difference score is equivalent to the number of transformation steps.

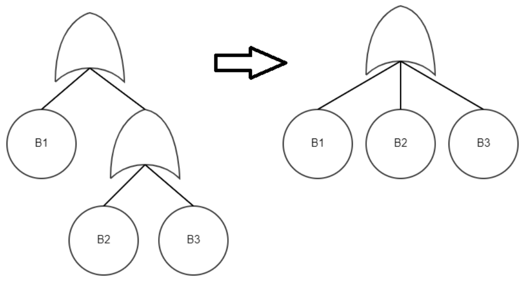


FIGURE 5.3: Example of the subsequent gate transformation

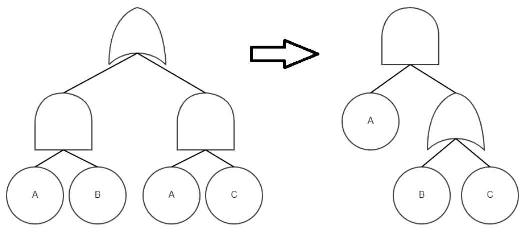


FIGURE 5.4: Example of the distributive law transformation

5.5 Pattern Matching

The third comparison algorithm that has been implemented is pattern matching. As introduced in Section 4.3, the implemented algorithm is a greedy algorithm, that iterates through one fault tree and matches the most similar node of the second fault tree to this node. Once all TreeNodes have been matched, an overall difference score can be calculated.

5.5.1 Difference Score

To create a similarity ranking between TreeNodes, a cost function is needed to calculate a difference score between two TreeNodes. Such a difference score can take different properties of a NodeTree into consideration, e.g. the type, the text and the number of child nodes.
To experiment with different difference score functions during the evaluation, an interface for such a score function has been created. Any implementation of this interface can be passed as a parameter to the comparison. If the pattern-matching comparison is called without any cost function, the cost function defined in Section 4.3 will be used, which compares the text, type and level of the TreeNode, as well as its children.

5.5.2 Variants

To further evaluate the pattern-matching algorithm, three different variants have been implemented. These variants differ, among others, in regards to which TreeNodes of the second fault tree are considered to match the current TreeNode of the first fault tree. The first variant is the one that has already been described above in Section 5.5, where all currently unmatched TreeNodes of the second fault tree are considered possible matches.
In the second variant, we pre-select the possible matches based on their gate type. This means, that we only consider those TreeNodes from the second fault tree as possible matches if they have the same TreeNoteType as the current TreeNode. By doing so, we want to evaluate if we can gather more information about which TreeNodeTypes cause the biggest increase in the distance score.
In the third variant, the matching happens in a two-step procedure: First, we iterate through the first fault tree and try to find matches for each node that have a difference score of 0. If such a TreeNode can be found, they will be matched and the iteration continues. If no TreeNode with a difference score of 0 can be found, the TreeNode remains unmatched. In a second iteration, we repeat the iteration through the first fault tree for all TreeNodes that have not been matched the first time. In this iteration, we again look for the TreeNode of the second fault tree with the smallest difference score. This time, this TreeNode will be matched regardless of the height of the score.

5.5.3 Feedback

To generate additional feedback for the pattern-matching approach, we identify which node matches contribute to an increase in the difference score. Each matching increasing this score implies that those nodes are not identical. Therefore, we return information about both nodes including the type, name and the children to highlight at which point they differ. Additionally, we return all nodes that remain unmatched from the student's submission, if this fault tree contains fewer elements than the teacher's solution. If the student's submission contains more elements, we return those unmatched from the teacher's solution instead.

Chapter 6

Testing Data

To evaluate the effectiveness of the three different comparison algorithms implemented in Section 5, we created a test set containing various UTML files. This test set builds the foundation of the evaluation in Section 7. The design of this test set will be the subject of Section 6.1. Section 6.2 introduces a workaround to use UTML to support all currently unsupported gates as well.

6.1 Testset

The test set contains several different scenarios modelled as a fault tree. For each of these scenarios, we created multiple fault tree variants. One of these fault trees has been marked as the correct solution and is, therefore, the fault tree to which all other variants will be compared.
The scenarios are divided into two different complexity categories. In the first category, the focus of the scenarios relies upon evaluating the effectiveness of the comparison algorithms for one specific gate, which will be described in Section 6.1. These examples are small and may not capture the full complexity of student exercises. However, by concentrating on a single gate type, they enable us to confirm whether the comparison logic has been implemented accurately. For instance, they can help verify whether the PAND gate is correctly processed for the BEC.
The second category addresses more complex fault trees, aiming to demonstrate how the comparison algorithms evaluate scenarios that reflect the complexity often encountered in actual student exercises. These will be introduced in Section 6.1.2. Some of these examples build on typical scenarios used to explain the concept of fault trees, such as Example 1 and Example 2. Additionally, Example 5 is drawn from a real student exercise in the STAR course, with test case scenarios extended to include model choices made in student submissions. By evaluating these test cases, we aim to assess how beneficial their outcomes are in a classroom setting. Specifically, we want to determine whether the ranking based on difference scores is fair and if the additional feedback serves as a valuable aid for providing constructive comments to students. Additionally, we seek to verify if equivalence is detected in more complex fault trees and if multiple errors in a student's solution can be accurately identified and differentiated.

6.1.1 Single Gate Tests

For each gate type, a minimalist fault tree has been created. An example of the correct version of the AND gate test fault tree can be seen in Figure 6.1, consisting of a single AND

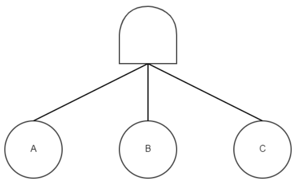


FIGURE 6.1: AND Gate Test scenario

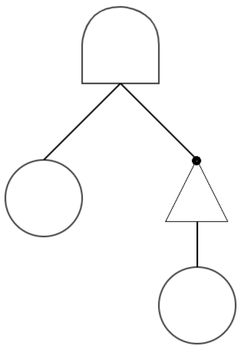


FIGURE 6.2: NOT gate test scenario

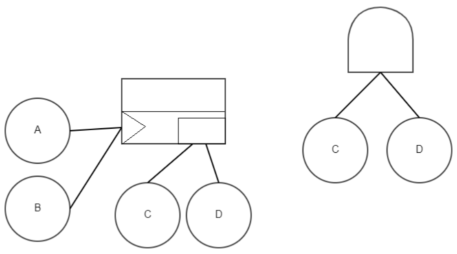


FIGURE 6.3: FDEP test scenario

gate with three basic events. The test cases for the gate types OR, XOR and PAND are constructed similarly. An example of the NOT gate scenario can be seen in Figure 6.2. To allow for a bit more complexity of the NOT logic, another AND gate has been added to the scenario. Lastly, an example of the FDEP scenario can be seen in Figure 6.3, containing the FDEP with two triggering and two triggered basic events and a single AND gate with the triggered basic events as children. The test sets for the VOTING and the SPARE gate look similar except that they are missing the additional AND gate.
Then, for each of the single gate scenarios, different variants have been created. An overview of those variants can be seen in Table 6.1: If the column is highlighted green, a variant has been created fitting the description. The test cases cover both variants that are logically equivalent to the solution, as well as variants that are not equivalent. Some columns include extra information about the scenario. For example, test case 7 "Switched gate type" specifies which type of gate has been used instead. If a column includes a "+", two different tests have been created for this scenario. The last row summarises, how many test cases have been created in total for each gate type.

		AND	OR	XOR	NOT	VOTING	PAND	SPARE	FDEP
1	Identical solution	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
2	Different order of nodes	🟩	🟩	🟩		🟩	🟩	🟩	🟩
3	With intermediate events	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
4	Additional child element	🟩	🟩	🟩	🟩	🟩	🟩	🟩 Normal + Spare	🟩 Trigger + Triggered
5	Missing child element	🟩	🟩	🟩	🟩	🟩	🟩	🟩 Normal + Spare	🟩 Trigger + Triggered
6	Switched child element	🟩	🟩	🟩	🟩	🟩	🟩	🟩 Normal + Spare	🟩 Trigger + Triggered
7	Switched gate type	🟩 OR	🟩 AND	🟩 OR		🟩 OR + AND	🟩 AND	🟩 OR + AND	🟩 VOTING
8	Add another gate as a child; logically equivalent	🟩	🟩	🟩			🟩		
9	Add another gate as a child; logically inequivalent due to different gate type	🟩	🟩	🟩			🟩		
10	Logic equivalent with static gates					🟩		🟩	🟩 two variants
11	Higher threshold					🟩		🟩	🟩
12	Lower threshold					🟩		🟩	🟩
Total test cases:		9	9	9	5	11	9	14	14

TABLE 6.1: Test cases for specific gate types.

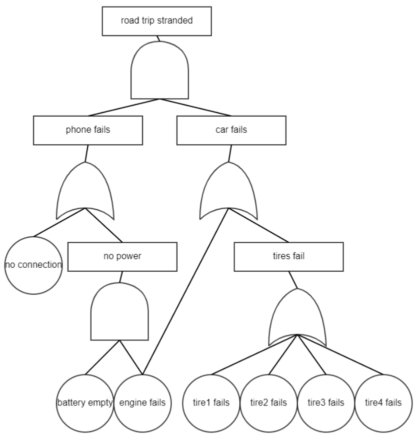


FIGURE 6.4: Correct static fault tree: Example 1

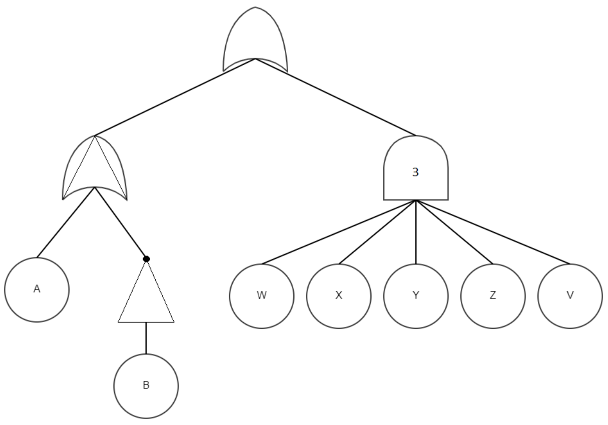


FIGURE 6.5: Correct static fault tree: Example 2

6.1.2 Complex Scenarios

Next to the tests focusing on a specific gate type, five more test scenarios have been created containing more complex fault trees. The focus of these test sets is for one to simulate how a fault tree exercise that students should solve might look like. Second, the fault tree variants also combine multiple types of error types in one variant so that it can be evaluated whether these errors can be distinguished or not.

Static Example 1

The first static example can be seen in Figure 6.4. The system described in the fault tree is similar to the road trip used during the algorithm explanation in Figure 4.1, with slight differences, e.g. not having a spare tire. This example focuses on the dynamic between the gates OR and AND since these are gates often used in fault trees. In addition, this fault tree model also includes intermediate events and assigns one basic event to multiple parents. The test cases in this scenario also aim to validate equivalence between the

	Error type	**Difference to main FT**
1	Identical Solution	
2	Different ordering	
3	No duplicate use of BEs	extra "engine fails" element
4	Joined OR gates	tire BEs and "engine fails" have the same parent OR
5	No intermediate events	
6	Additional existing BE	"no connection" added to "tires fail" OR gate
7	Additional new BE	"Spare tire fails" added to "tires fail" OR gate
8	Missing BE	"tire 1 fails" missing from "tires fail" OR gate
9	Missing subtree	AND gate and "no connection" missing from "phone fails"
10	Negated fault tree	All OR gates as AND gates; All AND gates as OR gates
11	Switched OR Gate	"tires fail" OR as AND
12	Switched TL Event	TL Event AND to OR
13	Switched BE	Switched "tires4 fail" to "no connection"
14	Switched BE	Switched "tires2 fail" to "tires3 fail"

TABLE 6.2: List of fault tree variants for the static road trip: Example 1.

	Error type	**Difference to main FT**
1	Identical solution	
2	Higher threshold for VOTING	Threshold: 4
3	Higher threshold for VOTING	Threshold: 5
4	Lower threshold for VOTING	Threshold: 2
5	Lower Threshold and missing BE for VOTING	Threshold: 1, Missing: "V"
6	Missing BE for VOTING	Missing: "V"
7	Missing top-level OR	VOTING set as child of XOR
8	Switched top-level OR	TL OR as AND
9	Missing XOR	"A" and "NOT" as TL OR children
10	Switched TL OR + Switched XOR	TL OR as AND; XOR as OR
11	Switched TL OR + Switched VOTING	TL OR as AND; VOTING as OR
12	Switched TL OR + Higher Threshold VOTING	TL OR as AND; Threshold: 4
13	Missing XOR + Missing BE for XOR	Missing: "A"; NOT as TL OR child
14	Missing XOR + Missing BE for XOR + Switched VOTING	Missing: "A"; NOT as TL OR child; VOTING as OR
15	Missing NOT gate + Missing BE for VOTING	"B" as child of XOR; Missing: "V",
16	Missing NOT gate + Missing BE for VOTING + Switched top level OR	"B" as child of XOR; Missing: "V"; TL OR as AND
17	Missing BE for VOTING + Switched XOR + Switched top level OR	Missing "V"; XOR as OR; TL OR as AND
18	Switched TL OR + Switched VOTING + Missing XOR + Missing BE for XOR	TL OR as AND; VOTING as OR; Missing "A"; NOT as TL OR child

Table 6.3: List of fault tree variants for the static fault tree example 2

different variations. A description of all variants that the correct solution will be compared to can be viewed in Table 6.2. The first column describes the error type, whereas the second column specifies the difference to the main fault tree.

Static Example 2

The second scenario of a static fault tree can be seen in Figure 6.5. Its focus is to evaluate how different static gate types interact and therefore contains four different types of static gates: OR, XOR, NOT and VOTING. A description of all variants that the correct solution will be compared to can be viewed in Table 6.3.

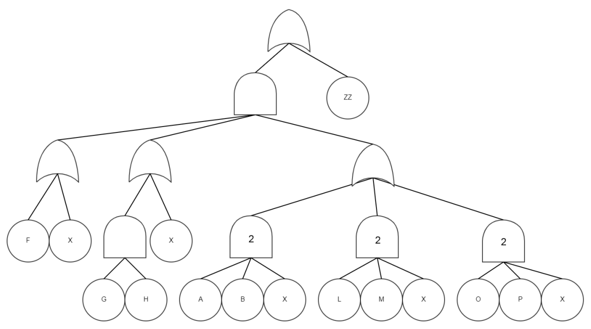


Figure 6.6: Correct static fault tree: Example 3

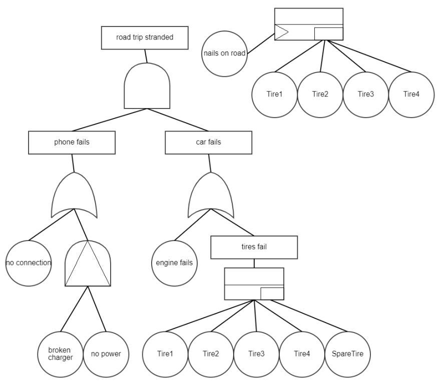


Figure 6.7: Correct dynamic fault tree: Example 5

	Error Type	**Difference to main FT**
1	Identical solution	
2	Switched BE	BE "G" as "A"
3	Switched BEs	"B" as "M", "P" as "B" and "M" as "P"
4	Switched BEs	All "X" in VOTINGs as "ZZ"
5	Switched OR	OR (above VOTINGs) as AND
6	Missing Subtree	Missing (OR (AND ("G" "H"))"X") subtree
7	Missing AND + Missing BE	Missing 4th level AND and its child "H", "G" as child of OR
8	Missing TL OR + Missing BE	Missing TL OR and its child "ZZ"
9	Switched VOTINGs	All VOTINGs as ORs
10	Switched VOTINGs + Missing BEs	All VOTINGs as ORs, Missing "B", "M" and "P" of VOTINGs
11	Switched VOTINGs + Missing subtree	All VOTINGs as ORs, Missing (OR (AND (G H))X) subtree
12	Switched VOTINGs + Missing AND + Missing BE	All VOTINGs as ORs, Missing 4th level AND and its child "H"
13	Switched VOTINGs + Missing TL OR + Missing BE	All VOTINGs as ORs, Missing TL OR and its child "ZZ"
14	Joined OR gates	Two OR children of the AND combined into one OR
15	Joined OR gates + 1st VOTING as static gates	Two OR children of the AND combined into one OR; 1st VOTING replaced by static gate equivalent
16	Joined OR gates + 1st & 2nd VOTINGs as static gates	Two OR children of the AND combined into one OR; 1st + 2nd VOTING replaced by static gate equivalent
17	Joined OR gates + all VOTINGs as static gates	Two OR children of the AND combined into one OR; all VOTING replaced by static gate equivalent

TABLE 6.4: List of fault tree variants for the static example 3.

Static Example 3

The third static example contains a more complex fault tree than the previous two examples and can be viewed in Figure 6.6. It consists of multiple AND and OR gates, and three different VOTING gates. With this test case, we want to explore how the comparison algorithms handle recurrences by having basic events reappear in multiple subtrees. It also explores the significance of the hierarchy of basic events by having a basic event as a direct child of the top-level event. All specified test variants are listed in Table 6.4.

Dynamic Example 4

The first dynamic example, which can be viewed in Figure 6.7, is similar to the dynamic road trip example used during the algorithm description in Figure 4.3, with the difference that four regular tires have been modelled instead of two. This fault tree contains all three different dynamic gates. Therefore, it allows us to evaluate how the different comparisons handle the interaction of these different gate types. A description of the variants is given in Table 6.5.

Dynamic Example 5

The second dynamic fault tree example is based on an exercise that students have been asked to solve after learning about dynamic fault trees and can be viewed in Figur 6.8. The fault tree contains multiple FDEPs as well as multiple PAND gates. It also has basic events reappear in multiple subtrees, so it provides a basis for evaluating how the different comparison algorithms react to the recurrence of basic events with dynamic gates. All specified test variants are listed in Table 6.6.

	Error Type	**Difference to main FT**
1	Identical solution	
2	No intermediate events (IE)	
3	No top-level intermediate event	
4	Switched BE	"no power" as "no battery"
5	Switched BE	"no power" as "engine fails"
6	Switched FDEP Trigger	"nails on road" as "nails"
7	Switched BE + Switched FDEP Trigger	"no power" as "no battery", "nails on road" as "nails"
8	Switched PAND	PAND as AND
9	Switched SPARE gate	SPARE as VOTING
10	Switched TL AND	TL AND as OR
11	Switched TL AND	TL AND as PAND
12	Higher threshold FDEP	Threshold: 2
13	Higher threshold SPARE	Threshold: 4
14	Missing AND	Missing "phone fails" AND, "no connection" as PAND child
15	Missing PAND	"broken charger" and "no power" added to parent AND
16	Missing AND + Missing BE	Missing "car fails" AND, Missing "engine fails", "tires fail" as child of "car fails"
17	Missing TL AND + Missing subtree	Missing TL AND, Missing subtree "phone fails"
18	Missing AND + Higher threshold SPARE	Threshold: 2, Missing "phone fails" AND, "no connection" as child of PAND
19	FDEP integrated + No intermediate events	FDEP logic added with equivalent ORs
20	FDEP integrated+ Additional triggered BE + No intermediate events	FDEP logic added with equivalent ORs, Additional triggered BE "SpareTire" of FDEP
21	FDEP integrated + Duplicate use of trigger BE + No intermediate events	FDEP logic added with equivalent ORs, Reuses trigger BEs "nails on road" for multiple parents
22	Duplicate use of triggered BE + No IE	FDEP reuses triggered BEs from SPARE
23	Duplicate use of triggered BE + Additional triggered event + No IE	FDEP reuses triggered BEs from SPARE, additional triggered event "SpareTire" for FDEP

Table 6.5: List of fault tree variants for the dynamic road trip example 4.

6.2 UTML script

The current UTML version does not support all gate types that are included in our comparison. Therefore, a workaround is needed to allow UTML to display all gate types and have the extracted JSON file in the correct format.
In UTML, intermediate events shall be used to represent the missing gates, with a text that indicates which type of gate it is replacing. An example is shown in Figure 6.9.

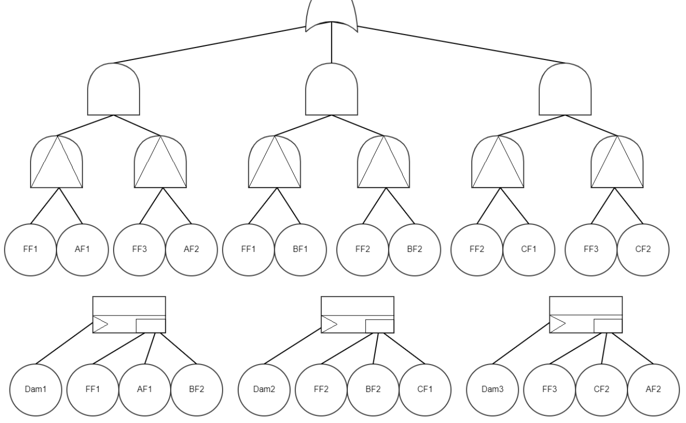


Figure 6.8: Correct dynamic fault tree: Example 4

	Error Type	**Difference to main FT**
1	Identical solution	
2	Add Intermediate events	
3	FDEP different ordering	Different ordering of triggered BE
4	FDEPs integrated	FDEP logic added with equivalent ORs
5	FDEPs integrated, distributive law applied	FDEP logic added with equivalent ORs, DL applied afterwards
6	Duplicate use of triggered BEs	FDEP reuses triggered BEs from PANDs
7	Switched PAND	All PAND as SPARE with threshold 1
8	Switched PAND, Duplicate use of triggered BEs	All PAND as SPARE with threshold 1 FDEP reuses triggered BEs from PANDs
9	Missing triggered BE	All FDEPs miss one triggered BE: "FF1", "FF2". "FF3"
10	Switched AND	all Level 2 ANDs as OR
11	Switched BE	One instance of "FF3" as "FF2" for PAND
12	Switched BE all	All instances of "FF3" as "FF2"
13	Switched TL OR	TL OR as VOTING with threshold 1
14	Additional VOTING gates and BEs	Add new TL OR with new VOTING subtree (using existing BE)
15	Additional SPARE with new BEs	one SPARE added to each PAND, spare BE is new element
16	Additional SPARE with existing BE	one SPARE added to each PAND, spare BE is existing element
17	Additional Spare gates existing BE + Switched AND gates	one SPARE added to each PAND, spare BE is existing element, All Level 2 ANDs as OR

TABLE 6.6: List of fault tree variants for the dynamic fault tree example 5 based on a student exercise.

- To represent an XOR gate, use an intermediate event with the text "XOR".
- To represent a NOT gate, use an intermediate event with the text "NOT".
- To represent a VOTING gate, use an intermediate event with the text "VOTING k", where k is the number of minimal input gates that need to fail for the voting gate to fail.
- To represent a PAND gate, use an intermediate event with the text "PAND". The order of the child nodes will be determined based on the x-coordinate of the position of the child node elements. For example, in Figure 6.9, the basic event "charger broken" is the leftmost child, and therefore the first child of the PAND node.

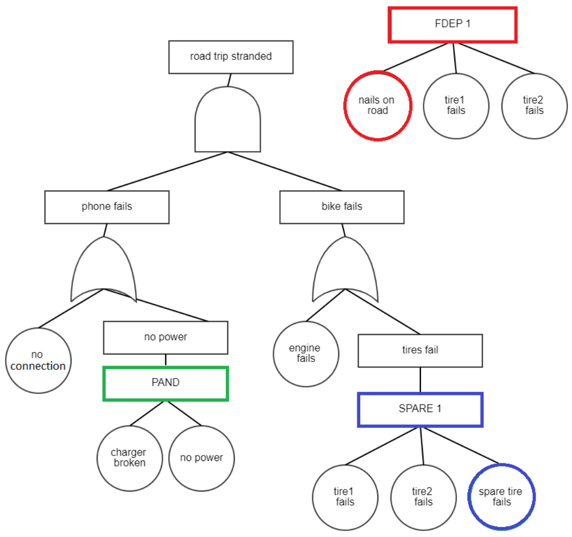


FIGURE 6.9: Example of UTML workaround to use unsupported gates

- To represent an FDEP, use an intermediate event with the text "FDEP k", where k represents the number of child nodes that should be displayed as triggers. The k leftmost child nodes regarding the x-coordinate of their position will read as the trigger nodes. An example can be seen marked in red in Figure 6.9, where the basic event "nails on road" represents the trigger.

- To represent a SPARE gate, use an intermediate event with the text "SPARE k", where k represents the number of child nodes that should be displayed as spares. The k rightmost child nodes in regards to the x-coordinate of their position will read as the spare nodes. An example can be seen marked in blue in Figure 6.9, where the basic event "spare fails" represents the spare element.

After the fault tree has been modelled in UTML as described above, we can execute a script called "UTMLConverter". For each gate disguised as an intermediate event, this script changes the JSON property type from "RectangleNode" to the respective node based on the JSON property text. For the gate types that require the childNodeInfo, the needed IDs of the child nodes will be added.
After the script has been executed, the modified JSON file will not be allowed to be imported into UTML anymore. Therefore, the script creates a copy of the file to modify, and the original file stays unchanged. If the fault tree requires modification, the original file can be imported and changed in UTML, then run through the "UTMLConverter" script again.

Chapter 7

Evaluation

In this chapter, we evaluate the three comparison algorithms introduced in the preceding chapters. The primary objective is to assess the accuracy of these algorithms using the test data sets described in Section 6. By systematically analyzing and comparing the results, we aim to identify the strengths and weaknesses of each algorithm in various scenarios. A highlight of the main findings is summarized in Section 7.4 .

7.1 Boolean Equation Algorithm

The first comparison algorithm to be evaluated is the boolean equation comparison. Since the boolean equation difference score is calculated based on the percentage of equivalent truth table allocations, all results for this method will yield a score between 0 and 1. If the distance score is 0, it implies that the boolean equation comparison detected an equivalence between the compared solutions. Next to the difference score, the results also include if the post process detected a difference in the usage of SPARE or PAND gates. If a difference in SPARE is detected, there will be an "S" next to the score, and for a difference in PAND gates a "P". The addition ">" or "<" indicates, if it is more or less than the correct solution. For example "< S" expresses that the test case has fewer SPARE gates than the correct solution.
The results in Table 7.1 to Table 7.6 are colour-coded from dark green to dark red, where the difference score is represented as follows: Dark green indicates a score of 0, light green between 0 and 0.1, yellow between 0.1 and 0.2, orange between 0.2 and 0.3, light red between 0.3 and 0.5 and dark red implies a score higher than 0.5. If dissimilarities in the usage of PAND or SPARE are detected, the colour category will be increased by one category. For example, if the difference score is zero, but a missing PAND is detected, the colour of the column will be light green instead of dark green.

7.1.1 Single Gates Evaluation

First, all single gate tests have been executed for the boolean equation algorithm. Table 7.1 shows the difference scores for all different gate types.
First of all, it can be observed that those events that are logically equivalent are also perceived as logically equivalent by having a difference score of 0. This includes most of the test cases from 1 to 3, 8 and 10. Only for the FDEP regarding the reordering of nodes in test case 2, the difference score is greater than 0. However, for this test case equivalence has not been expected, since the reordering happened between nodes that are triggering and nodes that are triggered. This means that, during the FDEP gate transformation step,

		AND	OR	XOR	PAND	NOT	VOTING	SPARE	FDEP
1	Identical solution	0	0	0	0	0	0	0	0
2	Different order of nodes	0	0	0	0		0	0	0,44
3	With intermediate events	0	0	0	0	0	0	0	0
4	Additional child element (Beginning)	0,06	0,06	0,50	0,06	0,50	0,13	0,19	0,19
4	Additional child element End							0,19	0
5	Missing child element (Beginning)	0,13	0,13	0,50	0,13	0,50	0,19	0,19	0,44
5	Missing child element End							0,19	0,38
6	Switched child element (Beginning)	0,13	0,13	0,50	0,13	0,50	0,19	0,19	0,19
6	Switched child element End							0,19	0,38
7	Switched gate type	0,75	0,75	0,38	0; < P		0,63	0,25; < S	0,56
7	Switched gate type						0,25	0,62; < S	
8	Add another gate as a child; logically eqivalent	0	0	0	0; < P				
9	Add another gate as a child; logically inequivalent due to different gate type	0,25	0,25	0,25	0				
10	Logic equivalent with static gates						0	0; < S	0
10	Logic equivalent with static gates, DL								0
11	Higher threshold						0,38	0,25	0,44
12	Lower threshold						0,25	0,38	0,19

TABLE 7.1: Boolean Equation difference score for all single gate test cases

the wrong basic event was appended to the wrong location. Therefore, it is positive that the boolean equation comparison detected this swap.
Furthermore, as expected, the table shows occurrences in test cases PAND 7, PAND 8 and SPARE 10, where the truth table comparison returns an equivalence, but where the post-process detects a difference in the usage of either the SPARE or PAND gates. For example, in those PAND test cases, we can conclude that either an AND gate has been added as a child node to the existing PAND, with AND child nodes originally belonging to the PAND gate. Or the PAND has been replaced by an AND gate. Therefore, it is evident why an equivalence has been detected and why the post-process found differences. Generally, it is positive to note that the comparison detects those dissimilarities in the usage of SPAREs and PANDs, especially if the fault trees are otherwise identical. Because only then is it possible to give feedback to the student, instead of mistakenly marking the attempt as correct.
On the other hand, we also see occurrences in both SPARE 7 test cases, where the difference score is greater than 0 and a missing SPARE gate has been detected. Although this outcome was anticipated, the difference score alone already indicated that the two fault trees are not equivalent. Here, the suggestion that a SPARE gate is missing serves more as an additional hint to help the student identify their mistake.
We can also identify test cases where an equivalent was detected, even though this was not necessarily expected. First, in PAND 9, the comparison concludes that A PAND B PAND C, which is using a single PAND gate, is logically equivalent to A PAND (B PAND C), using two PAND gates. However, the associative law does not hold for PAND gates, and therefore this should not be equivalent. However, due to the conversion into AND gates, this inequivalence does not get detected. Second, regarding FDEP 4.2, where an additional element has been added as a triggered node, the comparison also calculated an equivalence. This is because the additional node is a basic element that does not occur anywhere else in the fault tree, which means that it disappears after the transformation into static gates. If this had been a basic element that occurred elsewhere, the difference score would have been greater than 0. While we would have preferred to catch this occurrence, we do not anticipate this type of error — e.g. where a previously unknown basic event is added as a

triggered event — to happen frequently.
Except for these two test cases, in all other cases where an inequivalence was expected, the comparison algorithm detected one. However, getting a fair ranking based on these difference scores alone proves to be challenging. For example, we can see that the same error type, e.g. test case 7 "switched gate type" can lead to vastly different scores depending on which gate types have been switched. For instance, if a VOTING gate is incorrectly switched to an OR gate, the score is 0.25, whereas if it is replaced with an OR gate, the score is 0.63. However, we would prefer both scenarios to yield the same score, as we classify these mistakes in the same category. Such examples can also be seen within one test set, for example in the SPARE 7.2 and the VOTING 7.2 test cases.
Overall, this set of tests demonstrates that equivalent solutions are generally identified at a satisfactory level. However, developing a fair scoring system to rank these scenarios based on their difference score remains challenging. Additionally, we observe that the use of SPARE and PAND post-processing functions as intended in these small-scale fault trees.

7.1.2 Complex Test Case Evaluation

After the single gate tests have been evaluated, the five different complex test cases will be assessed one after the other. For each of these test cases, it will be examined if equivalence has been assigned rightfully and if the ranking of difference score provides useful insights.
Example 1: Static road trip. The focus of the static road trip test set is to evaluate if equivalence can be detected in a more complex fault tree. The results of the boolean equation comparison can be viewed in Table 7.2. We can observe that all logically equivalent test cases return a difference score of 0, whereas all logically inequivalent cases return a score greater than 0. In addition, we could categorise the difference scores based on their error type roughly: Every score between 0 and 0.2 is an error related to single basic events, whereas everything above 0.2 is related to at least one mistake in the structure of the fault tree.
Example 2: Static all gate types. The focus of the second static test set is to evaluate the dynamics of all different static gate types, and the results are displayed in Table 7.3.

	Error Type	**DS**
1	Identical Solution	0
2	Different ordering	0
3	No duplicate use of BEs	0
4	Joined OR gates	0
5	No intermediate events	0
6	Additional existing BE	0,02
7	Additional new BE	0,01
8	Missing BE	0,02
9	Missing Subtree	0,36
10	Negated Faulttree	0,25
11	Switched OR Gate	0,22
12	Switched TL Event	0,38
13	Switched BE	0,02
14	Switched BE	0,02

Table 7.2: BEC Example 1: static road trip

	Error Type	**DS**
1	Identical solution	0
2	Higher threshold for VOTING	0,16
3	Higher threshold for VOTING	0,23
4	Lower threshold for VOTING	0,16
5	Lower threshold and missing BE for VOTING	0,09
6	Missing BE for VOTING	0,09
7	Missing top-level OR	0,25
8	Switched top-level OR	0,50
9	Missing XOR	0,13
10	Switched TL OR + Switched XOR	0,38
11	Switched TL OR + Switched VOTING	0,27
12	Switched TL OR + Higher Threshold VOTING	0,66
13	Missing XOR + Missing BE for XOR	0,25
14	Missing XOR + Missing BE for XOR + Switched VOTING	0,25
15	Missing NOT + Missing BE for VOTING	0,59
16	Missing NOT gate + Missing BE for VOTING + Switched top level OR	0,59
17	Missing BE for VOTING + Switched XOR + Switched top level OR	0,52
18	Switched TL OR + Switched VOTING + Missing XOR + Missing BE for XOR	0,50

Table 7.3: BEC Example 2: static all gates

In this test set, the only equivalent solution is test 1, therefore all other tests have been rightfully assigned as not equivalent. By taking a look at the test cases 2 to 9 with single differences between the fault tree, we can observe a relationship between the difference score and the level where the error occurred: If the difference score is low, the error happened at the basic event level, and as the difference score gets higher, the gates tree level gets lower. However, once multiple errors happen in one test scenario, this ranking does not hold. We can also observe how the score changes when additional mistakes are introduced to an existing one. For example, test case 8 receives a score of 0,5 for a switched top-level event. Test cases 10 through 12 introduce extra mistakes. For example, Test case 10, includes both a switched top-level event and a switched XOR gate, but it results in a score of 0,38. This highlights the challenge of providing a fair scoring ranking. Ideally, test case 8 should receive a higher score compared to test cases 10 through 12.

Example 3: Static more compelx. The third example aims to evaluate how the comparison handles an even more complex static fault tree with recurring basic events, the calculated difference score can be seen in Table 7.4. It is positive to note that only the identical solution is marked as equivalent since it is the only test case providing an equivalent fault tree. Additionally, based on test cases 14 to 17, we can observe that the VOTING gate transformation also results in equivalent fault trees: While they are not equivalent to the solution, all differences between the four test cases are equivalent. Otherwise, the difference score provides as little context as the previous examples: It is difficult to create a fair ranking since we see a difference in the score for test cases with the same category, for example in test cases 3 and 4, which both contain a switched basic event. And, we can also observe that the stacking of error results in a decrease in the difference score, for example in test cases 9 and 10.

Example 4: Dynamic all gate types. The focus of the fourth test set is to evaluate the dynamics of all different dynamic gate types and the equivalence between the dynamic gates and the static counterparts. The comparison results are shown in Table 7.6. Again, all test cases that model an equivalent fault tree are detected as such. In the only occurrence in test case 11 where a difference score of 0 occurs, even if the solutions are not equivalent, the post-process signals that the correct solution contains more PAND gates than the test case. Additionally, in all other test cases where we expect the post-process to find differences, those differences are found successfully.

Otherwise, the difference score provides no further useful information. It is not possible to

	Error Type	**DS**
1	Identical solution	0
2	Switched BE	0,02
3	Switched BEs	0,01
4	Switched BEs	0,10
5	Switched OR	0,16
6	Missing Subtree	0,05
7	Missing AND + BE	0,02
8	Missing TL OR + Missing BE	0,24
9	Switched VOTINGs	0,02
10	Switched VOTINGs + Missing BEs	0,01
11	Switched VOTINGs + Missing Subtree	0,11
12	Switched VOTINGs + Missing AND + Missing BE	0,05
13	Switched VOTINGs + Missing TL OR + Missing BE	0,24
14	Joined OR gates	0,07
15	Joined OR gates + 1st VOTING as static gates	0,07
16	Joined OR gates + 1st & 2nd VOTINGs as static gates	0,07
17	Joined OR + all VOTINGs as static gates	0,07

Table 7.4: BEC Example 3: Static complex

	Error Type	**DS**
1	Identical solution	0
2	Add Intermediate events	0
3	FDEP different ordering	0
4	FDEPs integrated	0
5	FDEPs integrated, distributive law applied	0
6	Duplicate use of triggered BEs	0
7	Switched PAND	0; < P; > S
8	Switched PAND + Duplicate use of triggered BEs	0; < P; > S
9	Missing triggered BE	0,37
10	Switched AND	0,28
11	Switched BE	0,03
12	Switched BE all	0,09
13	Switched TL OR	0
14	Additional VOTING gates and BEs	0
15	Additional SPARE with new BEs	0,45; > S
16	Additional SPARE with existing BE	0; > S
17	Additional Spare gates existing BE + Switched AND gates	0,16; > S

Table 7.5: BEC Example 5: Dynamic lecture exercise

	Error Type	**DS**
1	Identical solution	0
2	No intermediate events (IE)	0
3	No top-level intermediate event	0
4	Switched BE	0,12
5	Switched BE	0,12
6	Switched FDEP Trigger	0,03
7	Switched BE + Switched FDEP Trigger	0,15
8	Switched PAND	0; $<$ P
9	Switched SPARE	0; $<$ S
10	Switched TL AND	0,39
11	Switched TL AND	0; $>$ P
12	Higher threshold FDEP	0,01
13	Higher threshold SPARE	0,20
14	Missing AND	0,48
15	Missing PAND	0,24; $<$ P
16	Missing AND + Missing BE	0,03
17	Missing TL AND + Missing subtree	0,36; $<$ P
18	Missing AND + Higher threshold SPARE	0,49
19	FDEP integrated + No intermediate events	0
20	FDEP integrated+ Additional triggered BE + No intermediate events	0
21	FDEP integrated + Duplicate use of trigger BE + No intermediate events	0
22	Duplicate use of triggered BE + No IE	0
23	Duplicate use of triggered BE + Additional triggered event + No IE	0

Table 7.6: BEC Example 4: Dynamic road trip

create valid error clusters, since this test set contains multiple occasions, where we would rank the error score identically, but the comparison returns vastly different results, for example in test cases 12 and 13. Additionally, we can observe again that combining two errors can lead to a smaller difference score than one of the errors alone, for example in cases 14 and 16.

Example 5: Dynamic student exercise. The fifth example has been inspired by an exercise a student had to solve while learning the concept of fault trees. Its comparison results can be seen in Table 7.5. This test set includes many logically equivalent test cases, which have all been marked accordingly since many students were able to create a correct solution. The only unexpected equivalence occurred in test case 14, which is a test case containing unnecessary additional logic. However, upon further inspection, it turned out that even though additional logic has been added, the fault trees are equivalent. We cannot extract from this comparison, whether or not the modelled solution was minimal or not. Generally, we would also like to provide feedback for improvement. Differences in the usage of PANDs and SPAREs have been detected correctly.

7.1.3 Basic event occurences

To evaluate how we can generate feedback from this comparison technique next to the difference score, we observed the occurrences of gates in those truth table allocations, where the two fault trees generated different results. For smaller trees, where each basic event usually only occurs once, some regularities could be observed. For example, if the occurrence counter is half of the amount of different truth table allocations, this basic event is most likely not involved in the error, indicating that the error happened in a different subtree.

However, once we analyzed the more complex test cases, we also encountered cases where basic events related to the error had an occurrence counter equal to half of the amount of different truth table allocations. Additionally, it also happened frequently that basic

events unrelated to the error had a specifically high or low occurrence counter.
Since students typically have to solve more complex fault trees, we conclude that this occurrence counter proves to be too unreliable to provide useful feedback for the student.

7.1.4 Summary

On one hand, the boolean equation comparison offers useful feedback by indicating whether two fault trees are equivalent. We could observe that nearly all equivalences and inequivalences throughout the test set have been detected successfully. We only encountered a few occurrences, where an equivalence has been detected where we would have preferred a difference score greater than 0. However, after a more thorough analysis of the scenario, we understood this behaviour and concluded that the comparison remains adequate for evaluating whether a student's solution is correct.
We can also conclude that the workaround with the PAND and SPARE transformations into AND and VOTING gates proves to be a good solution to handle them in boolean equations since we were able to detect all scenarios where such a transformation occurs. Therefore, we can additionally provide feedback on whether the student needs to use more or less PAND or SPARE gates.
On the other hand, we were not able to draw any other additional feedback beyond the equivalence and the PAND and SPARE gates. First, we can conclude that a sole ranking of the different scores returns unfair results since we observed vastly different scores in test cases where we would classify the error in the same category. We also observed that the score can get reduced if multiple errors occur in one test case, compared to when only one of these errors occurs. Second, the attempt to gain further feedback based on the basic event occurrences also remains unsatisfactory.
Therefore, we will limit the feedback provided by the boolean equation comparison to the equivalence, including the usage of SPARE and PAND gates.

7.2 Edit Distance Metrics

The second algorithm to be evaluated is the edit distance metric comparison. Each test case has been executed and compared in different levels of fault tree transformation, which divides the results shown in the following tables into the cases "O", "I", "F" and "M".

- "O": Comparison executed after reordering the child nodes.
- "I": "O" and removing of all intermediate events.
- "F": "I" and the transformation of FDEPs into their static equivalent.
- "M": "F" and the minimization transformation of consecutive gates and the distributive law.

To calculate the distance score, the default cost model has been used, meaning that all three operations UPDATE, INSERT and DELETE have a cost of 1.
The results in Table 7.7 to Table 7.13 are colour-coded from dark green to dark red, representing the difference scores as follows: Dark green indicates a score of 0, light green a score of 1, yellow represents scores of 2 and 3, orange of 4 and 5, light red for 6 to 10, and dark red of scores above 10.

		AND				OR				XOR				PAND			
		O	I	F	M	O	I	F	M	O	I	F	M	O	I	F	M
1	Identical solution	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Different order of nodes	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2
3	With intermediate events	3	0	0	0	3	0	0	0	3	0	0	0	1	0	0	0
4	Additional child element	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	Missing child element	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	Switched child element	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	Switched gate type	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	Add another gate as a child; logically eqivalent	3	3	3	0	3	3	3	0	3	3	3	3	1	1	1	1
9	Add another gate as a child; logically inequivalent due to different gate type	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1

Table 7.7: Edit distance score for single gate test cases AND, OR, XOR and PAND

		NOT				VOTING				SPARE				FDEP			
		O	I	F	M	O	I	F	M	O	I	F	M	O	I	F	M
1	Identical solution	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Different order of nodes					0	0	0	0	2	2	2	2	2	2	5	5
3	With intermediate events	2	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
4	Additional child element (beginning)	3	3	3	3	1	1	1	1	1	1	1	1	2	2	4	1
4	Additional child element (end)									2	2	2	2	1	1	0	0
5	Missing child element (beginning)	3	3	3	3	1	1	1	1	1	1	1	1	2	2	6	5
5	Missing child element (end)									2	2	2	2	1	1	3	6
6	Switched child element (beginning)	1	1	1	1	2	2	2	2	2	2	2	2	2	2	4	2
6	Switched child element (end)									2	2	2	2	1	1	3	6
7	Switched gate type					1	1	1	1	3	3	3	3				
7	Switched gate type					1	1	1	1	3	3	3	3	3	3	5	6
10	Logic equivalent with static gates					15	15	15	15	15	15	15	15	7	7	0	0
10	Logic equivalent with static gates after Distributive Law													6	6	7	0
11	Higher threshold					1	1	1	1	1	1	1	1	1	1	5	6
12	Lower threshold					1	1	1	1	1	1	1	1	1	1	2	1

Table 7.8: Edit distance score for single gate test cases NOT, VOTING, SPARE and FDEP

7.2.1 Single Gates Evaluation

First, the single gate test cases have been evaluated. Table 7.7 shows the distance scores for the test related to the gates AND, OR, XOR and PAND, and Table 7.8 those for the gates NOT, SPARE, VOTING and FDEP.
Overall, the results of the single gate examples look very promising. For the gate types AND, OR, XOR and NOT, we end up with an edit distance score of 0 if the two compared cases are logically equivalent. In tests 4 to 7 where one basic element is out of place (e.g. missing, added or switched), the edit distance score equals 1 and a transformation script addressing this element gets generated. The only unexpected results are those related to test case 8 "Add another gate as child; logically equivalent": We expected a single delete operation of this extra event, and therefore an edit distance of 1. However, due to the reordering of the nodes, the extra event messed up the original ordering, and therefore the edit distance score results in 3.
Since no reordering happens for the PAND gate, this phenomenon does not occur for the PAND test case 8. In addition, this also results in the expected edit distance of 2 for test case 2 "Different order of nodes", where the transformation script implies the reordering of those nodes.
For the VOTING gate, the following test case 6 is interesting to highlight: "Switched child Element" has an edit distance of 2, which is higher than the result of the same test case e.g. the AND gate, where the edit distance is 1. This is again due to the reordering of the

child nodes: If the wrongly used node has a different position in the child list after sorting than the correct node would have, the transformation script contains a DELETE and an INSERT operation instead of a single UPDATE operation.
The biggest edit distance that can be observed in the examples is for test case 10 "Logic equivalent with static gates" for the VOTING and SPARE gates. The edit distance remains equal through all transformations because the transformation of those gate types into a static representation is not part of the transformation steps. For the FDEP on the other hand, the transformation has been considered, and therefore the final edit distance equals 0. On the other hand, this leads to an increase in the edit distance for other FDEP test cases, like "Additional child element".
Overall, these single gate test cases show that the use of the edit distance metric comparison is very enlightening for small fault trees. However, just ranking the distance score does not provide a good overview regarding which test case is the most similar to the correct solution. For example, for some gate types, the test case "Wrong Event Type" has an edit distance of 1, and for others an edit distance of 3. Instead, the transformation script is more helpful in providing feedback.

7.2.2 Complex Test Case Evaluation

After evaluating the single gate scenarios, the five different complex test cases will be assessed one after the other. For each of these test cases, we will examine how the edit distance evolves throughout the transformation and if the generated transformation script provides usable information.
Example 1: Static road trip. The results for this static example can be seen in Table 7.9. One focus of this test set is to validate equivalents between the fault trees. In the table, we can observe that all test cases including an equivalent scenario, which are test cases 1 to 5, end up with an edit distance of 0. Here, we can also observe that we can provide feedback for improvements to correct the fault tree: For example, in test case 4, we can still return the differences between the two solutions in the first comparison step.
For most other scenarios the expected edit distance has been calculated, based on how many nodes have been changed, added or removed, as long as we consider the lowest edit distance throughout the transformation instead of the last. In these scenarios, the transformation script for those lowest distances provides useful information to fix the mistakes. The only exceptions are test case 10 and test case 13. For test case 13, we encounter the reordering

	Error Type	**O**	**I**	**F**	**M**
1	Identical Solution	0	0	0	0
2	Different ordering	0	0	0	0
3	No duplicate use of BEs	0	0	0	0
4	Joined OR gates	4	3	3	0
5	No intermediate events	5	0	0	0
6	Additional existing BE	1	1	1	1
7	Additional new BE	1	1	1	1
8	Missing BE	1	1	1	1
9	Missing Subtree	3	2	2	7
10	Negated Faulttree	5	5	5	4
11	Switched OR Gate	1	1	1	3
12	Switched TL Event	1	1	1	5
13	Switched BE	2	2	2	2
14	Switched BE	1	1	1	1

Table 7.9: EDM Example 1: static road trip

	Error type	**O**	**I**	**F**	**M**
1	Identical solution	0	0	0	0
2	Higher threshold for VOTING	1	1	1	1
3	Higher threshold for VOTING	1	1	1	1
4	Lower threshold for VOTING	1	1	1	1
5	Lower Threshold and missing BE VOTING	2	2	2	2
6	Missing BE for VOTING	1	1	1	1
7	Missing top level OR	4	4	4	4
8	Switched top level OR	1	1	1	1
9	Missing XOR	3	3	3	3
10	Switched TL OR + Switched XOR	2	2	2	2
11	Switched TL OR + VOTING as static gate	10	10	10	10
12	Switched TL OR + Higher Threshold VOTING	2	2	2	2
13	Missing XOR + Missing BE for XOR	2	2	2	2
14	Missing XOR + Missing BE for XOR + Switched VOTING	7	7	7	3
15	Missing NOT gate + Missing BE for VOTING	4	4	4	4
16	Missing NOT gate + Missing BE for VOTING + Switched TL OR	5	5	5	5
17	Missing BE for VOTING + Switched XOR + Switched TL OR	3	3	3	3
18	Switched TL OR + Switched VOTING + Missing XOR + Missing BE for XOR	8	8	8	8

Table 7.10: EDM Example 2: static all gates

problem again, where the renaming of the node results in a different ordering of its siblings and therefore a higher edit distance. For test case 10 we encounter the phenomenon where the edit distance gets smaller but does not reach 0. This results in a transformation script for the lowest edit distance where the transformation steps leave a gap between the test case's fault tree and the correct solution. Therefore we should consider giving feedback based on the original comparison, instead of the lowest comparison.

Example 2: Static all gate types. The comparison output for the static example including all static gate types is displayed in Table 7.10. Overall, considering test cases 1 to 15 either provide the edit distance we expected or the edit distance is a bit higher due to the reordering of the nodes, and all equivalent scenarios have a difference score of 0. Here, we encounter several scenarios, where the transformation script contains more than 3 steps. Therefore we also want to examine if the script is still understandable. For example, we can observe that it remains understandable for test case 10 since most operations are DELETE operations. The only test cases where it does not immediately become clear how the fault tree needs to be transformed are test cases 16 and 18, which is due to the number of transformation steps and the usage of all three types of operations.

Example 3: Static more compelx. The first thing to notice in the comparison output in Table 7.11 is that we encounter multiple scenarios in test cases 15 to 17 where the lowest edit distance is still greater than 10. The transformation script for these tests also contains a mix of insert, update and delete functions and is therefore not intuitive to use. The same applies to test case 2, where the reordering due to a simple switch of basic event names results in a higher edit distance than expected. All other tests however result in the expected edit distance, and since all their transformation scripts only contain two different operations, the instructions remain clear.

Example 4: Dynamic all gate types. The results of the dynamic road trip example can be viewed in Table 7.13. For all test cases that are equivalent to the solution, the fault tree transformation process ends with an edit distance of 0. For all other cases, the transformation script provides clear instructions, indicating that they can either be easily followed because they have only a few operations in total, or because they mainly contain only one operation type.

Example 5: Dynamic student exercise. Compared to the other examples, Table 7.12 seems to have much higher edit distances. We also encounter many scenarios where the edit distance increases throughout the different transformation steps. By assessing the examples further, most of these high results can be reasoned with and their transformation

	Error Types	**O**	**I**	**F**	**M**
1	Identical solution	0	0	0	0
2	Switched BE	6	6	6	6
3	Switched BEs	3	3	3	3
4	Switched BEs	4	4	4	4
5	Switched OR	1	1	1	17
6	Missing Subtree	5	5	5	5
7	Missing AND + Missing BE	2	2	2	2
8	Missing TL OR + Missing BE	2	2	2	2
9	Switched VOTINGs	3	3	3	7
10	Swi. VOTINGs + Mis. BEs	6	6	6	8
11	Swi. VOTINGs + Mis. Subtree	8	8	8	15
12	Swi. VOTINGs + Mis. AND + Mis. BE	5	5	5	16
13	Swi. VOTINGs + Mis. TL OR + Mis. BE	5	5	5	9
14	Joined OR gates	4	4	4	4
15	Join. OR gates + 1st VOTING as static gates	11	11	11	11
16	Join. OR gates + 1st 2nd VOTINGs as statics	18	18	18	18
17	Join. OR gates + all VOTINGs as static gates	24	24	24	23

TABLE 7.11: EDM Example 3: Static complex

	Error Type	**O**	**I**	**F**	**M**
1	Identical solution	0	0	0	0
2	Add Intermediate events	0	0	0	0
3	FDEP different ordering	0	0	0	0
4	FDEPs integrated	39	39	0	0
5	FDEPs integrated, distributive law applied	27	27	30	0
6	Duplicate use of triggered BEs	0	0	0	0
7	Switched PAND	18	18	30	36
8	Swi. PAND, Duplicate use of triggered BEs	18	18	30	36
9	Missing triggered BE	3	3	12	18
10	Switched AND	3	3	3	13
11	Switched BE	1	1	2	20
12	Switched BE all	3	3	6	18
13	Switched TL OR	1	1	1	1
14	Additional VOTING gates and BEs	14	14	32	19
15	Additional SPARE with new BEs	12	12	12	42
16	Additional SPARE with existing BE	12	12	24	54
17	Add. SPARE with existing BE + Swi. ANDs	15	15	27	58

TABLE 7.12: EDM Example 5: Dynamic lecture exercise

	Error Type	**O**	**I**	**F**	**M**
1	Identical solution	0	0	0	0
2	No intermediate events (IE)	4	0	0	0
3	No toplevel intermediate event	1	0	0	0
4	Switched BE	1	1	1	1
5	Switched BE	1	1	1	1
6	Switched FDEP Trigger	1	1	4	4
7	Switched BE + Switched FDEP Trigger	2	2	5	5
8	Switched PAND	1	1	1	1
9	Switched SPARE gate	6	2	2	2
10	Switched TL AND	2	1	1	5
11	Switched TL AND	2	1	1	1
12	Higher threshold FDEP	1	1	7	7
13	Higher threshold SPARE	3	3	3	3
14	Missing AND	3	3	3	3
15	Missing PAND	3	3	3	3
16	Missing AND + Missing BE	6	2	2	2
17	Missing TL AND + Missing Subtree	12	9	6	6
18	Missing AND + Higher threshold SPARE	4	4	3	3
19	FDEP integrated + No intermediate events	19	15	0	0
20	FDEP integrated+ Additional triggered BE + No intermediate events	21	17	4	4
21	FDEP integrated + Duplicate use of trigger BE + No intermediate events	19	15	0	0
22	Duplicate use of triggered BE + No IE	4	0	0	0
23	Duplicate use of triggered BE + Additional triggered event + No IE	5	1	4	4

TABLE 7.13: EDM Example 4: Dynamic road trip

script still provides clear instructions. However, there are exceptions: Test cases 7 and 8 result in a complex transformation script, since variables have to be renamed from their "SPARE_x" to their original name "x".

7.2.3 Transformation script

The transformation script is the most valuable feedback returned from the edit distance metric comparison. For minor modifications, such as a missing basic event or a misplaced gate, the transformation script explicitly identifies these issues. If a larger part of the fault trees solution is missing, the transformation script will consist of only INSERT operations. And if a larger number of additional elements have been added, it will be DELETE operations instead. Therefore, next to explicitly stating what changes need to be made, the transformation script could also be used to indicate what type of error has been made.
However, if the transformation script contains more than 5 steps including a mix of all three types of operations, it gets harder to parse through. Since some of these occurrences happen because of reordering transformations, we conclude that the transformation script can still be improved. In future works, it could be evaluated if it would be possible to return partial feedback so that the student can correct their mistakes step by step.

7.2.4 Summary

Overall, the edit distance metric returned promising results. The main advantage of the edit distance metric is the transformation script. In many cases, it provides clear feedback on how the student needs to modify their fault tree to match the correct solution. Additionally, it enables us to offer feedback on how to refine their fault tree, even if the solution is already correct. However, we also encountered instances where the edit distance between two fault trees was significant enough that the transformation script became unclear. This usually occurs when it contains more than 5 steps and includes a mix of all three operations.

Regarding the identification of equivalences, the edit distance comparison detects many, but not all, equivalences between equivalent solutions. Several equivalences are only identified after comparing different transformation levels of the fault trees. However, we recognize that some equivalences remain undetected. On a positive note, this comparison did not incorrectly identify any inequivalent solutions as equivalent.
The difference score in this comparison is based on the edit distance between the two fault trees and therefore equals the amount of UPDATE, INSERT or DELETE operations that have to be executed to transform one fault tree into the other. One of the main struggles we face is that due to the set order of the nodes, we often encounter a situation where one node is deleted and inserted as a child to the same parent node. While a ranking solely based on the difference score yields fairer results than the boolean equation comparison, we still encounter test cases of the same categories that yield noticeably different difference scores.

7.3 Pattern Matching

The final comparison algorithm to be evaluated is the pattern-matching comparison. According to the algorithm description in Section 4.3, the comparison has been implemented using three distinct variants: the first variant involves comparing all gates to each other, the second variant includes a preselection of gates based on their type, and the third variant employs a two-step matching process, where the first iteration only matches gates if an equivalent gate has been found. Upon comparing these three variants, it was observed that the third variant consistently returns an equal or lower distance score. Therefore, all further evaluations will be based exclusively on the results of the third variant.
Similar to the edit distance metric comparison, each test case has been executed and compared in different levels of fault tree transformation, which divides the results shown in the following tables into the cases "O", "I", "F" and "M":

- "O": Comparison executed after reordering the child nodes.
- "I": "O" and removing of all intermediate events.
- "F": "I" and the transformation of FDEPs into their static equivalent.
- "M": "F" and the minimization transformation of consecutive gates and the distributive law.

To calculate the difference score, the in Section 4.3 proposed default cost model has been used, with a slight modification that the level of the nodes has been removed from the equation. This means that the text, type and child nodes influence the score. If a comparison yields a difference score of 0, it implies that the fault trees are identical.
Again, the results in Table 7.14 to Table 7.20 show a colour coding from dark green to dark red, representing the difference scores: Dark green indicates a score of 0, light green a score of 1, yellow represents scores of 2 and 3, orange of 4 and 5, light red for 6 to 15, and dark red of scores above 15.

7.3.1 Single Gates Evaluation

First, the single gate test cases were evaluated using pattern matching. Table 7.14 presents the difference scores for the AND, OR, XOR, and PAND gates, while Table 7.15 displays the scores for the NOT, SPARE, VOTING, and FDEP gates.

		AND				OR				XOR				PAND			
	Error Type	O	I	F	M	O	I	F	M	O	I	F	M	O	I	F	M
1	Identical solution	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Different order of nodes	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2
3	With intermediate events	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
4	Additional child element	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	Missing child element	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	Switched child element	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
7	Switched gate type	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	Add another gate as a child; logically equivalent	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1
9	Add another gate as a child; logically inequivalent due to different gate type	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

TABLE 7.14: Pattern matching score for single gate test cases AND, OR, XOR and PAND

		NOT				VOTING				SPARE				FDEP			
	Error Type	O	I	F	M	O	I	F	M	O	I	F	M	O	I	F	M
1	Identical solution	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Different order of nodes					0	0	0	0	0	0	0	0	1	1	14	6
3	With intermediate events	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	Additional child element (beginning)	2	2	2	2	1	1	1	1	1	1	1	1	1	1	4	2
4	Additional child element (end)									1	1	1	1	1	1	0	0
5	Missing child element (beginning)	12	12	12	12	1	1	1	1	1	1	1	1	1	1	14	6
5	Missing child element (end)									1	1	1	1	1	1	12	4
6	Switched child element (beginning)	2	2	2	2	2	2	2	2	2	2	2	2	2	2	12	4
6	Switched child element (end)									2	2	2	2	2	2	4	2
7	Switched gate type					2	2	2	2	1	1	1	1	4	4	18	6
7	Switched gate type					2	2	2	2	1	1	1	1				
10	Logic equivalent with static gates					4	4	4	4	2	2	2	2	5	5	16	0
10	Logic equivalent with static gates after DL													7	7	0	0
11	Higher threshold					1	1	1	1	1	1	1	1	1	1	13	5
12	Lower threshold					1	1	1	1	1	1	1	1	1	1	2	1

TABLE 7.15: Pattern matching score for single gate test cases NOT, VOTING, SPARE and FDEP

The results for the AND, OR, XOR, and PAND test sets appear quite promising. Almost all logically equivalent test cases yield a minimum difference score of 0, whereas all test cases representing a single difference from the correct solution yield a difference score of 1 or 2: A difference score of 1 is typically returned if a basic element has been added or is missing. A 2 is returned, if a basic element has been switched out. We can also observe that the ordering of the child nodes of the PAND gates influences the difference score since the PAND 2 test results in a score of 2.
In contrast, the results for the NOT, VOTING, SPARE, and FDEP gates are less intuitive, and we encounter a few higher difference scores. The first thing we can observe is that the equivalent test cases 1 to 3 are all detected as equivalent since their difference score is 0. The only exception is FDEP test 2, where an inequivalence has been rightfully detected. For test case 10, which also represents a logically equivalent solution, the equivalence gets only detected for the FDEP. This occurs, similar to the EDM, since the FDEP to static gates transformation is the only fault tree transformation added to the evaluation process. Regarding test cases 11 and 12 related to the modification of the threshold of these gates, we can observe that those changes were detected. Lastly, for cases 4 to 7, which contain single differences compared to the correct solution, we can observe a difference between the adding and deleting of basic events compared to switching basic events again. The exception of NOT case 5, with a difference score of 12, occurs because we are missing the

NOT gate, whereas the other test cases are missing a basic event.
Regarding the FDEP test cases, many tests yield a significant increase in the score after the FDEP transformation. This is also caused by the fact, that those cases miss gates and not just basic events. However, since we are interested in the lowest score, this behaviour can be disregarded.
After analyzing the matching of the nodes, we can conclude that the textual output provides sufficient feedback to guide how the errors can be removed for these test cases.

7.3.2 Complex Test Case Evaluation

Second, the more complex test cases are compared to their correct solutions, and the results are evaluated sequentially in this Section. For one, the focus of this evaluation is to check if equivalence has been detected correctly and if the difference score is reasonable. Furthermore, we want to evaluate if we can generate useful information based on the matching of the nodes that happened throughout the comparison.
Example 1: Static road trip. The focus of the static road trip test set is to evaluate if equivalence can be detected in a more complex fault tree. The results are shown in Table 7.16. First, it can be observed that only the logically equivalent test cases 1 to 5 have been marked as logically equivalent. Second, we can observe that most of the test cases 6 to 7 and 11 to 14 with a singular difference to the correct solution have their lowest difference score at either 1 or 2. In addition, based on the textual feedback, the node matchings that increase the difference score show which changes have to be made. For case 11, the score is higher than we would have originally expected. However, by looking at the matching, we can see that this happens because the error occurred in neither the top-level event nor a leaf node, and therefore the error appears while evaluating the parent and the node itself. Even though this is not ideal, we want to keep both checks in the calculation of the distance score. Considering test case 10, the node matching does not provide useful information on how to transform the fault tree. Due to the multiple switches of AND to OR and OR to AND gates, the comparison e.g. matches new AND gates to old AND gates. Lastly, for test case 9, the matching provides the information that the missing OR node and its children remain unmatched and are therefore valuable information.
Example 2: Static all gate types. For the second test set, focusing on all static gate types shown in Table 7.17, equivalence has only been detected where it was expected. For test cases 2 to 6 and 8, the difference score is reasonable and the node matching provides

	Error Type	O	I	F	M
1	Identical Solution	0	0	0	0
2	Different ordering	0	0	0	0
3	No duplicate use of BEs	0	0	0	0
4	Joined OR gates	25	15	15	0
5	No intermediate events	60	0	0	0
6	Additional existing BE	1	1	1	1
7	Additional new BE	1	1	1	1
8	Missing BE	1	1	1	1
9	Missing Subtree	22	12	12	23
10	Negated Faulttree	11	9	9	6
11	Switched OR Gate	3	3	3	2
12	Switched TL Event	2	1	1	30
13	Switched BE	2	2	2	2
14	Switched BE	2	2	2	2

TABLE 7.16: PM Example 1: static road trip

	Error Type	O	I	F	M
1	Identical solution	0	0	0	0
2	Higher threshold for VOTING	1	1	1	1
3	Higher threshold for VOTING	1	1	1	1
4	Lower threshold for VOTING	1	1	1	1
5	Lower Threshold and missing BE for VOTING	3	3	3	3
6	Missing BE for VOTING	1	1	1	1
7	Missing top level OR	14	14	14	14
8	Switched top level OR	1	1	1	1
9	Missing XOR	13	13	13	13
10	Switched TL OR + Switched XOR	4	4	4	4
11	Switched TL OR + Switched VOTING	5	5	5	5
12	Switched TL OR + Higher Threshold VOTING	2	2	2	2
13	Missing XOR + Missing BE for XOR	12	12	12	12
14	Missing XOR + Missing BE for XOR + Switched VOTING	22	22	22	28
15	Mis. NOT gate + Mis. BE for VOTING	19	19	19	19
16	Mis. NOT gate + Mis. BE for VOTING + Swi. top-level OR	20	20	20	20
17	Mis. BE for VOTING + Swi. XOR + Swi. top-level OR	5	5	5	5
18	Switched TL OR + Switched VOTING + Missing XOR + Missing BE for XOR	23	23	23	23

TABLE 7.17: PM Example 2: static all gates

	Error Type	O	I	F	M
1	Identical solution	0	0	0	0
2	Switched BE	2	2	2	2
3	Switched BEs	6	6	6	6
4	Switched BEs	6	6	6	6
5	Switched OR	3	3	3	14
6	Missing subtree	21	21	21	21
7	Missing AND + Missing BE	12	12	12	12
8	Missing TL OR + Missing BE	10	10	10	10
9	Switched VOTINGs	12	12	12	42
10	Swi. VOTINGs + Mis. BEs	15	15	15	36
11	Swi. VOTINGs + Mis. subtree	29	29	29	69
12	Swi. VOTINGs + Mis. AND + Mis. BE	21	21	21	70
13	Swi. VOTINGs + Mis. TL OR + Mis. BE	26	26	26	49
14	Joi. OR gates	11	11	11	11
15	Joi. OR gates + 1st VOTING as statics	6	6	6	9
16	Joi. OR gates+ 1st 2nd VOTINGs as statics	9	9	9	12
17	Joi. OR gates + all VOTINGs as statics	20	20	20	20

Table 7.18: PM Example 3: Static complex

	Error Type	O	I	F	M
1	Identical solution	0	0	0	0
2	Add Intermediate events	6	0	0	0
3	FDEP different ordering	0	0	0	0
4	FDEPs integrated	30	30	0	0
5	FDEPs integrated, dist. law applied	29	29	102	0
6	Duplicate use of triggered BEs	0	0	0	0
7	Switched PAND	18	18	18	33
8	Swi. PAND; Duplicate use of triggered BEs	18	18	18	33
9	Missing triggered BE	3	3	77	44
10	Switched AND	9	9	13	39
11	Switched BE	2	2	4	8
12	Switched BE all	6	6	8	8
13	Switched TL OR	2	2	2	2
14	Additional VOTING gates and BEs	25	25	25	25
15	Additional SPARE with new BEs	12	12	12	39
16	Additional SPARE with existing BE	12	12	6	33
17	Add. SPARE with existing BE + Swi. ANDs	21	21	18	38

Table 7.19: PM Example 5: Dynamic lecture exercise

all the necessary information to remove the mistakes. For test case 7, the matching needs to be parsed more carefully. While it detects an unmatched gate, the algorithm matches the actual missing node OR to the XOR gate and then claims that the XOR is unmatched. However, due to the greedy nature of this algorithm, such behaviour should be expected, even if it is unwelcome. Then, there are also test cases where the matching is too unspecific to give further details on how to fix the error, for example for case 9. While the score for test cases 10 to 18 seems reasonable, these are further examples where the matching does not provide useful information. This generally seems to occur as soon as two or more gate types have been used incorrectly.

Example 3: Static more compelx. The results of the third static test can be viewed in Table 7.18. Similar to the previous examples, equivalence could be detected, and for single errors or missing subtrees, useful information from the mismatch can be drawn. However, as soon as multiple gate nodes are affected by the changes, the matching usually does not provide useful information about how the fault tree should be modified. We can also observe, that the difference score alone cannot be used to provide a fair ranking. Similar to the other approaches, we can observe in test cases 13 to 15, that the score can decrease even if the same error occurs.

Example 4: Dynamic all gate types. This example focuses on validating the equivalence and interaction of all dynamic gate types. The results of the comparison can be seen in Table 7.20. Generally, equivalence between the solutions is detected correctly. Since most of the test cases also focus on one single error in the fault tree, the matching of the nodes provides sufficient feedback to easily correct the mistake. However, for those test cases that combine multiple errors, specifically cases 16 to 18, the feedback becomes less informative.

Example 5: Dynamic student exercise. The last example can be viewed in Table 7.19. This example supports the previously found findings: All equivalent solutions can be detected and for most singular errors, usable feedback gets generated. For test cases, where complete subtrees are added, like test case 14, the output is useful by indicating that those gates do not occur in the correct solution. However, for test cases with multiple errors, the feedback is more confusing than helping

	Error Type	O	I	F	M
1	Identical solution	0	0	0	0
2	No intermediate events (IE)	5	0	0	0
3	No toplevel intermediate event	1	0	0	0
4	Switched BE	2	2	2	2
5	Switched BE	2	2	2	2
6	Switched FDEP Trigger	2	2	8	8
7	Switched BE + Switched FDEP Trigger	4	4	10	10
8	Switched PAND	3	3	3	3
9	Switched SPARE gate	9	4	4	4
10	Switched TL AND	10	10	10	27
11	Switched TL AND	10	10	10	10
12	Higher threshold FDEP	1	1	16	16
13	Higher threshold SPARE	1	1	1	1
14	Missing AND	16	16	16	16
15	Missing PAND	13	13	13	13
16	Missing AND + Missing BE	13	12	16	16
17	Missing TL AND + Missing Subtree	32	32	37	37
18	Missing AND + Higher threshold SPARE	16	16	16	16
19	FDEP integrated + No intermediate events	18	11	0	0
20	FDEP integrated+ Additional triggered BE + No intermediates	19	15	2	2
21	FDEP integrated + Duplicate use of trigger BE + No intermediates	18	11	0	0
22	Duplicate use of triggered BE + No IE	5	0	0	0
23	Duplicate use of triggered BE + Additional triggered event + No IE	6	1	1	1

TABLE 7.20: PM Example 4: Dynamic road trip

7.3.3 Node matching feedback

The textual feedback we can gather from the pattern matching is a list of all node matches that are not completely identical and a list of all unmatched nodes of either the student's solution or the teacher's solution. As described in the evaluation, this output can easily get overwhelming, due to the greedy nature of this algorithm.
As long as we encounter single errors in fault trees, the feedback provides clear and useful information to correct the mistakes. However, as soon as we encounter multiple error types at once, for example, if multiple gate types have been switched around, the feedback becomes cumbersome to decipher.

7.3.4 Summary

Compared to the other two approaches, the pattern-matching comparison yields disappointing results. The evaluation started with good results in the single gates test, indicating that the general behaviour of the different gates was generally detected. Additionally, the textual feedback provided clear instructions on how to modify the fault tree. During the extended test set, however, we observed that the feedback was overwhelming in many cases, and it was cumbersome to extract all necessary instructions on how to modify the fault tree.
Regarding the identification of equivalence, the pattern-matching comparison detects many, but not all. Similar to the edit distance metric, several equivalences are only detected after some fault tree transformation steps. It also did not identify any inequivalence as equivalent.
Lastly, the difference score has been calculated based on a similarity metric between two TreeNodes. Since this metric contains many different aspects of the TreeNodes, like all its children and its childInfo individually, it is challenging to trace it back to each specific component. This also implies that a single error can be counted twice. For instance, if a gate is switched, the error will be detected both when comparing the node itself and when comparing its parent.

7.4 Discussion

All three test cases produced satisfactory results in terms of equivalence. However, the boolean equation algorithm occasionally detected equivalence where there should be none. For example, it identified equivalence when an FDEP gate included an additional, previously non-existent basic event as a triggered event. On the other hand, the other two comparison techniques missed some equivalences. For instance, if the logic of a VOTING gate is modelled using equivalent AND and OR gates, these comparisons would not recognize it. We anticipate that there are other equivalencies we did not cover with our test set that the edit distance and the pattern matching do not detect, but the boolean equation comparison does. Therefore, we conclude that the boolean equation comparison is the best approach to finding equivalences among two fault trees.
Next, we evaluated the difference scores from all three comparisons. For each algorithm individually, we concluded that the score alone is insufficient feedback, as it can vary significantly for the same error type. Therefore, we considered additional textual feedback. The boolean equation comparison includes post-process results indicating whether SPARE and PAND gates are used similarly across both fault trees, which is already factored into the equivalence assessment. For edit distance and pattern matching, the textual outputs describe how the fault tree needs to be transformed to match the correct solution. For small examples, both comparisons yield clear and understandable instructions on how to modify the fault tree. However, we also encountered limitations, where the feedback of both comparisons was hard to decipher. With pattern matching, this can already occur when two gates have a different type than the solution. With the edit distance, this usually occurs as soon as the transformation script contains more than 5 steps and includes a mix of all three operations. Therefore, this generally transpires earlier for the pattern matching than it does for the edit distance metric.
Due to the greedy nature of the pattern-matching comparison, we also cannot guarantee that the feedback steps contain a minimal version of the necessary improvements. On the other hand, the updated script for the edit distance metric is also not necessarily minimal, due to the reordering problem.
Overall, none of the algorithms seem sufficient enough to solve the use case of comparing two fault trees on its own. While the boolean equation comparison yields the most promising results in detecting equivalences, it cannot detect if a fault tree is minimal or not. It also provides no additional feedback on how to create the correct solution apart from the PAND and SPARE comparison. On the other hand, the edit distance metric and the pattern matching might not detect all equivalences, but they do provide guides on how to correct and improve the submission. Therefore, we propose a dual approach. Since the edit distance metric yields more promising feedback results, we propose to combine the boolean equation comparison with the edit distance metrics.

7.5 Final feedback

Once a student submits their solution for evaluation, the comparison algorithm will compare their solution to the correct one handed in by the teacher. Then, the student receives feedback that looks as follows: First, based on the boolean equation comparison, the student gets notified if their solution is logically equivalent to the teacher's solution or not. The boolean equation will only return a logic equivalence result if the difference score between the solution is 0 and if the post process did not detect any differences regarding the SPARE and the PAND gates.

Then, the students will also get an output based on the transformation script of the edit distance metric. First, the output will indicate if the solution is logically correct or not. If it is logically correct, the output will also mention improvements the student could make, e.g. by adding intermediate events. If the solution is incorrect, the student will be notified. Upon further request, they can additionally get information about which elements are incorrect or which elements are missing. If they still need help, they can request specific information about how each element needs to be modified, including the deletion and insertion of elements.

Examples

To show what kind of feedback we can expect, the output students will get for varying fault tree models will be shown.
First, if a student submitted a fault tree identical to the correct solution, they can expect the positive feedback given in Listing 7.1.

```
Well done, your solution is correct.
```

Listing 7.1: Feedback identical solution

If the student submitted a correct version of the fault tree but omitted the intermediate events, they will receive feedback as shown in Listing 7.2.

```
Well done, your solution is correct.
However, you could improve your fault tree by adding intermediate
    events.
```

Listing 7.2: Feedback identical solution - missing intermediate events

When the submitted solution is correct, but is constructed more complex than the teachers version, the student can expect feedback shown in Listing 7.3.

```
Well done, your solution is correct.
However, your fault tree is not minimal und could therefore be
   improved.
```

Listing 7.3: Feedback identical solution - not minimal

Lastly, if the student submits a version that has any kind of error, they will receive feedback that indicates what part of their fault tree they need to fix. In the example shown in Listing 7.4, the student mixed up a SPARE gate with a VOTING gate.

```
Your solution is not completely correct yet, try again!
Update the VOTING with a SPARE.
```

Listing 7.4: Feedback incorrect solution

Chapter 8

Conclusions

In this thesis, we have conducted a comprehensive comparison of three distinct algorithmic approaches to analyze the similarities and differences between two fault trees: Boolean equation comparison, edit distance metric comparison, and pattern matching comparison. Each algorithm has been thoroughly introduced and its implementation details have been described. Through testing on a diverse set of fault tree scenarios, the performance and efficacy of these algorithms have been assessed.
Our evaluation reveals that achieving a uniformly fair numerical scoring system across one or all algorithms is challenging since we usually observe a difference in the total score for errors of the same category, e.g. the category "Switched gate type". Or we observe, that scenarios we would classify as having bigger mistakes than others occasionally get a lower score.
However, a combination of these approaches allows us to provide valuable feedback to students regarding the correctness of their fault trees. Specifically, the boolean equation comparison proves effective in determining the equivalence of fault trees, while the edit distance metric comparison offers constructive feedback on necessary improvements and corrections, such as minimizing the fault tree structure.
Therefore, our final proposal is a comparison algorithm integrating both the boolean equation comparison and edit distance metrics comparison. This dual approach allows the student to verify the correctness of their solutions and also assists in refining their skills by highlighting areas for improvement.

Future Work

So far, the final approach has been evaluated exclusively with predefined test sets. As a next step, we could conduct a case study where students receive feedback on their fault trees and assess the perceived usefulness of this feedback. Based on this assessment, the feedback we obtain from the edit distance metric can be modified. For example, we can experiment if it is useful to start with only giving partial feedback.
Additionally, the current solution is implemented as a console application, requiring both the student's and the teacher's fault trees as inputs. Once the concept is validated, the application could be extended into a web service or integrated into a learning platform, allowing teachers to upload their solutions. Then, the student only needs to submit their solution for the correct exercise type and will receive feedback automatically.

Bibliography

[1] Ald fault tree analysis (fta) software. https://aldservice.com/Reliability-Products/fta.html. [Accessed 18-Nov-2023].

[2] Aptedsharp repository. [last accessed: 14 June 2024].

[3] Boolean expression parser - github. https://github.com/tomc128/boolean-expression-parser. [last accessed: 12 June 2024].

[4] Cafta technology package. https://polestartechnicalservices.com/cafta-software/. [Accessed 20-Nov-2023].

[5] Free web fault tree analysis (fta) software tool. https://www.fault-tree-analysis-software.com/fault-tree-analyser. [Accessed 18-Nov-2023].

[6] Icomparer interface. https://learn.microsoft.com/en-us/dotnet/api/system.collections.icomparer?view=net-8.0. [last accessed: 14 June 2024].

[7] Isograph fault tree analysis. https://www.isograph.com/software/reliability-workbench/fault-tree-analysis-software/. [Accessed 15-Nov-2023].

[8] Item toolkit: Fully integrated reliability analysis and safety software. https://www.itemsoft.com/item_toolkit.html. [Accessed 20-Nov-2023].

[9] Reliasoft (reliability analysis and management). https://www.hbkworld.com/en/products/software/analysis-simulation/reliability. [Accessed 18-Nov-2023].

[10] Software technology master program. https://www.utwente.nl/en/eemcs/fmt/education/st/. [Accessed 30-Oct-2023].

[11] Software testing and risk assessment – course description. https://osiris.utwente.nl/student/OnderwijsCatalogusSelect.do?selectie=cursus&cursus=202001472&collegejaar=2022&taal=en. [Accessed 30-Oct-2023].

[12] Rym Aiouni, Anis Bey, and Tahar Bensebaa. An automated assessment tool of flowchart programs in introductory programming course using graph matching. *Je-LKS : Journal of e-Learning and Knowledge Society*, 12:141–150, 01 2016. doi:10.20368/1971-8829/1138.

[13] Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978. doi:10.1109/TC.1978.1675141.

[14] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, and Dileep Kini. Automated grading of dfa constructions. In *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1976–1982, August 2013. URL: https://www.microsoft.com/en-us/research/publication/automated-grading-dfa-constructions/.

[15] Jayanshi Tripathi Andrew Heath, Marten Voorberg and Platon Frolov. Utml design project, 2021.

[16] JD Andrews. To not or not to not. In *Proceedings of the 18th international system safety conference*, volume 267275, 2000.

[17] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1):217–239, 2005. URL: https://www.sciencedirect.com/science/article/pii/S0304397505000174, doi:10.1016/j.tcs.2004.12.030.

[18] Yves Crama and Peter L Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

[19] Y. Dutuit and A. Rauzy. A linear-time algorithm to find modules of fault trees. *IEEE Transactions on Reliability*, 45(3):422–425, 1996. doi:10.1109/24.537011.

[20] R. Gulati and J.B. Dugan. A modular approach for analyzing static and dynamic fault trees. In *Annual Reliability and Maintainability Symposium*, pages 57–63, 1997. doi:10.1109/RAMS.1997.571665.

[21] Pablo Gómez-Abajo, Esther Guerra, and Juan Lara. Automated generation and correction of diagram-based exercises for moodle. *Computer Applications in Engineering Education*, 31, 08 2023. doi:10.1002/cae.22676.

[22] Colin A. Higgins and Brett Bligh. Formative computer based assessment in diagram based domains. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, page 98–102, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1140124.1140152.

[23] Christoph M Hoffmann and Michael J O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[24] Gil Hoggarth and Mike Lockyer. An automated student diagram assessment system. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education*, ITiCSE '98, page 122–124, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/282991.283089.

[25] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, Arend Rensink, and Mariëlle Stoelinga. Fault trees on a diet: automated reduction by graph rewriting. *Formal aspects of computing*, 29:651–703, 2017.

[26] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, and Mariëlle Stoelinga. Uncovering dynamic fault trees. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 299–310, 2016. doi:10.1109/DSN.2016.35.

[27] Dong Liu, Chunyuan Zhang, Weiyan Xing, Rui Li, and Haiyan Li. Quantification of cut sequence set for fault tree analysis. In Ronald Perrott, Barbara M. Chapman, Jaspal Subhlok, Rodrigo Fernandes de Mello, and Laurence T. Yang, editors, *High Performance Computing and Communications*, pages 755–765, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[28] Jun Ni, Wencheng Tang, and Yan Xing. A simple algebra for fault tree analysis of static and dynamic systems. *Reliability, IEEE Transactions on*, 62:846–861, 12 2013. doi:10.1109/TR.2013.2285035.

[29] David Nicol and Debra Macfarlane. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in Higher Education*, 31:199–218, 05 2006. doi:10.1080/03075070600572090.

[30] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):1–40, 2015.

[31] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0306437915001611, doi:10.1016/j.is.2015.08.004.

[32] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.3190010410, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jgt.3190010410, doi:10.1002/jgt.3190010410.

[33] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15-16:29–62, 2015. URL: https://www.sciencedirect.com/science/article/pii/S1574013715000027, doi:10.1016/j.cosrev.2015.03.001.

[34] J. Soler, I. Boada, F. Prados, J. Poch, and R. Fabregat. A web-based e-learning tool for uml class diagrams. In *IEEE EDUCON 2010 Conference*, pages 973–979, 2010. doi:10.1109/EDUCON.2010.5492473.

[35] Pete Thomas, Kevin Waugh, and Neil Smith. Experiments in the automatic marking of er-diagrams. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 37, 09 2005. doi:10.1145/1151954.1067490.

[36] M. G. Thomason and E. W. Page. Boolean difference techniques in fault tree analysis. *International Journal of Computer & Information Sciences*, 5(1):81–88, Mar 1976. doi:10.1007/BF00991073.

[37] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, jan 1976. doi:10.1145/321921.321925.

[38] William E Vesely, Francine F Goldberg, Norman H Roberts, David F Haasl, et al. *Fault tree handbook*. Systems and Reliability Research, Office of Nuclear Regulatory Research, US . . . , 1981.

[39] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, jan 1974. doi:10.1145/321796.321811.

[40] Hugh A Watson et al. Launch control safety study. *Bell labs*, 1961.

[41] Liudong Xing and Suprasad V. Amari. *Fault Tree Analysis*, pages 595–620. Springer London, London, 2008. doi:10.1007/978-1-84800-131-2_38.

[42] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989. doi:10.1137/0218082.