Converting Decision Trees into Fault Trees

DAN NEGRU, University of Twente, The Netherlands
SUPERVISOR: MILAN LOPUHAÄ-ZWAKENBERG, University of Twente, The Netherlands

ABSTRACT

Decision trees and fault trees serve as foundational models for assessing the reliability and performance of complex systems, albeit originating from distinct domains. While their underlying information is fundamentally equivalent, their different structures serve different analytical perspectives, leading to distinct applications in various fields. Despite extensive research into converting fault trees to decision trees, the reverse process remains largely unexplored. This paper seeks to bridge the gap between these two essential models. We propose novel algorithms based on cut sets and recursion to facilitate this conversion process, highlighting the effectiveness of the latter in our experimental results. The focus of this research extends beyond mere translation, aiming to optimize the visual representation of the generated fault trees and to acknowledge the inherent challenges in transitioning from a decision-based framework to a fault-based representation.

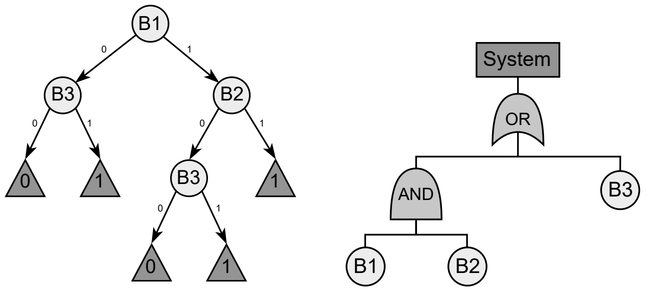
Keywords: decision tree, fault tree, conversion algorithm, reliability engineering, decision making.

1 INTRODUCTION

Complex systems often require models to assess their reliability and make informed decisions regarding their design, maintenance, and operation. Decision trees (DTs) and fault trees (FTs) are two popular graph models used for this purpose. They provide graphic representations of a hierarchical data structure and are widely used in different domains and applications such as reliability engineering, system analysis, and computer memory optimization [4].

Decision trees, as illustrated in Figure 1a, are flowchart-like structures that model decisions and their possible consequences [4]. A decision tree consists of nodes that form a rooted tree, meaning it is a directed tree with a node called "root" that has no incoming edges. All other nodes have exactly one incoming edge [8]. A node with outgoing edges is called a decision node and it contains a binary variable. All other nodes are called leaf nodes, representing the end point of a decision path. DTs can cope with numerical and nominal attributes. They have an intuitive interpretation and are applied in text classification, diagnosis of diseases, fraud detection, speech recognition, video analysis, among others [4].

Fault trees, as illustrated in Figure 1b, are graphical methods that model how failures propagate through the system, i.e., how component failures lead to system failures [9]. FTs can be described as a graphical probabilistic risk assessment technique whereby an undesirable event (called the top event) is postulated and the possible ways for this top event to occur are systematically deduced for combinations of initiating and intermediate events. The events are generally binary, i.e., they either occur or not. System components are either in parallel or in series so combinations of events that lead to failure are identified with logic gates. The two basic gate symbols used for fault tree construction are the AND gate and the OR gate. The AND gate demonstrates that the higher-level event (the gate's output) will occur if and only all immediate lower-level events (the gate's inputs) occur. Similarly, OR gates demonstrate that only one of the gate's inputs must occur for the gate's output to occur. Although a gate can have many inputs, it only has one output [3]. Sometimes, a third gate can be used: a k/N gate (or VOT gate), which indicates that the associated event will occur if minimally k of the N gate's inputs will occur [4]. Additionally, it is important to mention that the NOT gate is not being used in the construction of FTs, as this can result in a non-coherent structure, meaning that components' working states contribute to the system failure [1].



(a) Decision Tree (b) Fault Tree

Fig. 1. Example of equivalent trees representing the Boolean expression $(B1 \wedge B2) \vee B3$.

Conversion methods are mathematical transformations that convert one formalism into another, while preserving relevant properties. Although both FTs and DTs decode Boolean functions, they do so in different ways: FTs typically use logical gates and probability distributions, reflecting the hierarchical relationships and dependencies among events leading to system failures, while DTs partition the input space based on attribute values, aiming to classify or predict outcomes through a series of binary decisions. An example of equivalent trees is depicted in Figure 1, showcasing the potential for interchangeability between decision trees and fault trees.

In the transformation FT $\longrightarrow$ DT, the basic events in the FT become decision nodes in the DT, and the top events in the FT become the leaf nodes in the DT. A few methods have been developed to accomplish this task [2, 10] and they will be further discussed in the

TScIT 41, July 5, 2024, Enschede, The Netherlands

related work section. On the other side, in the DT → FT conversion, the decision nodes in the DT become basic events in the FT, and the leaf nodes in the DT become the top events in the FT [4].

While transformations from DTs to FTs are not inherently difficult to achieve, the challenge lies in crafting a transformation that is deemed "good". In this context, a "good" transformation refers to one that minimizes the number of intermediate gates in the resulting FT. This criterion is crucial as it directly impacts the clarity and efficiency of the FT model derived from the DT. By reducing unnecessary complexity and ensuring a concise structure, such transformations not only enhance our theoretical understanding of the resulting models, but also improve the practical applications in system reliability analysis and decision-making. In the following sections, we aim to delve deeper into identifying methodologies that fulfill these criteria effectively, thereby advancing our capability to develop safer and more dependable systems across diverse domains.

This research focused on developing algorithms to convert DTs into FTs. Additionally, some optimizations were introduced to simplify the resulting models, making them more concise and comprehensible. Lastly, experiments were conducted to assess the effectiveness of the respective algorithms.

In this paper, we will start by addressing the related work for the conversion of FTs into DTs in section 2, and then we will discuss the proposed research question in section 3. Next, in section 4, we will describe the methodology used to develop the algorithms for transforming DTs into FTs and, in section 5, we will cover the experiments that we conducted to assess their effectiveness. Finally, section 6 will highlight the potential areas of focus for future work, and section 7 will provide the conclusion for this research paper.

2 RELATED WORK

In this section we will go over some of the related work about the conversion algorithms of fault trees into decision trees, and about the inference of fault tree models.

In 2004, Assaf and Dugan [2] described a methodology for designing a diagnostic decision tree from a dynamic fault tree, which makes use of Markov chains. The Vesely-Fussell measure of importance is used as the corner stone of the methodology, because it provides an accurate measure of components' relevance from a diagnosis perspective. The outcome represents a diagnostic decision tree generated for a real dynamic system, and it can be used by repair and maintenance crew to diagnose a system without having previous knowledge or experience about the diagnosed system.

In their 2009 paper, Tao et al. [10] presented a fault tree analysis method to generate diagnostic decision trees. All minimal cut sets, their occurrence probabilities and components' diagnosis importance factors are determined via fault tree analysis used for system reliability. Minimal cut sets represent minimal sets of component failures that cause a system failure. Using the diagnostic sequence of system components, a diagnostic decision tree can be generated.

The paper by Jimenez-Roa et al. [4] compared the similarities and differences between the three prominent graph models commonly used in reliability engineering: fault trees (FTs), decision trees (DTs), and binary decision diagrams (BDDs). The comparison focused on their purpose, application, structural representation, analysis methods, construction, benefits and limitations. The results showed that, given that FTs, DTs and BDDs have different purposes and application domains, they adopt different structural representations and analysis methodologies that entail a variety of benefits and limitations. Addressing the latter can be achieved by employing conversion methods or extensions.

In their 2023 paper, Jimenez-Roa et al. [5] introduced the FT-MOEA algorithm, which is based on multi-objective evolutionary algorithms. Its goal is to infer efficient FT structures that achieve a complete representation of the failure mechanisms contained in a given failure data set without human intervention. The algorithm enables the simultaneous optimization of different relevant metrics such as the FT size, the error computed based on the failure data set and the Minimal Cut Sets.

A different paper by Jimenez-Roa et al. [6] presented SymLearn, a method to automatically infer FT models from data. It takes as input the failure data of the system components and exploits evolutionary algorithms to learn a compact FT matching the input data. SymLearn uses the symmetries in the failure data set to learn the symmetric FT parts only once, and partitions the input data into independent modules, subdividing the inference problem into smaller parts.

3 RESEARCH QUESTION

How can decision trees be effectively converted into fault trees?

This main research question can be answered with the following two sub-questions:

(1) What are the key challenges in converting decision trees into fault trees?
(2) What techniques can be employed to generate compact fault trees while preserving essential information?

4 METHODOLOGY

In this section, we outline the methodologies employed to convert decision trees into fault trees. Our approach involves two main algorithms: one based on cut sets and the other based on recursion. In the following figures from this section, a system failure in the decision tree is represented by a leaf node with value 1, which is equivalent to the value True, indicating that the system fails. Conversely, a leaf node with value 0 (False) means that the system does not fail. Additionally, for each node in the decision tree, we have two options depicted by its outgoing edges: 1 or 0, depending on whether the event represented by the given node fails or not.

4.1 Algorithm Using Cut Sets

Cut sets are combinations of component failures that lead to a system failure. In a decision tree, cut sets can be defined as sets of decision nodes with value 1 that lie on a path to a leaf node with value 1. Minimal cut sets are special cut sets where removing any component from the set would prevent it from causing the system failure, meaning that it would no longer be a cut set. They are particularly significant because they point to system vulnerabilities, highlighting the most critical points where failure can occur with the least number of component failures.

To obtain cut sets, we derive them directly from the decision tree. For example, consider the decision tree illustrated in Figure 2a. The tree has three decision nodes: B1, B2, and B3. The paths from the root to the leaf nodes with value 1 are {B1} and {B2, B3}. Both of them represent minimal cut sets. To offer an example of a cut set that is not minimal, consider the set {B1, B2, B3}. We can remove B1 from the set and we will be left with {B2, B3}, or we can remove B2 and B3 from the set and we will be left with {B1}. In both cases, the result is still a cut set.

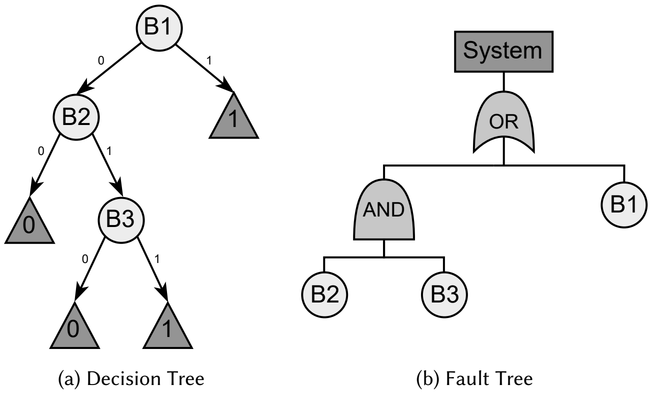


Fig. 2. Example using the Cut Sets Algorithm.

Each cut set consists of decision nodes that we can translate into basic events for the fault tree. Then, we can combine the basic events of each cut set using AND gates, signifying that all of them must occur for the system to fail. These AND gates are then consolidated into a single main OR gate, which connects to the system. This process essentially uses the disjunctive normal form to create the fault tree, and it can be described as an OR of ANDs, illustrating different combinations of events that can lead to a system failure.

Coming back to the example from Figure 2, when constructing the fault tree, as shown in Figure 2b, the cut set {B1} directly leads to a system failure, so we connect it to the main OR gate, avoiding an intermediary AND gate with a single event. On the other hand, the cut set {B2, B3} requires both B2 and B3 to fail. These events are connected to an intermediary AND gate, which is then linked to the main OR gate. Finally, the main OR gate represents the overall system failure conditions.

There are challenges and considerations to keep in mind when using this method. The process may produce sub-optimal fault tree structures, for instance due to overlapping paths that occur when multiple cut sets share common elements, leading to redundancy in the fault tree. An example of this is illustrated in Figure 3b: the basic event B1 is repeated two times. Additionally, consider if we would add another decision node – B4 in the decision tree and the Boolean expression would change to $B1 \wedge (B2 \vee B3 \vee B4)$. In this case, the resulting fault tree would get another intermediate AND gate between B1 and B4. Therefore, post-processing steps, such as simplification and elimination of redundant nodes, may be required to ensure the resulting fault tree is as concise as possible.

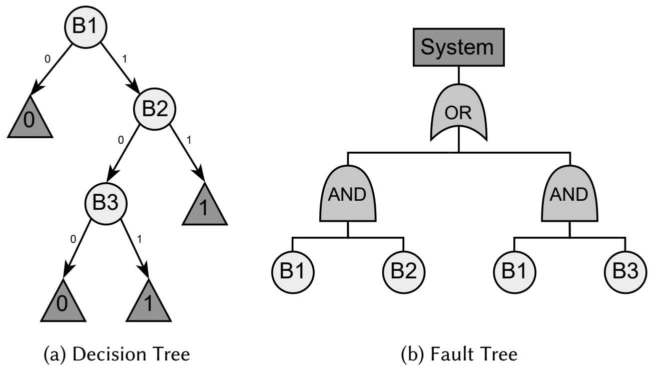


Fig. 3. Sub-optimal Fault Tree resulting from the Cut Sets Algorithm. The initial Decision Tree represents the Boolean expression $B1 \wedge (B2 \vee B3)$.

In addition, the resulting fault tree will typically be very wide, but not so tall. This may obscure the hierarchical relationships between system components and their failure modes. This hierarchical clarity is crucial for understanding how failures propagate through different levels of the system: from basic components to higher-level system failures. Without it, stakeholders may find it difficult to prioritize critical failure paths or identify sub-component failures. Furthermore, the complexity introduced by a wide fault tree can hinder efficient analysis and communication, potentially delaying decision-making processes and compromising the accuracy of system improvement strategies.

4.2 Recursive Algorithm

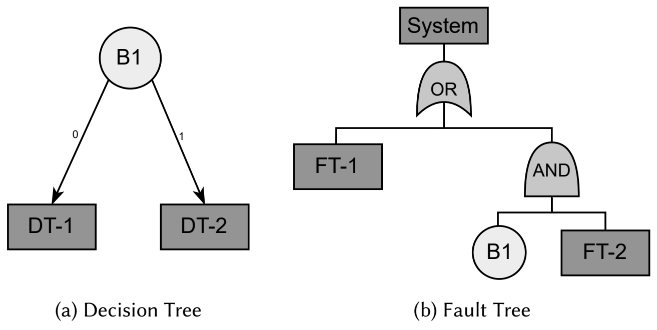
The recursive algorithm constructs a fault tree by recursively splitting the decision tree into sub-trees and combining the results. This method involves several steps and optimizations to enhance the compactness and efficiency of the resulting fault tree.

4.2.1 Basic Implementation.

The basic recursive approach operates in the following way. First, for each node in the decision tree, we split it into two sub-trees: the left sub-tree (where the node value is 0) and the right sub-tree (where the node value is 1). Next, we construct the fault tree by performing an OR operation between the fault tree represented by the left sub-tree and an intermediate event. The intermediate event is an AND operation between the current node and the fault tree represented by the right sub-tree. This method captures the logical dependencies and failure paths within the decision tree, translating them into the fault tree structure, as illustrated in Figure 4.

The decision tree in Figure 4a shows a simple example where node B1 is split into two sub-trees: DT-1 and DT-2. The corresponding fault tree structure in Figure 4b is constructed by performing an OR operation between FT-1 (the fault tree obtained from DT-1) and an intermediate event, which is an AND operation between the current node B1 and FT-2 (the fault tree obtained from DT-2).

To present an example with the intermediary and final results of this algorithm, consider again the decision tree from Figure 2a. In Figure 5, we can visualise the intermediate result of converting the



(a) Decision Tree (b) Fault Tree

Fig. 4. Recursive Algorithm Approach.

decision tree into a fault tree using the recursive approach. Note that some basic events are mirrored from the leaf nodes of the decision tree and have values of 0 and 1. These represent special cases that are handled differently in our algorithm. If DT-2 is a leaf node with value 1, this would translate into FT-2 also having value 1, and the intermediate AND gate would, for example, look like this: $(B1 \land 1)$. By applying the Boolean logic, we can replace this with just $B1$. In the same manner, if DT-1 is a leaf node with value 0, FT-1 would have value 0 as well, and the intermediate OR gate would, for example, look like this: $(B3 \lor 0)$. Again, by applying the Boolean logic, we can replace this with just $B3$. Finally, if we handle these special cases, the resulting fault tree obtains the same structure as in Figure 2b.

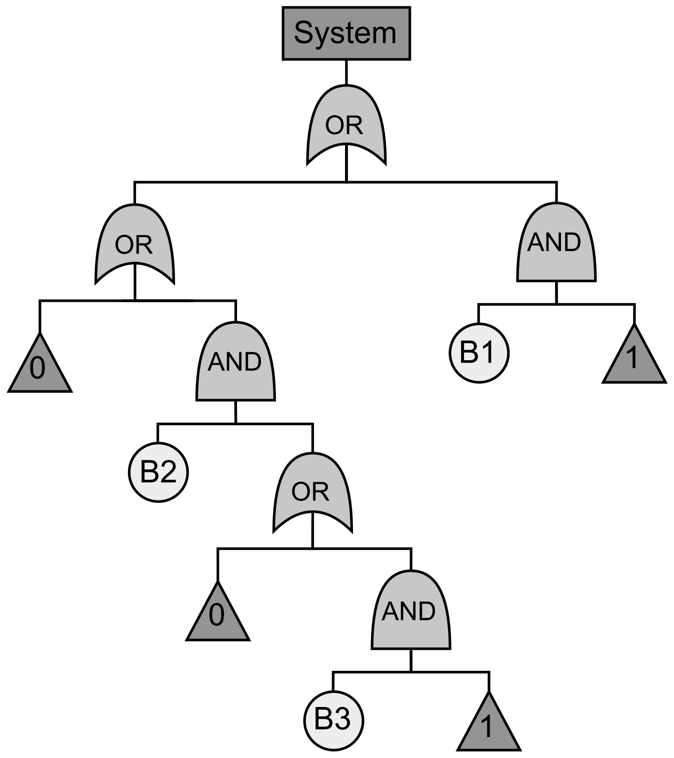
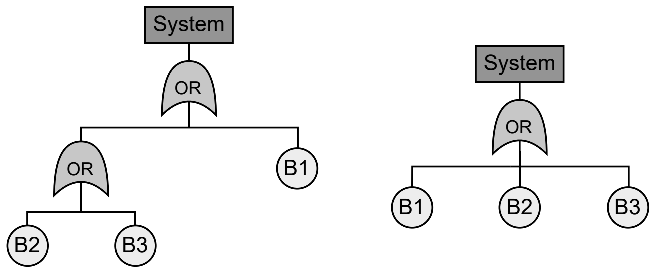


Fig. 5. Recursive Algorithm Intermediate Result.

4.2.2 *Optimization 1: Merging Intermediate Events.*

The fault trees resulting from our basic recursive algorithm generally have a lot of intermediate gates with only two children: a basic event and another intermediate gate of the same type (operand). An example of this can be found in Figure 6a. Thus, during the construction of the fault tree, we identify these intermediate events and we merge them, in order to reduce redundancy and enhance compactness. The result of the given example is shown in Figure 6b. It is also important to mention that this optimization applies not only to OR gates, as shown in the example, but also to AND gates.



(a) Initial Structure (b) Result After Merge

Fig. 6. Merging Intermediate Events.

This process is particularly useful in complex systems where multiple intermediate events can be combined to simplify the fault tree, thus making its analysis more efficient by reducing the number of gates and edges between them.

While this technique generally makes the fault tree more concise, there are cases where keeping the initial structure (Figure 6a) might be preferable. For instance, if the initial structure visually represents distinct sub-components of the system, maintaining this organization could aid in understanding the fault tree's hierarchical structure.

4.2.3 *Optimization 2: Removing Duplicate Intermediate Events.*

Decision trees can sometimes contain duplicated sub-trees in their structure. In these cases, the duplicates would also be reflected in the fault tree resulting from the converting algorithm. To prevent against this redundancy, this optimization checks if an intermediate event that has to be added to the fault tree already exists and it only adds it in case it does not. This ensures that each intermediate event in the fault tree is unique, thereby maintaining clarity and reducing complexity.

For example, consider the scenario in Figure 7a. The fault tree includes multiple instances of the same intermediate event: an OR gate between the two basic events B1 and B2. Instead of adding multiple identical gates for each occurrence, the algorithm uses the respective already existing gate, such as in the result from Figure 7b.

While this optimization generally reduces the size of the fault tree, it can introduce a drawback regarding its visual structure. Specifically, allowing an intermediate event to have multiple parents (or be referenced in multiple parts of the fault tree) can add a level of visual

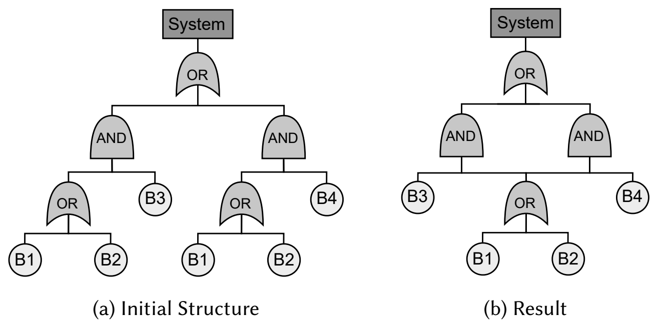


Fig. 7. Removing Duplicate Events.

complexity due to the increased number of edges per event. For small trees, since there are not a lot of connections between events, this complexity is typically outweighed by the benefits of avoiding duplicate events and maintaining a streamlined structure. However, for larger trees, the number of edges becomes substantial and in some cases this could make it hard for stakeholders to interpret and understand the given models.

Lastly, it is essential to note that implementing this optimization requires careful management of event creation and modification processes to maintain the integrity of the fault tree model.

4.2.4 *Post-Processing.*

Post-processing can be used to further simplify and refine the resulting models. As the name suggests, this technique is not a part of the algorithm, but it can be employed on the fault trees obtained from it. One method is to apply the Distributive Law from the Boolean logic to restructure the resulting fault tree. For instance, when multiple AND gates in the fault tree combine the same intermediate event and then these AND gates are linked via an OR gate, we can apply the law: $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$. Similarly, for OR gates: $(A \vee B) \wedge (A \vee C) = A \vee (B \wedge C)$. These simplifications reduce the number of gates and edges, making the fault tree more concise and comprehensible. It's important to mention that this is just one specific example of how Boolean logic laws can be applied. Various other methods and patterns could be implemented as a post-processing step, but their exploration will be left for future work.

5 EXPERIMENTS

In this section, we present the results of applying our conversion methodologies, evaluating their performance and efficiency based on the size and complexity of the resulting fault trees.

5.1 Approach

To assess the effectiveness of our conversion algorithms, we conducted a series of experiments. For these experiments, we needed a large set of different decision trees that would serve as our test suite. However, creating representative and valid decision trees in a random manner is not an easy task. For this reason, we decided to follow a different approach that creates the given decision trees from randomly created fault trees. So, first, we generated multiple random fault trees of varying sizes, ranging from 5 to 50 nodes. There has already been done some research on this topic [7], providing open-source code in Python with the necessary functions to generate fault trees, and we adapted these to be suitable for our own implementation. Then, we used these fault trees to obtain their corresponding decision trees, ensuring a diverse set of input data for our algorithms. To do this, we parsed each fault tree and constructed the Boolean formula it represented, which we then used to create the respective BDD using the dd package in Python. Next, we created the decision tree from the BDD, which is a trivial transformation. We then applied the cut sets algorithm and the three variations of the recursive algorithm (basic, with merging, and with duplicates removal) to convert the decision trees back into fault trees. It is important to mention that the conversion for a single decision tree is fast, ranging from 0.1 ms to a few milliseconds, depending on the size and complexity of the decision tree and on the used optimization. Finally, we compared the initial fault trees with the ones resulting from our algorithms. For this analysis, we used two metrics: the number of nodes, which measures the total number of basic and intermediate events in the fault tree, and the number of edges, which measures the total number of connections between the nodes in the fault tree. These metrics were chosen to reflect the size and complexity of the fault trees, which are critical factors for practical applications.

To ensure the robustness of our results, each experiment was repeated 5000 times. This extensive repetition allowed us to account for variability in the randomly generated fault trees. Specifically, for each size category, we generated 5000 random fault trees. For the results (number of nodes and edges in the converted fault trees), we computed the average across these 5000 repetitions. It is important to note that due to this experimental approach, some fault trees may have been generated multiple times, particularly for smaller sizes where the number of distinct fault trees is limited. This occurrence, however, does not undermine the reliability of our findings. By repeating each experiment 5000 times and averaging the results, we ensured robust statistical representation across varying tree sizes. This approach effectively mitigates any potential biases that could arise from duplicated fault trees. Moreover, encountering similar fault scenarios multiple times is reflective of real-world scenarios, where certain structures might be more common.

5.2 Results

The results of our experiments are summarized in Figures 8 and 9. They illustrate the relationship between the size of the initial fault trees and the size of the converted fault trees, measured in terms of nodes and edges, respectively.

In Figure 8, we observe that the basic recursive algorithm results in a significantly larger fault tree compared to the other methods. This increase in size is due to the lack of optimization in the basic approach, leading to redundant nodes. The cut sets algorithm and the recursive algorithm with merging show similar performance, with a moderate increase in the number of nodes as the size of the initial fault tree increases. The recursive algorithm with duplicates removal consistently produces the smallest fault tree, highlighting the effectiveness of this optimization in reducing redundancy.

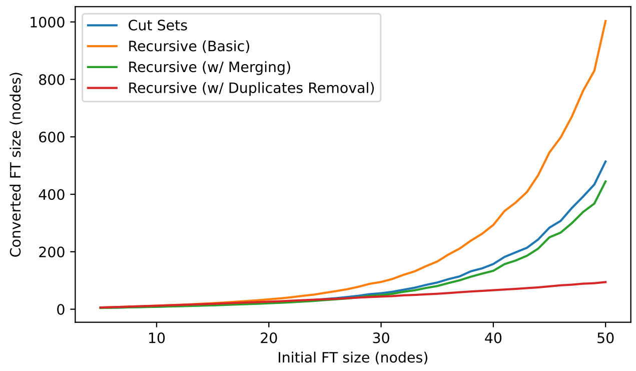


Fig. 8. Results based on the number of nodes.

Table 1 shows the detailed results based on the number of nodes for the Recursive (w/ Duplicates Removal) algorithm, providing a clear understanding of its performance. This is necessary, since Figure 8 makes it clear that this algorithm outperforms the others, however it is hard to evaluate how well it does against the baseline $y = x$. The data reveals that for smaller trees, the converted model's size based on nodes is very similar to the one of the initial structure, while for larger trees it tends to increase at a higher rate, nearly doubling for trees with 50 nodes. Still, this is much better than the other algorithms, with, for example, the Basic Recursive version almost reaching 1000 nodes for the same initial size of 50 nodes.

Table 1. Detailed results based on the number of nodes for the Recursive (w/ Duplicates Removal) Algorithm from Figure 8.

Initial FT size (x)	10	20	30	40	50
Converted FT size (y)	12.1	26.2	44.1	66.2	94.4

Figure 9 presents the results in terms of the number of edges. Here, the cut sets algorithm results in fault trees with the highest number of edges, reflecting its exhaustive enumeration of failure paths. The recursive algorithm with merging and duplicates removal again show improved performance compared to the basic approach, with the duplicates removal optimization yielding the most compact fault trees.

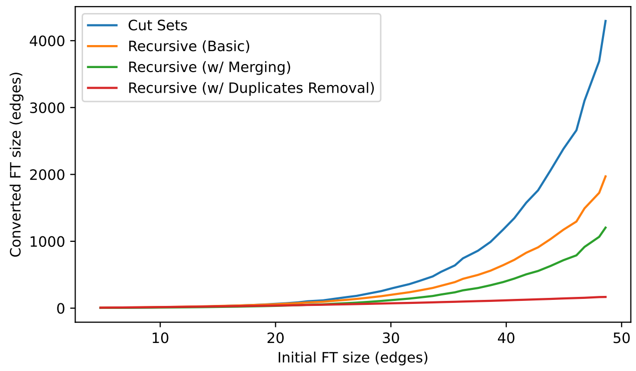


Fig. 9. Results based on the number of edges.

Table 2 shows the detailed results based on the number of edges for the Recursive (w/ Duplicates Removal) algorithm, offering a clear insight into its performance. The motivation behind the need for this table is the same as the one for the number of nodes. It is also important to note that, unlike for nodes, fault trees cannot be generated for a specific number of edges directly. Instead, we generate fault trees based on a set number of nodes and then calculate their average number of edges. This is why the initial values in the table are not integers – they represent these averages. The data shows that for smaller trees, the converted model's size based on edges is very similar to the one of the initial structure, while for larger trees it tends to increase at a higher rate, more than tripling for trees with an average of 48.6 edges. Still, this is better than the other algorithms, with, for example, the Cut Sets version surpassing 4000 edges for the same initial average size of 48.6 edges.

Table 2. Detailed results based on the number of edges for the Recursive (w/ Duplicates Removal) Algorithm from Figure 9.

Initial FT size (x)	10.3	20.9	31.6	41.7	48.6
Converted FT size (y)	16.9	42.0	78.4	126.8	167.8

These results demonstrate the importance of optimization techniques in the recursive algorithm, which significantly improve the compactness of the resulting fault trees. Our findings suggest that while the cut sets algorithm could be useful for some applications, the optimized recursive algorithms offer a more efficient approach for converting decision trees into fault trees, creating better results according to our initial definition of a "good" transformation. Overall, our experiments validate the proposed methodologies and provide a comprehensive comparison of their performance.

6 FUTURE WORK

While our research has made significant progress in converting decision trees into fault trees and optimizing the resulting structures, there are several avenues for future work that can further enhance the effectiveness and applicability of our methodologies.

6.1 Advanced Optimization Techniques

Future research could explore more advanced optimization techniques to further minimize the size and complexity of the fault trees. Some of these techniques include: using the Quine-McCluskey or Espresso Algorithms to minimize the Boolean functions represented by the converted fault trees; applying genetic algorithms to evolve the fault tree structures and select the smaller and less complex models over successive generations; developing parallel algorithms for fault tree simplification that can leverage multi-core processors to handle large and complex fault trees more efficiently.

6.2 Integration with Machine Learning Models

Integrating our conversion techniques with machine learning models could allow us to automatically generate fault trees from data-driven decision models. Importantly, our conversion process from decision trees to fault trees is fast, addressing a significant bottleneck in existing fault tree learning techniques which often struggle with computation time constraints.

6.3 Post-Processing Enhancements

Further development of post-processing techniques, including the application of additional Boolean logic simplifications and the exploration of other logical laws, can improve the compactness and readability of fault trees. However, it is worth noting that there exists a trade-off between the size of the fault trees and the algorithm performance: more sophisticated post-processing methods may lead to an improved tree quality but they could also increase the computation time of the algorithm.

6.4 Tool Development

Creating user-friendly software tools that use our algorithms and allow stakeholders to visualise the conversion process would facilitate a broader adoption and practical use of our research.

7 CONCLUSION

In this research, we addressed the main question: *How can decision trees be effectively converted into fault trees?* We developed and evaluated several algorithms for this conversion, focusing on preserving essential information while generating compact and comprehensible fault tree models.

Our work began with the identification of the key challenges in converting decision trees into fault trees. Based on our initial insights, we proposed and implemented various techniques, such as the cut sets algorithm and the recursive algorithms with different optimizations (merging and duplicates removal), to tackle these challenges.

The experimental results demonstrated that while the basic recursive algorithm tends to produce larger fault trees, the optimized versions significantly improve compactness and efficiency. The recursive algorithm with duplicates removal, in particular, consistently yielded the most compact fault trees, highlighting the importance of this optimization.

Additionally, we briefly discussed post-processing techniques using the Boolean logic to further refine and simplify the fault trees. By applying laws such as the Distributive Law, we can reduce the number of gates and edges, making the fault trees easier to analyze.

It is also important to highlight the broader implications of our work within the context of existing data-driven techniques. The problem of generating decision trees from data is well-known. By effectively converting decision trees into fault trees, we establish a novel pathway to generate fault trees directly from data. This is achieved by first creating decision trees from data using established machine learning techniques and then applying our conversion methods. This new approach not only leverages the strengths of decision trees in initial data analysis but also enhances the subsequent fault tree generation, providing a powerful tool for predictive maintenance and risk assessment based on data-driven decision models.

In conclusion, our research provides a comprehensive methodology for converting decision trees into fault trees, offering various algorithms and optimization techniques to suit different needs. Future work will build on these foundations, exploring advanced optimizations, machine learning integrations, and post-processing enhancements to further advance the field of fault tree analysis.

REFERENCES

[1] J. D. Andrews. 2001. The use of not logic in fault tree analysis. *Quality and Reliability Engineering International* 17, 3 (May 2001), 143–150. https://doi.org/10.1002/qre.405

[2] T. Assaf and J.B. Dugan. 2004. Diagnostic expert systems from dynamic fault trees. In *Annual Symposium Reliability and Maintainability, 2004 - RAMS*. IEEE, Los Angeles, CA, USA, 444–450. https://doi.org/10.1109/RAMS.2004.1285489

[3] PhD Dillon-Merrill and L. Robin. 2008. Logic trees: Fault, success, attack, event, probability, and decision trees. *Wiley Handbook of Science and Technology for Homeland Security* (2008), 1–22. https://www.researchgate.net/profile/Gregory-Parnell/publication/229578763_Logic_Trees_Fault_Success_Attack_Event_Probability_and_Decision_Trees/links/54bd10690cf218da9390ef83/Logic-Trees-Fault-Success-Attack-Event-Probability-and-Decision-Trees Publisher: John Wiley & Sons, Inc. Hoboken, NJ, USA.

[4] L. A. Jimenez-Roa, T. Heskes, and M. Stoelinga. 2021. Fault Trees, Decision Trees, And Binary Decision Diagrams: A Systematic Comparison. In *Proceedings of the 31st European Safety and Reliability Conference (ESREL 2021)*. 673–680. https://doi.org/10.3850/978-981-18-2016-8_241-cd arXiv:2310.04448 [cs].

[5] Lisandro Arturo Jimenez-Roa, Tom Heskes, Tiedo Tinga, and Mariëlle Stoelinga. 2023. Automatic Inference of Fault Tree Models Via Multi-Objective Evolutionary Algorithms. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (July 2023), 3317–3327. https://doi.org/10.1109/TDSC.2022.3203805 Conference Name: IEEE Transactions on Dependable and Secure Computing.

[6] Lisandro Arturo Jimenez-Roa, Matthias Volk, and Mariëlle Stoelinga. 2022. Data-Driven Inference of Fault Tree Models Exploiting Symmetry and Modularization. In *Computer Safety, Reliability, and Security*, Mario Trapp, Francesca Saglietti, Marc Spisländer, and Friedemann Bitsch (Eds.). Springer International Publishing, Cham, 46–61. https://doi.org/10.1007/978-3-031-14835-4_4

[7] Meike Nauta, Doina Bucur, and Mariëlle Stoelinga. 2018. LIFT: Learning Fault Trees from Observational Data. In *Quantitative Evaluation of Systems*, Annabelle McIver and Andras Horvath (Eds.). Springer International Publishing, Cham, 306–322. https://doi.org/10.1007/978-3-319-99154-2_19

[8] Lior Rokach and Oded Maimon. 2005. Decision Trees. In *Data Mining and Knowledge Discovery Handbook*, Oded Maimon and Lior Rokach (Eds.). Springer US, Boston, MA, 165–192. https://doi.org/10.1007/0-387-25465-X_9

[9] Enno Ruijters and Mariëlle Stoelinga. 2015. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review* 15-16 (Feb. 2015), 29–62. https://doi.org/10.1016/j.cosrev.2015.03.001

[10] Yongjian Tao, Decun Dong, and Peng Ren. 2009. Notice of Violation of IEEE Publication Principles: Decision Trees Generation Based on Fault Trees Analysis. In *2009 International Forum on Information Technology and Applications*, Vol. 2. 178–180. https://doi.org/10.1109/IFITA.2009.192

APPENDIX

During the preparation of this work, the author used ChatGPT to correct grammatical errors, express his own ideas more coherently, and quickly adapt figures and tables to the LaTeX syntax. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the work.