Data-Driven Inference of Fault Tree Models Exploiting Symmetry and Modularization

Lisandro Arturo Jimenez-Roa[1(✉)], Matthias Volk[1], and Mariëlle Stoelinga[1,2]

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{l.jimenezroa,m.volk,m.i.a.stoelinga}@utwente.nl
[2] Department of Software Science, Radboud University, Nijmegen, The Netherlands

Abstract. We present *SymLearn*, a method to automatically infer fault tree (FT) models from data. *SymLearn* takes as input failure data of the system components and exploits evolutionary algorithms to learn a compact FT matching the input data. *SymLearn* achieves scalability by leveraging two common phenomena in FTs: (i) We automatically identify symmetries in the failure data set, learning symmetric FT parts only once. (ii) We partition the input data into independent modules, subdividing the inference problem into smaller parts.

We validate our approach via case studies, including several truss systems, which are symmetric structures commonly found in infrastructures, such as bridges. Our experiments show that, in most cases, the exploitation of modules and symmetries accelerates the FT inference from hours to under three minutes.

1 Introduction

Fault Tree Analysis (FTA) [23,25] is one of the most prominent methods in reliability engineering, used on a daily basis by thousands of engineers. *Fault Trees (FTs)* are a graphical model describing how failures occurring in (atomic) system components propagate through a system and eventually lead to an overall system failure. The quantitative and qualitative analysis of FTs is essential for risk management of complex engineering systems.

An important challenge in FTA is the creation of faithful FT models. Therefore, inference of FTs, also known as *construction* [24], *synthesis* [8], or *induction* [16], has been investigated since the 1970s. Three categories of approaches exist: (i) *Knowledge-based* methods were investigated first, and are semi-automated approaches that derives an FT from a knowledge-based representation using heuristics [3]. These deploy techniques such as decision tables [24,29],

This research has been partially funded by NWO under the grant PrimaVera number NWA.1160.18.238 and by the ERC Consolidator grant CAESAR number 864075.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
M. Trapp et al. (Eds.): SAFECOMP 2022, LNCS 13414, pp. 46–61, 2022.
https://doi.org/10.1007/978-3-031-14835-4_4

mini FTs [21,26], and Piping and Instrumentation Diagrams [26,31]. (ii) *Model-based* techniques derive an FT by translating a system model (e.g., using AADL [11,17], Digraphs [5,12], Simulink [30], or SysML [18,30]) into a FT.

(iii) Due to the increasing availability of inspection and monitoring data, *data-driven* inference methods have emerged. These automatically infer an FT closely matching a given structured data set, exploiting techniques like Bayesian networks [15] and genetic algorithms [10,14]. The resulting FTs closely match the given data set but only contain events also present in the data—and therefore may lack rare events. Nevertheless, data-driven inference can provide a good basis for fault tree creation. A key drawback of data-driven inference methods is that they still lack sufficient *scalability* for larger systems.

In this work, we tackle the scalability challenge of FT inference by exploiting two concepts commonly used in FTs: symmetries and modules. *Symmetries* between components are commonly present in real-world systems, e.g., due to structural properties or redundancies in safety-critical systems. *Modules* correspond to subsystems and allow to subdivide the inference problem into smaller, possibly independent, problems. Our approach, called *SymLearn*, automatically identifies symmetries and modules, and exploits them to reduce the solution space.

We implemented the SymLearn method in Python and numerically evaluated it in five case studies, including three truss system models, which are structural systems typically found in civil infrastructures such as roofs, transmission towers, and bridges. We compare SymLearn to the previous FT-MOEA implementation [10], which was shown to be faster than its predecessor FT-EA [14]. Our experiments show that: (1) SymLearn is orders of magnitude faster than FT-MOEA if modules and symmetries can be exploited; (2) SymLearn is in some cases slower than inference based on Boolean formulas, it yields, however, more compact FTs than Boolean methods.

Contributions. Our main contributions are:

(i) We define modules and symmetries based on the minimal cut sets (MCSs).
(ii) We present algorithms to automatically identify modules and symmetries from the MCSs.
(iii) We introduce *SymLearn*, an approach to automatically infer FTs from failure data sets by exploiting modules and symmetries.
(iv) We implemented SymLearn in Python and numerically evaluated it in several case studies.

The implementation and all data are available at zenodo.org/record/5571811.

Related Work. An early technique for *data-driven* FT inference is the *IFT* algorithm [16], which deploys Quinlan's ID3 algorithm to induce Decision Trees. Inspired by Causal Decision Trees, the *LIFT* algorithm [20] exploits the *Mantel-Haenszel* test to discover dependencies between events. While most data-driven approaches only require information about basic events, LIFT also needs information about failures of intermediate events. Both the *ILTA* [27] and *MILTA* [28]

algorithms make use of *Knowledge Discovery in Data sets*, *Interpretable Logic Tree Analysis*, and *Bayesian probability rules*. The method in [15] first learns a *Bayesian Network* and then translates it into an FT model, using *blacklists* and *whitelists* to define missing or present arcs. The *DDFTA* algorithm [13] infers FTs from time series of failure data via binarization techniques and simplification of Boolean equations. Approaches based on evolutionary algorithms include our earlier work *FT-EA* [14] and *FT-MOEA* [10]. FT-MOEA uses a multi-objective cost function, which outperforms the one-dimensional cost function in FT-EA.

Since FTs encode Boolean functions, FT inference is closely related to synthesis of Boolean circuits with a minimal number of gates [9,19]. Manual simplification of Boolean functions in the context of FT inference is considered in [13]. Common automated methods for simplifying Boolean functions are the Quine–McCluskey algorithm [4] that finds the optimal solution based on prime implicants but only works for a few variables, and the Espresso algorithm [1] that uses efficient heuristics, but does not guarantee finding the optimal solution.

Outline. Section 2 introduces FTs. Sect. 3 defines modules and symmetries. Section 4 details the SymLearn approach. In Sect. 5, we evaluate SymLearn on truss system models and discuss the results. We conclude in Sect. 6 and present future work.

2 Fault Trees

Fault Trees. A *fault tree (FT)* is a directed acyclic graph that models how system component failures occur, propagate, and can lead to a system failure [23,25]. The leaves, called *basic events (BE)*, model (atomic) system components. The intermediate nodes are equipped with a logical *gate* and model how failures propagate through the system. Intermediate nodes with an AND-gate fail if all successor nodes fail, nodes with an OR-gate fail if at least one successor node fails. An FT $\mathcal{F}$ fails if the root node has failed. Figure 1 depicts an FT modeling a computer. *Computer* is equipped with an OR-gate, *Memory* and *Processor* with AND-gates, circles indicate BE.

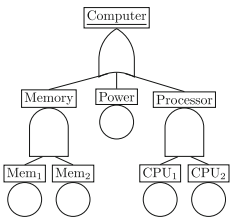


Fig. 1. Example FT.

Definition 1 (Fault tree). *A* fault tree (FT) *is a rooted directed acyclic graph* (V, E) *with a function* $Tp : V \to \{BE, AND, OR\}$ *satisfying* $Tp(v) = BE$ *iff* v *is a leaf. The successors of a node* v *are called the* inputs *of* v *and their set is denoted by* $I(v)$*. All nodes in* V *must be reachable from the dedicated root* Top*.*

We use $\mathsf{BEs} := \{v \in V \mid Tp(v) = \mathsf{BE}\}$ to denote all nodes of type BE. A vector $\vec{b} = \langle b_1, \ldots, b_{|\mathsf{BEs}|} \rangle \in \{0,1\}^{|\mathsf{BEs}|}$ is called a *status vector*. Here $b_i = 1$ indicates that the i-th BE has failed, and $b_i = 0$ that it is functioning properly, respectively. The semantics of an FT $\mathcal{F}$ is given by its *structure function* f.

Definition 2 (Semantics of FT). *Given a status vector $\vec{b}$, the structure function $f : \{0,1\}^{|BEs|} \times V \to \{0,1\}$ returns the status of node v. It is given by*

$$f(\vec{b}, v) := \begin{cases} b_i & \text{if } Tp(v) = \mathit{BE} \text{ and } v \text{ is the } i\text{-th } \mathit{BE}, \\ \bigwedge_{v' \in I(v)} f(\vec{b}, v') & \text{if } Tp(v) = \mathit{AND}, \\ \bigvee_{v' \in I(v)} f(\vec{b}, v') & \text{if } Tp(v) = \mathit{OR}. \end{cases}$$

We use the shorthand $f(\vec{b}) := f(\vec{b}, \mathsf{Top})$. We say FT $\mathcal{F}$ fails for $\vec{b}$ if $f(\vec{b}) = 1$. A status vector $\vec{b}$ can also be given as the set $C = \{b_i \in \vec{b} \mid b_i = 1\}$ of failed BE and we often write $f(C)$ instead of $f(\vec{b})$.

Minimal Cut Sets. *Minimal cut sets (MCSs)* are a common representation of the structure function f. A MCS is a minimal set of BE s.t. the FT fails.

Definition 3 ((Minimal) cut sets). *A* cut set *for FT $\mathcal{F}$ is a set $C \subseteq BEs$ with $f(C) = 1$. A* minimal cut set (MCS) *for $\mathcal{F}$ is a cut set C which is minimal, i.e., for all proper subsets $C' \subsetneq C$, $f(C') = 0$ holds. We denote the set of* all *minimal cuts sets for FT $\mathcal{F}$ by $\mathcal{C}_\mathcal{F}$.*

The FT in Fig. 1 has 3 MCSs: $\mathcal{C}_\mathcal{F} = \{\{\mathrm{Mem}_1, \mathrm{Mem}_2\}, \{\mathrm{Power}\}, \{\mathrm{CPU}_1, \mathrm{CPU}_2\}\}$.

3 Modules and Symmetries

Given a failure data set D, we want to find a compact FT $\mathcal{F}_D$ which matches D.

Failure Data Set. The failure data D is given as a labelled binary data set indicating the failure status of each component, together with the corresponding status of the overall system. Table 1 gives an example corresponding to the FT in Fig. 1 where M_1 corresponds to Mem_1, etc. We assume the data is *coherent*, i.e., once the system fails, it cannot become operational again through further component failures, and it is *noise-free*, i.e., observations with unchanged component states always yield the same system state.

Table 1. Example data.

M_1	M_2	P	C_1	C_2	Sys.
0	0	0	0	1	0
0	0	0	1	1	1
0	0	1	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮

We can also identify MCSs in the failure data D. A (minimal) cut set C of D is a (minimal) set of BEs s.t. the corresponding status vector $\vec{b}$ yields a system failure in D. The set of all MCSs in D is denoted by $\mathcal{C}_D$.

Problem Statement. We want to find an FT $\mathcal{F}_D$ s.t. the structure function f of $\mathcal{F}_D$ captures failure data D as accurately as possible. To assess the quality of the resulting FT w.r.t. input data D, we use three metrics [10]:

– *Size of the FT* ($|\mathcal{F}_D|$) is the number of nodes $|\mathcal{F}_D| := |V|$ in the FT.

- *Error based on data set* D (ϕ_d) is the fraction of times where $\mathcal{F}_D$ fails and the system (according to data set D) does not, and vice versa. Let $E := \left\{\vec{b} \in \{0,1\}^{|\mathsf{BEs}|} \mid f(\vec{b}) \neq D(\vec{b})\right\}$ denote the status vectors which yield different results for $\mathcal{F}_D$ and D. Then the error based on D is given by $\phi_d := \frac{|E|}{|D|}$.
- *Error based on the MCSs* (ϕ_c) compares the set $\mathcal{C}_{\mathcal{F}_D}$ of MCSs of the FT $\mathcal{F}_D$ and the set of MCSs $\mathcal{C}_D$ derived from the data D. The metric ϕ_c computes the similarities between both sets of MCSs based on the RV-coefficient [22], see [10] for the details.

Formal Problem. Given a failure data set D, create a (compact) FT $\mathcal{F}_D$ s.t. its BEs correspond to the atomic components in D and $f(\vec{b})$ captures the system failures in D as accurately as possible. In other words, ϕ_c and ϕ_d should be (close to) zero, and $|\mathcal{F}_D|$ should be as small as possible.

In our approach, we first create $\mathcal{C}_D$ from D and infer the FT $\mathcal{F}_{\mathcal{C}_D}$.

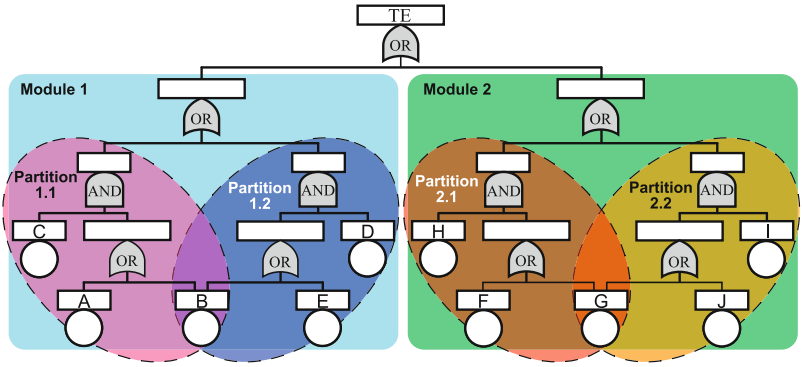


Fig. 2. FT with independent modules and further partitioning. (Color figure online)

3.1 Modules

Instead of directly inferring an FT $\mathcal{F}_{\mathcal{C}_D}$ from the MCSs $\mathcal{C}_D$, we aim to first partition $\mathcal{C}_D$ into multiple parts, infer individual FTs for each of them, and then combine the FTs into the overall FT $\mathcal{F}_{\mathcal{C}_D}$.

Definition 4 (MCS partitioning). *Let* $M_1, \ldots, M_n \subseteq \mathcal{C}$ *be a* partitioning of the set $\mathcal{C}$ of MCSs, *i.e.,* $M_i \cap M_j = \emptyset$ *for all* $i \neq j$ *and* $M_1 \cup \cdots \cup M_n = \mathcal{C}$. *For a partition* M_i, *we let* $\mathsf{BEs}^{M_i} := \bigcup_{C \in M_i} C$ *denote the set of* BE *occurring in* M_i. BE *occurring in multiple partitions are called the* shared BE.

In the case of a large number of shared BE, the inferred FTs—which each might be optimal individually—can yield an overall FT which is sub-optimal. For example, gates with (some of the) shared BE as input might occur in multiple

FTs. Thus, the goal is to find a partitioning such that the number of shared BE is as small as possible. If no BE are shared, the resulting partitioning of BEs forms *independent modules*. In FTs, (independent) modules are independent subtrees, where only the root node is connected to other parts of the FT [7]. Modules can therefore be thought of as coherent entities in the context of the overall system, e.g., components. Modularization is used to simplify the FT analysis.

Definition 5 (Modules). *A partitioning $M_1, \dots M_n$ of the set $\mathcal{C}$ of MCSs is called a* module partitioning *if the corresponding $\mathsf{BEs}^{M_1}, \dots, \mathsf{BEs}^{M_n}$ form a partitioning of BEs. A subset $\mathfrak{M}$ of BEs is called an* independent module *if it is part of a module partitioning, i.e., all BE of $\mathfrak{M}$ are included in MCSs of a single M_i.*

An independent module $\mathfrak{M}$ does not share BE. Thus, the BE in $\mathfrak{M}$ are not connected to other parts of the FT and they belong to an independent subtree.

Example 1 (Modules). The partitioning for the FT in Fig. 2 is given by colored boxes. The BEs $\{A, B, C, D, E\}$ and $\{F, G, H, I, K\}$ form independent modules. The corresponding MCSs can be further subdivided. For instance, Partition 1.1 with $\{\{A, C\}\{B, C\}\}$ and Partition 1.2 with $\{\{B, D\}, \{D, E\}\}$ share BE B.

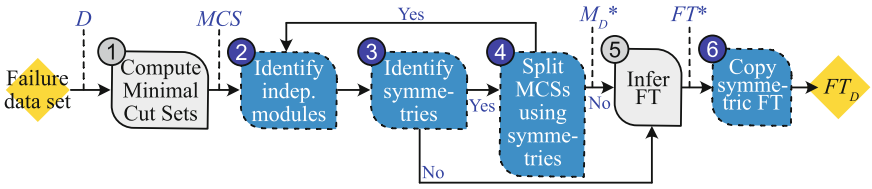


Fig. 3. *SymLearn* tool chain overview. Blue boxes indicate novel steps. (Color figure online)

3.2 Symmetries

Symmetries in an FT describe components, e.g., BE or complete subtrees, that can be swapped without changing the failure behavior of the FT. In our setting, symmetries reduce the computational effort for inferring FTs as only one of the sub-trees must be constructed; other subtree(s) can be copied from the (original) subtree because of the symmetry. We define symmetries on the MCSs. Applying a symmetry on the MCSs yields the same MCSs, i.e., swapping symmetric BE does not change the structure function of the FT.

Definition 6 (Symmetry on MCSs). *A* symmetry *on the set $\mathcal{C}$ of all MCSs is a permutation $\sigma : \mathsf{BEs} \to \mathsf{BEs}$ which preserves $\mathcal{C}$, i.e., $\sigma(\mathcal{C}) = \mathcal{C}$ where $\sigma(\mathcal{C}) := \{\sigma(C) \mid C \in \mathcal{C}\}$ and $\sigma(C) := \{\sigma(b) \mid b \in C\}$.*

We denote all possible symmetries on $\mathcal{C}$ by $\mathcal{S}_\mathcal{C}$. A *symmetry between sets* $A, B \subseteq$ BEs is a symmetry $\sigma \in \mathcal{S}_\mathcal{C}$ with $\sigma(A) \subseteq B$ and $\sigma(B) \subseteq A$. Note that we define symmetries only on BEs and not on gates. The definition is thus more general and allows symmetries even in cases where sub-trees are not isomorphic.

Lemma 1 (Necessary condition for symmetry). *If $\sigma \in \mathcal{S}_\mathcal{C}$ is a symmetry on the MCSs $\mathcal{C}$, then count(b) = count$(\sigma(b))$ for all $b \in$ BEs, where count$(b) := |\{C \in \mathcal{C} \mid b \in C\}|$ denotes the number of occurrences of b in $\mathcal{C}$.*

Example 2 (Symmetry). Consider again the FT $\mathcal{F}$ in Fig. 2. The permutation $\sigma_1 = (AF)(BG)(CH)(DI)(EJ)$ is a symmetry in $\mathcal{F}$ (between the independent modules). For example, $\sigma_1(\{A, C\}) = \{F, H\} \in \mathcal{C}_\mathcal{F}$. Symmetries within the modules are given by $\sigma_2 = (AE)(CD) \in \mathcal{S}_{\mathcal{C}_\mathcal{F}}$ and $\sigma_3 = (FJ)(HI) \in \mathcal{S}_{\mathcal{C}_\mathcal{F}}$.

4 Exploiting Modules and Symmetries in FT Inference

Our *SymLearn* approach is outlined in Fig. 3 and consists of 6 steps:

Step 1 computes the set of all *MCSs* $\mathcal{C}_D$ associated with input data set D.
Step 2 finds a partitioning $M_1, \ldots, M_n$ of $\mathcal{C}_D$ s.t. the corresponding BEs form *independent modules* $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$. In the worst case, no proper partitioning is possible and the independent module consists of all BEs.
Step 3 identifies the *symmetries* $\mathcal{S}_{\mathcal{C}_D}$ on $\mathcal{C}_D$. If symmetries exist between independent modules, then only one of these modules needs to be considered in the following. Otherwise, SymLearn directly goes to Step 5.
Step 4 tries to further *split* the MCSs M_i of each module $\mathfrak{M}_i$ via a symmetry $\sigma \in \mathcal{S}_{\mathcal{C}_D}$. The split into M_i^1 and M_i^2 should satisfy $\sigma(M_i^1) = M_i^2$ and preferably have a small number of shared BE. If a split is found, SymLearn recursively starts again with Step 2 for M_i^1; otherwise it proceeds with Step 5.
Step 5 infers an FT $\mathcal{F}_M$ for each partition M of the MCSs. Several approaches can be used, e.g., *FT-MOEA* [10] or simplification of Boolean formulas [13].
Step 6 creates for each set of symmetric MCSs M_i^2 a corresponding *symmetric FT* $\mathcal{F}_{M_i^2}$ by copying the "original" FT $\mathcal{F}_{M_i^1}$ and renaming the BEs according to the symmetry σ. Last, all inferred FTs are joined under an OR-gate.

We provide details on all steps of SymLearn in the following.

Step 1: Compute Minimal Cut Sets. SymLearn starts by extracting all the MCSs $\mathcal{C}_D$ from the data D. We use the algorithm from [13], but employ an improved computation of the MCSs from the cut sets. Here, we iteratively select a cut set C with minimal cardinality and remove all cut sets that include C. The runtime complexity of the algorithm is quadratic in D, i.e., $\mathcal{O}(D^2) = \mathcal{O}(2^{2 \cdot |\text{BEs}|})$.

Algorithm 1. Identifying independent modules $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$ from the MCSs $\mathcal{C}_D$.

Input: MCSs $\mathcal{C}_D$.

Output: Partitioning $M_1, \ldots, M_n$ of $\mathcal{C}_D$, corresp. independent modules $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$.

$Partitioning \leftarrow \{\{C\} \mid C \in \mathcal{C}_D\}$

while $\exists M, M' \in Partitioning$ with M and M' sharing BE **do**

$Partitioning \leftarrow (Partitioning \setminus \{M, M'\}) \cup \{M \cup M'\}$

return $Partitioning = \{M_1, \ldots, M_n\}$, modules $\{\mathfrak{M}_1 = \mathsf{BEs}^{M_1}, \ldots, \mathfrak{M}_n = \mathsf{BEs}^{M_n}\}$

Step 2: Identify Independent Modules. Our aim is to partition the MCSs $\mathcal{C}_D$ s.t. an FT for each partition can be learned individually. This allows for a more efficient inference which could even be performed in parallel.

We start by trying to find independent modules from $\mathcal{C}_D$ as described in Algorithm 1. The initial partitioning uses each cut set of $\mathcal{C}_D$ as its own partition. If two partitions share BE, they must be merged to satisfy the constraint for independent modules in Definition 5. We iteratively merge partitions until their BEs are disjoint. The BEs then form the independent modules. The following Steps 3–5 are performed for each independent module and corresponding MCSs individually. The FTs created for the modules are combined by an OR-gate in the end.

Example 3 (Identify independent modules). We use the MCSs $\mathcal{C}_D = \{\{A, C\}, \{B, C\}, \{B, D\}, \{D, E\}, \{F, H\}, \{G, H\}, \{G, I\}, \{I, K\}\}$ corresponding to Fig. 2. Applying the algorithm, cut sets $\{A, C\}$ and $\{B, C\}$, for instance, are merged as they share BE C. In the end, the independent modules and partitioning are:

$$\mathfrak{M}_1 = \{A, B, C, D, E\} \qquad M_1 : \{\{A, C\}\{B, C\}, \{B, D\}, \{D, E\}\}$$
$$\mathfrak{M}_2 = \{F, G, H, I, K\} \qquad M_2 : \{\{F, H\}\{G, H\}, \{G, I\}, \{I, K\}\}$$

Extraction of BE. As an additional optimization, we automatically derive BE which occur in all minimal cut sets of a partition. In order for the partition to cause a system failure, all these BE must fail. Hence, they are excluded from all MCSs and the approach continues on the reduced MCS. In the end, the excluded BE are joined under an AND-gate with the FT resulting from the reduced MCSs.

Step 3: Identify Symmetries. Next, we identify the symmetries $\mathcal{S}_{\mathcal{C}_D}$ from $\mathcal{C}_D$ in a fully automated manner. The simplest way is a brute-force approach trying out all possible permutations and checking whether they are valid symmetries according to Definition 6. While this approach is factorial in $|\mathsf{BEs}|$, we obtain good performance in practice by exploiting two optimizations.

Symmetries Between Independent Modules. The most efficient approach is to exploit the independent modules from the previous step. Symmetries between two independent modules $\mathfrak{M}, \mathfrak{M}'$ can be quickly found by restricting the permutations to only the ones matching each BE in $\mathfrak{M}$ to one in $\mathfrak{M}'$.

Algorithm 2. Splitting of MCS M_i into two symmetric parts M_i^1 and M_i^2.

Input: MCS M_i, symmetry $\sigma \in \mathcal{S}_{\mathcal{C}_D}$
Output: Symmetric MCSs M_i^1, M_i^2 with corresponding contained BE $\mathsf{BEs}^{M_i^1}, \mathsf{BEs}^{M_i^2}$

```
M_i^1 ← ∅, M_i^2 ← ∅, BEs_1 ← ∅, BEs_2 ← ∅
Q ← C_D
while C ∈ Q do
    if C = σ(C) then return M_i, ∅, BEs^{M_i}, ∅
    Q ← Q \ {C, σ(C)}
    if |C ∩ BEs_1| ≥ |C ∩ BEs_2| then
        M_i^1 ← M_i^1 ∪ {C}, M_i^2 ← M_i^2 ∪ {σ(C)}, BEs_1 ← BEs_1 ∪ C, BEs_2 ← BEs_2 ∪ σ(C)
    else
        M_i^1 ← M_i^1 ∪ {σ(C)}, M_i^2 ← M_i^2 ∪ {C}, BEs_1 ← BEs_1 ∪ σ(C), BEs_2 ← BEs_2 ∪ C
return M_i^1, M_i^2, BEs_1, BEs_2
```

Fast Exclusion of Non-symmetric BEs. If only one independent module was found in Step 2, then the symmetries must be computed by an exhaustive search. However, we can exclude infeasible permutation candidates early on by using Lemma 1. Two BE with different numbers of occurrences in $\mathcal{C}_D$ cannot be symmetric and thus, all permutations containing such mappings are excluded.

Example 4 (Identify symmetries). Continuing Example 3, we find the symmetry $\sigma_1 = (AF)(BG)(CH)(DI)(EK)$ between independent modules $\mathfrak{M}_1$ and $\mathfrak{M}_2$. As a result, the symmetric set M_2 of MCSs will not be considered in the remainder. We continue by searching for symmetries within $\mathfrak{M}_1$ according to M_1. Candidate permutations such as (AC) are quickly excluded, because $\text{count}(A) = 1 \neq 2 = \text{count}(C)$. In the end, symmetry $\sigma_2 = (AE)(CD)$ is found.

Step 4: Split MCSs Using Symmetries. A symmetry σ found in the previous step can be used to split the MCSs M_i. We restrict ourselves to splits into two parts here, but more parts work in the same manner. A successful split creates two symmetric subsets M_i^1 and M_i^2 of M_i with $\sigma(M_i^1) = M_i^2$.

Algorithm 2 describes the split of the MCSs M_i according to a symmetry $\sigma \in \mathcal{S}_{\mathcal{C}_D}$. Initially, the queue Q contains all MCSs from $\mathcal{C}_D$. For each MCS C we compute the symmetric MCS $\sigma(C)$. If C is symmetric to itself ($C = \sigma(C)$), a split would add the same MCS to both parts. As this would only increase the size of the resulting FTs, we do not proceed further. If both MCSs are distinct, we add C to the set of MCSs with which it shares the most BE. For example, we add C to M_i^1 if $|C \cap \mathsf{BEs}_1| \geq |C \cap \mathsf{BEs}_2|$. By this choice, we ensure that adding C to M_i^1 does not add too many new BE to BEs_1 and we keep the number of shared BE between BEs_1 and BEs_2 small.

Note that the split can still yield two parts which share a significant amount of BE. Composing the two resulting FTs can therefore yield an FT which is larger than the single FT inferred without the split. However, the composed FT will capture the symmetric structure present in the given MCSs.

Example 5 (Split the MCSs). We continue with symmetry $\sigma_2 = (AE)(CD)$ and MCSs $M_1 = \{\{A, C\}, \{B, C\}, \{B, D\}, \{D, E\}\}$ from Example 4. We start the algorithm with MCS $\{A, C\}$. The symmetric MCS is $\sigma(\{A, C\}) = \{D, E\}$. The first split yields $M_1^1 = \{\{A, C\}\}$ and $M_1^2 = \{\{D, E\}\}$. The next MCS $\{B, C\}$ is added to M_1^1 because they both share BE C. The final split is:

$$M_1^1 = \{\{A, C\}, \{B, C\}\} \qquad \mathsf{BEs}_1 = \{A, B, C\},$$
$$M_1^2 = \{\{D, E\}, \{B, D\}\} \qquad \mathsf{BEs}_2 = \{B, D, E\}.$$

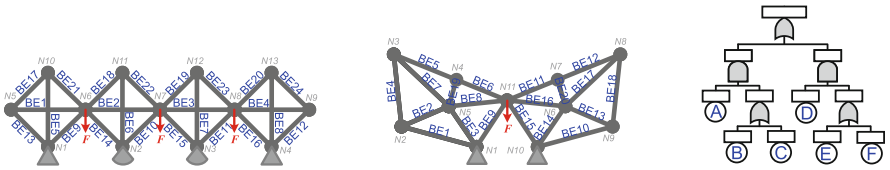
The split corresponds to the purple and dark blue sub-trees in Fig. 2.

Step 5: Infer FT. If no further partitioning of the MCSs M_i w.r.t. Steps 2–4 is possible, we use existing techniques to infer an FT from the (reduced) MCSs. SymLearn is modular and supports the use of any learning approach in this step, for example, based on genetic algorithms [14] or Boolean logic [13]. In our setting, we use the multi-objective evolutionary algorithm *FT-MOEA* [10].

FT-MOEA starts in the first generation by default with two *parent FTs*: one FT consists of an AND-gate connected to all BEs, and the other one uses an OR-gate. In each generation, several *genetic operators* are applied which randomly modify the FT structure. Each FT is evaluated according to three metrics given in Sect. 3: size of the FT $|\mathcal{F}|$, error based on the failure data set (ϕ_d), and error based on the set of MCSs (ϕ_c). The aim is to minimize the multi-objective function $(|\mathcal{F}|, \phi_d, \phi_c)$ by applying the *Elitist Non-dominated Sorting Genetic Algorithm* (NSGA-II) [6] and obtain the Pareto sets. Only the best candidates according to the metrics are then passed to the next generation. The algorithm stops if no improvement was made in a given number of generations and returns the FTs ordered according to the multi-objective function.

Example 6 (FT-MOEA). Given the MCS $\{\{A, C\}, \{B, C\}\}$, we use FT-MOEA to infer a FT. The resulting FT is the sub-tree indicated by purple color in Fig. 2.

Step 6: Copy Symmetric FTs. After obtaining an FT $\mathcal{F}_M$ for MCSs M, we obtain the symmetric FT $\mathcal{F}_{M'}$ for the symmetric MCSs $M' = \sigma(M)$ by copying $\mathcal{F}_M$ and replacing each BE b with its symmetric BE $\sigma(b)$. The original and the symmetric FT are then joined under an OR-gate.



(a) Case TS2. (b) Case TS3. (c) Case SC.

Fig. 4. Visualization of case studies TS2, TS3 and SC.

Example 7 (Copy symmetric FT). We continue with Example 6. Copying the purple sub-tree in Fig. 2 and applying symmetry $\sigma_2 = (AE)(CD)$ yields the symmetric (dark blue) FT. Joining both FTs with an OR-gate yields *Module 1*.

5 Experimental Evaluation

We implemented the *SymLearn* methodology in a Python toolchain, available at zenodo.org/record/5571811, and evaluate our approach on five case studies, see Table 2: *Cases SC* and *SS* are two small systems, depicted in Fig. 4c (case SC) and running example of Fig. 2 (case SS). We also consider three *truss system models*.

Table 2. Overview of case studies.

Case	#BEs	$\lvert D\rvert$	$\lvert\mathcal{C}_D\rvert$
SC	6	64	4
SS	10	1024	8
TS1	10	1024	16
TS2	24	16 777 216	26
TS3	20	1 048 576	18

Truss System Cases. Truss systems are commonly used in civil infrastructures such as roofs, transmission towers, and bridges, see Fig. 5a. Truss systems are composed of elements connected by nodes, generating rigid bodies with the elements acting under tensile stresses.

Truss systems feature a high degree of symmetry and a modular structure. Moreover, as elaborated below, they allow us to obtain the failure data sets via structural analysis (similar to [2]). Therefore, we consider truss systems to be a very suitable model to evaluate *SymLearn* in a realistic setting.

We use three truss system variants: Cases TS1 (Fig. 5a) and TS2 (Fig. 4a) are typical configurations in bridges, while Case TS3 (Fig. 4b) is found in roofs. Note that Case TS1 contains no independent modules, whereas TS2 and TS3 contain four and two modules, respectively.

Generation of Failure Data Set. Based on case TS1 (Fig. 5) we explain how we use numerical truss system models to generate complete failure data sets. TS1 consists of 10 elements (interpreted as BEs), and two symmetric loads applied on the control nodes. We model damage by reducing close to zero the cross-sectional area of at least one element in the truss system model, and by determining the displacements and stresses in the components due to the applied loads at the

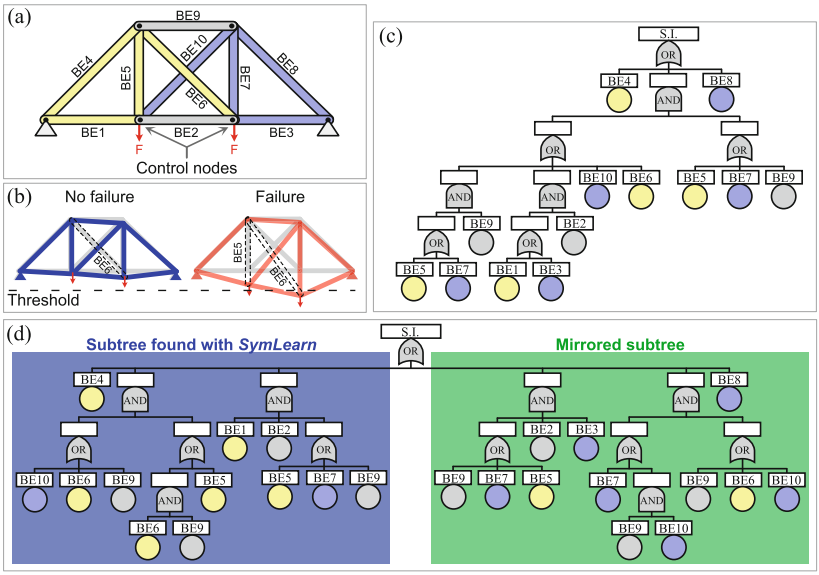


Fig. 5. Example case TS1 modeling a symmetric truss bridge system. (a) Model. (b) Depiction of failure/no-failure states. (c) FT inferred by FT-MOEA. (d) FT inferred by SymLearn. Top corresponds to the truss system instability. (Color figure online)

nodes of the numerical model. We generate a synthetic failure data set D by randomly drawing 10^6 data points for the status of elements in the truss model via Monte Carlo simulation, and evaluating *structural instability* (S.I.) based on the displacement of control nodes.

Experimental Setup. We compare the SymLearn tool with 3 different backends in Step 5, to infer the FT from data.

- *FT-MOEA* is used in 4 different settings: (1) *All* is the default setting using both modules and symmetries; (2) *No Sym* is *All* but without symmetries; (3) *No rec.* is *All* but without recursive calls for further sub-division; (4) *FT-MOEA* is the original implementation [10] without modules and symmetries.
- *Espresso* translates a set of MCSs $\mathcal{C}_D$ into a Boolean formula $\bigvee_{C \in \mathcal{C}_D} \bigwedge_{b \in C} b$ and simplifies it via the *ESPRESSO* algorithm [1] available in PYEDA[1]. The resulting formula is then translated into an FT.
- *Sympy* is similar to *Espresso* but uses the SYMPY library[2] for simplification.

We ran all case studies three times on a CPU with 2.3 GHz and 8 GB of RAM.

[1] https://pyeda.readthedocs.io/en/latest/2llm.html.
[2] https://docs.sympy.org/latest/modules/logic.html.

Results. We compare the FTs for case TS1 inferred via FT-MOEA (Fig. 5c) and via SymLearn in configuration *All* (Fig. 5d). Colors depict the connections of the BEs to the components in Fig. 5a. SymLearn identified the symmetry (between yellow and blue BE) and was able to infer the left subtree using FT-MOEA while the right subtree was obtained by simple mirroring.

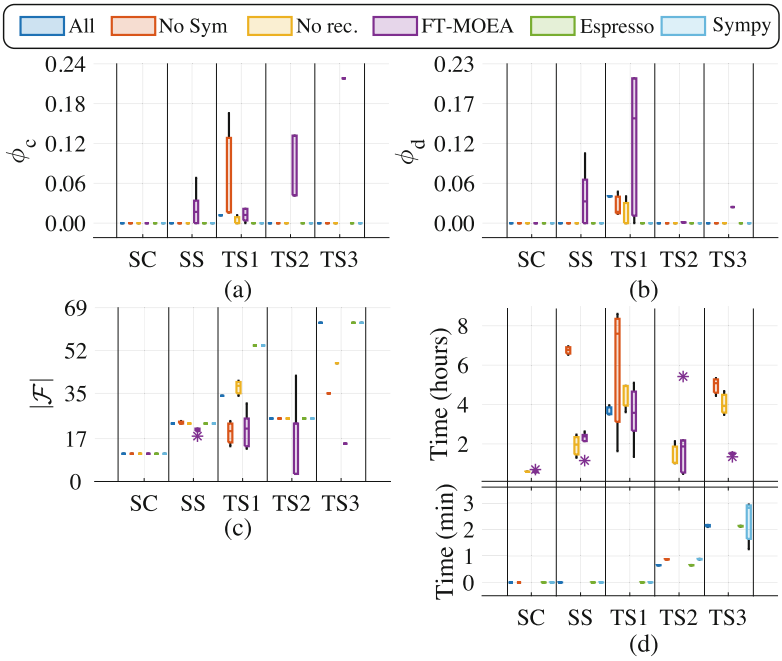


Fig. 6. Results for the case studies and different metrics: (a) error ϕ_c based on the MCSs, (b) error ϕ_d based on data set, (c) FT size $|\mathcal{F}|$, and (d) runtime.

The box charts in Fig. 6 compare the different configurations in all five cases w.r.t. the three metrics in Sect. 3: the size $|\mathcal{F}|$ of the FT, the error ϕ_d based on the failure data set, and the error ϕ_c based on the MCSs. From Fig. 6a and 6b, we see that the SymLearn configurations based on Boolean functions as a back-end (i.e., *Espresso* and *Sympy*) always yield an FT that exactly matches the input, i.e., $\phi_c = \phi_d = 0$. This is expected since the Boolean logic formula perfectly encodes all the MCSs. In contrast, the other configurations using FT-MOEA did not always yield a completely accurate FT (i.e., $\phi_c, \phi_d > 0.0$), for example, case TS1. The error stems from the multi-objective optimization which also aims to provide a small FT and the evolutionary algorithm which can fall into local optima. However, for the cases TS2 and TS3 (with independent modules), all configurations of SymLearn (*All*, *No Sym*, *No rec.*) outperformed FT-MOEA by returning an FT that accurately reflects the input ($\phi_c = \phi_d = 0.0$). This shows the clear benefit of subdividing the problem using independent modules.

Figure 6c shows the advantage of using FT-MOEA as a back-end compared to Boolean logic, since the sizes of the returned FTs can be considerably smaller. The FTs inferred using Espresso or Sympy can be twice as large as the ones resulting from FT-MOEA. The reason is that for the Boolean logic formulas, no simplifications were performed by the libraries and the resulting FTs are therefore exactly encoding all the MCSs. Notice that the original FT-MOEA yields smaller or equal FT sizes than any of the configurations of SymLearn. This smaller size can however also come at the cost of losing accuracy, as demonstrated by case TS2. The larger FTs in SymLearn mostly stem from the composition of partitions where shared BE occur in both sub-trees, see for example Fig. 5c and 5d. While explicitly capturing the symmetries can therefore increase the size of the resulting FT, it also provides more insights into the system.

Figure 6d shows that SymLearn (*All*) runs significantly faster than FT-MOEA alone. If independent modules are present (cases TS2, TS3, SC and SS), SymLearn yields an FT within at most 2 min while FT-MOEA requires at least 1 h. The benefit of exploiting symmetries and modules can also be seen when comparing configuration *All* to *No Sym* and *No. rec.* which both run longer. Note that for SymLearn nearly all computation time is spent in the FT-MOEA backend (Step 5). Computing the modules and symmetries (Steps 2–4) took 50 ms at most whereas the computation of the MCSs (Step 1) took 43 s at most (for case TS2). Configurations based on Boolean functions always yield a result within minutes, but yield significantly larger FTs.

6 Conclusions

We presented *SymLearn*, a data-driven algorithm that infers a Fault Tree model from given failure data in a fully automatic way by identifying and exploiting modules and symmetries. Our evaluation based on truss system models shows that SymLearn is significantly faster than only using evolutionary algorithms when modules and symmetries can be exploited.

In the future, we aim to further improve the scalability by *optimizing the inference process*. First, the current partitioning of the MCSs requires the top gate to be an OR-gate. We aim to support the AND-gate as well. In addition, the inference back-end can be improved by either optimizing FT-MOEA or developing new inference approaches.

We also plan to *relax restrictions on the input data*. In the current approach, the resulting FTs are only as good as the given input data, which may be incomplete, e.g., due to rare events not present in the data. Moreover, the input may not completely represent the reality due to noise in the data. Hence, we aim to extend our approach to account for missing information and noise.

Acknowledgment. We thank Milan Lopuhaä-Zwakenberg for useful comments on an earlier version of this paper.

References

1. Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L.: Logic Minimization Algorithms for VLSI Synthesis. The Springer International Series in Engineering and Computer Science, vol. 2. Springer, New York (1984). https://doi.org/10.1007/978-1-4613-2821-6
2. Byun, J., Song, J.: Efficient probabilistic multi-objective optimization of complex systems using matrix-based Bayesian network. Reliab. Eng. Syst. Saf. **200**, 106899 (2020)
3. Carpignano, A., Poucet, A.: Computer assisted fault tree construction: a review of methods and concerns. RESS **44**(3), 265–278 (1994)
4. Coudert, O.: Two-level logic minimization: an overview. Integration **17**(2), 97–140 (1994)
5. De Vries, R.C.: An automated methodology for generating a fault tree. IEEE Trans. Reliab. **39**(1), 76–86 (1990)
6. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
7. Dutuit, Y., Rauzy, A.: A linear-time algorithm to find modules of fault trees. IEEE Trans. Reliab. **45**(3), 422–425 (1996)
8. Hunt, A., Kelly, B., Mullhi, J., Lees, F., Rushton, A.: The propagation of faults in process plants: 6, overview of, and modelling for, fault tree synthesis. RESS **39**(2), 173–194 (1993)
9. Jiang, J.H.R., Devadas, S.: Logic synthesis in a nutshell. In: Electronic Design Automation, pp. 299–404. Elsevier, Amsterdam (2009)
10. Jimenez-Roa, L.A., Heskes, T., Tinga, T., Stoelinga, M.: Automatic inference of fault tree models via multi-objective evolutionary algorithms. CoRR abs/2204.03743 (2022)
11. Joshi, A., Vestal, S., Binns, P.: Automatic generation of static fault trees from AADL models (2007)
12. Lapp, S.A., Powers, G.J.: Computer-aided synthesis of fault-trees. IEEE Trans. Reliab. **26**(1), 2–13 (1977)
13. Lazarova-Molnar, S., Niloofar, P., Barta, G.K.: Data-driven fault tree modeling for reliability assessment of cyber-physical systems. In: WSC. IEEE (2020)
14. Linard, A., Bucur, D., Stoelinga, M.: Fault trees from data: efficient learning with an evolutionary algorithm. In: Guan, N., Katoen, J.-P., Sun, J. (eds.) SETTA 2019. LNCS, vol. 11951, pp. 19–37. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35540-1_2
15. Linard, A., Bueno, M.L., Bucur, D., Stoelinga, M.: Induction of fault trees through Bayesian networks. In: ESREL, pp. 910–917. Research Publishing (2019)
16. Madden, M.G., Nolan, P.J.: Generation of fault trees from simulated incipient fault case data. WIT Trans. Inf. Commun. Technol. **6** (1994)
17. Mahmud, N., Mian, Z.: Automatic generation of temporal fault trees from AADL models. In: ESREL, pp. 2741–2749 (2013)
18. Mhenni, F., Nguyen, N., Choley, J.: Automatic fault tree generation from SysML system models. In: AIM, pp. 715–720. IEEE (2014)
19. Murray, C.D., Williams, R.R.: On the (non) NP-hardness of computing circuit complexity. Theory Comput. **13**(1), 1–22 (2017)
20. Nauta, M., Bucur, D., Stoelinga, M.: LIFT: learning fault trees from observational data. In: McIver, A., Horvath, A. (eds.) QEST 2018. LNCS, vol. 11024, pp. 306–322. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99154-2_19

21. Powers, G.J., Tompkins, F.C., Jr.: Fault tree synthesis for chemical processes. AIChE J. **20**(2), 376–387 (1974)
22. Robert, P., Escoufier, Y.: A unifying tool for linear multivariate statistical methods: the rv- coefficient. J. Roy. Stat. Soc. Ser. C (Appl. Stat.) **25**(3), 257–265 (1976)
23. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015)
24. Salem, S.L., Apostolakis, G., Okrent, D.: Computer-oriented approach to fault-tree construction. Technical report, California University (1976)
25. Stamatelatos, M., Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: Fault tree handbook with aerospace applications (2002)
26. Taylor, J.: An algorithm for fault-tree construction. IEEE Trans. Reliab. **31**(2), 137–146 (1982)
27. Waghen, K., Ouali, M.: Interpretable logic tree analysis: a data-driven fault tree methodology for causality analysis. Expert Syst. Appl. **136**, 376–391 (2019)
28. Waghen, K., Ouali, M.: Multi-level interpretable logic tree analysis: a data-driven approach for hierarchical causality analysis. Expert Syst. Appl. **178**, 115035 (2021)
29. Wang, J., Liu, T.: A component behavioural model for automatic fault tree construction. RESS **42**(1), 87–100 (1993)
30. Xiang, J., Yanoo, K., Maeno, Y., Tadano, K.: Automatic synthesis of static fault trees from system models. In: SSIRI, pp. 127–136. IEEE Computer Society (2011)
31. Xie, G., Xue, D., Xi, S.: Tree-expert: a tree-based expert system for fault tree construction. RESS **40**(3), 295–309 (1993)