Fault Tree Reliability Analysis via Squarefree Polynomials

Milan Lopuhaä-Zwakenberg
University of Twente, Enschede, The Netherlands

Keywords: Fault Trees, Reliability Analysis, Polynomial Algebra.

Abstract: Fault tree (FT) analysis is a prominent risk assessment method in industrial systems. Unreliability is one of the key safety metrics in quantitative FT analysis. Existing algorithms for unreliability analysis are based on binary decision diagrams, for which it is hard to give time complexity guarantees beyond a worst-case exponential bound. In this paper, we present a novel method to calculate FT unreliability based on algebras of squarefree polynomials and prove its validity. We furthermore prove that time complexity is low when the number of multiparent nodes is limited. Experiments show that our method is competitive with the state-of-the-art and outperforms it for FTs with few multiparent nodes.

1 INTRODUCTION

Fault trees. Fault trees (FTs) form a prominent risk assessment method to categorize safety risks on industrial systems. A FT is a hierarchical graphical model that shows how failures may propagate and lead to system failure. Because of its flexibility and rigor, FT analysis is incorporated in many risk assessment methods employed in industry, including FaultTree+ (IsoTree, 2023) and TopEvent FTA (Reliotech, 2023).

A FT is a directed acyclic graph (not necessarily a tree) whose root represents system failure. The leaves are called *basic events* (BEs) and represent atomic failure events. Intermediate nodes are AND/OR-gates, whose activation depends on that of their children; the system as a whole fails when the root is activated. An example is given in Fig. 1.

Quantitative analysis. Besides a qualitative analysis of what sets of events cause overall system failure, FTs also play an important role in *quantitative risk analysis*, which seeks to express the safety of the system in terms of safety metrics, such as the total expected downtime, availability, etc. An important safety metric is *(un)reliability*, which, given the failure probability of each BE, calculates the probability of system failure. As the size of FTs can grow into the hundreds of nodes (Ruijters et al., 2019), calculating the unreliability efficiently is crucial for giving safety and availability guarantees.

There exist two main approaches to calculating unreliability (Ruijters and Stoelinga, 2015). The first approach works bottom-up, recursively calculating the failure probability of each gate. This algorithm is fast (linear time complexity), but only works as long as the FT is actually a tree. However, nodes with multiple parents (DAG-like FTs) are necessary to model more intricate systems. For such FTs, as we show in this paper, calculating unreliability is NP-hard. The main approach for such FTs is based on translating the FT into a binary decision diagram (BDD) and performing a bottom-up analysis on the BDD. This BDD is of worst-case exponential size, though heuristics exist. The BDD corresponding to an FT depends on a linear ordering of the BEs, with different orderings yielding BDDs of wildly varying size; although any single one of them can be used to calculate unreliability, finding the optimal BE ordering is an NP-hard problem in itself. As a result, it is hard to give guarantees on the runtime of this unreliability calculation algorithm in terms of properties of the FT.

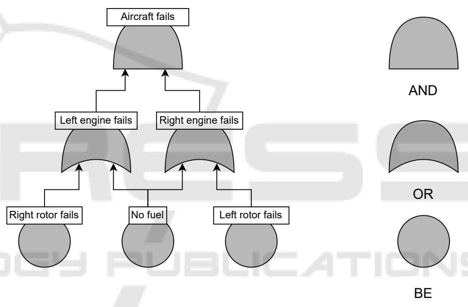


Figure 1: A fault tree for a small aircraft. The aircraft fails if both its engines fail; each engine fails if either its rotor fails or it has no fuel (the plane has a single fuel tank).

Lopuhaä-Zwakenberg, M.
Fault Tree Reliability Analysis via Squarefree Polynomials.
DOI: 10.5220/0012334000003645
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 39-49
ISBN: 978-989-758-682-8; ISSN: 2184-4348
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

Contributions. In this paper, we present a radical new way for calculating unreliability for general FTs. The bottom-up algorithm does not work for DAG-like FTs, since it does not recognize multiple copies of the same node in the calculation, leading to double counting. In our approach we amend this by keeping track of nodes with multiple parents, as these may occur twice in the same calculation. Then, instead of propagating failure probabilities as real numbers, we propagate squarefree polynomials whose variables represent the failure probabilities of nodes with multiple parents; keeping these formal variables allows us to detect and account for double counting. Furthermore, to keep complexity down we replace formal variables with real numbers whenever we are able. At the root all formal variables have been substituted away, yielding the unreliability as a real number.

This approach has as advantage over BDD-based algorithms that we are able to give upper bounds to computational complexity. Most of the complexity comes from the fact that we do arithmetic with polynomials rather than real numbers. However, if the number of multiparent nodes is limited, these polynomials have limited degree, and computation is still fast. We prove this formally, by showing that the time complexity is linear when the number of multiparent nodes is bounded, and in experiments, in which we compare our method to Storm-dft (Basgöze et al., 2022b), a state-of-the-art tool for FT analysis using a BDD-based approach: here our method is competitive in general and is considerably faster for FTs with few multiparent nodes.

Summarized our contributions are the following:

1. A new algorithm for fault tree reliability analysis based on squarefree polynomial algebras;
2. A proof of the algorithm's validity and bounds on its time complexity;
3. Experiments comparing our algorithm to the state-of-the-art.

An artefact of the experiments is available at (Lopuhaä-Zwakenberg, 2023a). Furthermore, a version of this paper including proofs of the mathematical statements is available at (Lopuhaä-Zwakenberg, 2023b).

2 RELATED WORK

There exists a considerable amount of work on FT reliability analysis. FTs were first introduced in (Watson, 1961). A bottom-up algorithm to calculate their reliability is presented in (Ruijters and Stoelinga, 2015), which is a formalization of mathematical principles that have been used since the beginning of FT analysis. This algorithm works only for FTs that are actually trees, i.e., do not contain nodes with multiple parents. In (Rauzy, 1993), a BDD-based method for calculating reliability was introduced that also works for DAG-shaped FTs. This method is still the state of the art, although some improvements have been made, notably by tweaking variable ordering to obtain smaller BDDs (Bouissou et al., 1997) and dividing the FT into so-called modules that can be handled separately (Rauzy and Dutuit, 1997).

FTs are also used to model the system's reliability over time. The failure time of each BE is modeled as a random variable (typically exponentially distributed), and additional gate types are introduced to model more elaborate timing behavior. Reliability analysis of these *dynamic FTs* is done via stochastic model checking (Basgöze et al., 2022b).

3 FAULT TREES

In this section, we give the formal definition of fault trees (FTs) and their reliability used in this paper. For us FTs are static (only AND/OR gates), and each basic event v is assigned a failure probability $\mathrm{p}(v)$. Thus we define:

Definition 3.1. *A* fault tree *(FT) is a tuple* $T = (V, E, \gamma, \mathrm{p})$ *where:*

- (V, E) *is a rooted directed acyclic graph;*
- γ *is a function* $\gamma\colon V \to \{\mathtt{OR}, \mathtt{AND}, \mathtt{BE}\}$ *such that* $\gamma(v) = \mathtt{BE}$ *if and only if v is a leaf;*
- p *is a function* $\mathrm{p}\colon \mathrm{BE}_T \to [0,1]$*, where* $\mathrm{BE}_T = \{v \in V \mid \gamma(v) = \mathtt{BE}\}$.

Note that a FT is not necessarily a tree as gates may share children. The root of T is denoted R_T. For a node v, we let $\mathrm{ch}(v)$ be the set of children of v.

The *structure function* determines, given a gate and a safety event, whether the event succesfully propagates to the gate. Here we model a safety event as the set of BEs happening, which can be encoded as a binary vector $\vec{f} \in \mathbb{B}^{\mathrm{BE}_T}$, where $f_v = 1$ denotes that the BE v occurs in the event. The structure function is then defined as follows.

Definition 3.2. *Let* $T = (V, E, \gamma, \mathrm{p})$ *be a FT.*

1. *A* safety event *is an element of* $\mathbb{B}^{\mathrm{BE}_T}$.
2. *The* structure function *of T is the function* $\mathrm{S}_T : V \times \mathbb{B}^{\mathrm{BE}_T} \to \mathbb{B}$ *defined recursively by*

$$\mathrm{S}_T(v, \vec{f}) = \begin{cases} f_v, & \text{if } \gamma(v) = \mathtt{BE}, \\ \bigvee_{w \in \mathrm{ch}(v)} \mathrm{S}_T(w, \vec{f}), & \text{if } \gamma(v) = \mathtt{OR}, \\ \bigwedge_{w \in \mathrm{ch}(v)} \mathrm{S}_T(w, \vec{f}), & \text{if } \gamma(v) = \mathtt{AND}. \end{cases}$$

3. *A safety event $\vec{f}$ such that $\mathrm{S}_T(\mathrm{R}_T,\vec{f}) = 1$ is called a* cut set*; the set of all cut sets is denoted* CS_T.

Quantitative analysis of a FT is typically done via its *unreliability*, i.e., the probability of a cut set occurring, where each BE v has probability $\mathrm{p}(v)$ of happening. The BE failure probabilities are considered to be independent. The reasoning behind this is that when they are not independent, this is due to some common cause; this common cause should then be explicitely modeled in the FT framework, by replacing the two non-independent BEs with sub-FTs that share common nodes (Pandey, 2005).

Definition 3.3. *Let $T = (V,E,\gamma,\mathrm{p})$ be a FT. Let $\vec{F} \in \mathbb{B}^{\mathrm{BE}_T}$ be the random variable defined by $\mathbb{P}(F_v = 1) = \mathrm{p}(v)$ for all $v \in \mathrm{BE}_T$, and all these events are independent. Then the* unreliability *of T is defined as*

$$\begin{aligned} U(T) &= \mathbb{P}(\vec{F} \in \mathrm{CS}_T) \\ &= \sum_{\vec{f}\in\mathrm{CS}_T} \prod_{v:\, f_v=1} \mathrm{p}(v) \prod_{v:\, f_v=0} (1-\mathrm{p}(v)). \end{aligned}$$

Example 3.4. Consider the FT T from Fig. 1. Abbreviating BE names, assume $\mathrm{p}(\mathsf{rrf}) = \mathrm{p}(\mathsf{lrf}) = 0.4$ and $\mathrm{p}(\mathsf{nf}) = 0.3$. Furthermore, write $\vec{f} \in \mathbb{B}^{\mathrm{BE}_T}$ as $f_{\mathsf{rrf}} f_{\mathsf{nf}} f_{\mathsf{lrf}}$. Then $\mathrm{CS}_T = \{010, 011, 101, 110, 111\}$, so

$$\begin{aligned} U(T) = {} & 0.6\cdot 0.3\cdot 0.6 \\ & +0.6\cdot 0.3\cdot 0.4 \\ & +0.4\cdot 0.7\cdot 0.4 \\ & +0.4\cdot 0.3\cdot 0.6 \\ & +0.4\cdot 0.3\cdot 0.4 = 0.412. \end{aligned}$$

The unreliability $U(T)$ represents the probability of failure of the system modeled by the fault tree and is crucial to providing safety and availability guarantees. The expression in Definition 3.3 becomes too large to handle for large FTs very quickly; thus it is important to find efficient solutions to the following problem.

Problem 3.5. *Given a FT T, calculate $U(T)$.*

Unfortunately, we show that this problem is NP-hard. The reason for this is that with appropriately chosen probabilities, one can find from $U(T)$ a minimal element of CS_T, and finding such a so-called *minimal cut set* is known to be NP-hard (Rauzy, 1993).

Theorem 3.6. *Problem 3.5 is NP-hard.*

3.1 Existing $U(T)$ Algorithms

There are two prominent algorithms for calculating $U(T)$. The first one calculates, for each node v, the probability $g_v = \mathbb{P}(\mathrm{S}_T(v,\vec{F}) = 1)$ bottom-up (Ruijters and Stoelinga, 2015). For BEs one has $g_v = \mathrm{p}(v)$. For an AND-gate, one has $g_v = \prod_{w\in\mathrm{ch}(v)} g_w$ as long as the events $\mathrm{S}_T(w,\vec{F}) = 1$ are independent as w ranges over all children of v. This happens when no two children of v have any shared descendants. For OR-gates one likewise has $g_v = 1 - \prod_{w\in\mathrm{ch}(v)} g_w$. This gives rise to a linear-time algorithm that calculates g_v bottom-up. Unfortunately, this algorithm only works for FTs that have a tree structure: as soon as a node has multiple parents, the independence assumption will be violated at some point in the calculation.

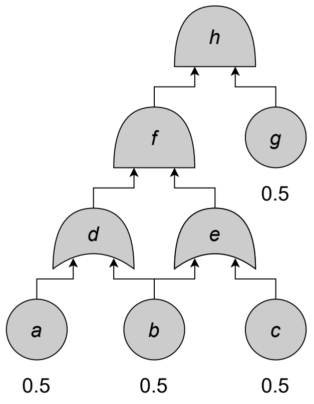


Figure 2: An example FT with failure probabilities.

The second algorithm (Rauzy, 1993) works for general FTs, and works by translating the Boolean function $\mathrm{S}_T(\mathrm{R}_T,-)$ into a *binary decision diagram*, which is a directed acyclic graph, in which the evaluation of the function at a boolean vector is represented by a path through the graph. After the BDD is found, $U(T)$ can be calculated using a bottom-up algorithm on the BDD, whose time complexity is linear in the size of the BDD. Unfortunately, the size of the BDD is worst-case exponential, although this worst case is seldomly attained in practice (Rauzy and Dutuit, 1997; Bobbio et al., 2013). To construct the BDD, one first has to linearly order the variables; finding the order that minimizes BDD size is an NP-hard problem, although heuristics exist (Valiant, 1979; Lê et al., 2014).

4 AN EXAMPLE OF OUR METHOD

Before we dive into the details of our method, we go through an example to showcase how the method works and to motivate the technical sections.

Consider the FT of Fig. 2. In a bottom-up method, we calculate the failure probability g_v of each node v: thus for the BEs we have $g_a = g_b = g_c = g_g = 0.5$. For d, the bottom-up method dictates that we should calculate $g_d = g_a + g_b - g_a g_b$. However, this will cause problems at f, since d and e share b as child, and so their failure probabilities will not be independent. Thus, at d, we modify g_d to 'remember' its dependence on b. We do so by introducing a formal variable L_b representing b's failure probability, yielding $g_d = g_a + \mathsf{L}_b - g_a\mathsf{L}_b = 0.5 + 0.5\mathsf{L}_b$. We also get $g_e = 0.5 + 0.5\mathsf{L}_b$. Note that we only introduce a formal variable for b, and not for a and c, as the latter only have one parent node and thus have no chance of appearing twice in the same calculation.

At f, we calculate $g_f = g_d g_e = 0.25 + 0.5\mathsf{L}_b + 0.25\mathsf{L}_b^2$. Here we introduce the rule $\mathsf{L}_b^2 = \mathsf{L}_b$, so that $g_f = 0.25 + 0.75\mathsf{L}_b$. The idea behind this is that we use multiplication to determine the possibility of two events occurring simultaneously. Since L_b represents $\mathbb{P}(\mathsf{S}_T(b,\vec{F}) = 1)$, the term L_b^2 actually represents $\mathbb{P}(\mathsf{S}_T(b,\vec{F}) = 1 \wedge \mathsf{S}_T(b,\vec{F}) = 1)$. This is equal to $\mathbb{P}(\mathsf{S}_T(b,\vec{F}) = 1)$, so $\mathsf{L}_b^2 = \mathsf{L}_b$.

At this point, the graph structure tells us that b cannot appear twice in the same calculation any more. Thus we can safely substitute $g_b = 0.5$ for L_b in $g_f = 0.25 + 0.75\mathsf{L}_b$, yielding $g_f = 0.625$. Finally, we get $g_h = g_f g_g = 0.3125$.

In the following sections, we introduce two mathematical tools needed to apply this method in greater generality. In Section 5 we review the graph-theoretic notion of dominators, which will tell us for which nodes we need to introduce formal variables and at what point they can be substituted away. In Section 6 we formalize the polynomial algebra in which our arithmetic takes place.

5 PRELIMINARIES I: DOMINATORS

In this section we review the concept of dominators and apply them to FTs. We need this to determine at what point in the bottom-up calculation, outlined in the previous section, we can replace a formal variable L_v with the expression g_v. Informally, a dominator of v is present on all paths from the root to v.

Definition 5.1. (Prosser, 1959) *Let $T = (V, E, \gamma, \mathrm{p})$ be a FT.*

1. *Define a partial order $\preceq$ on V by $x \preceq y$ iff there is a path $y \to x$ in T.*
2. *Given two nodes $v, w \in V$, we say that w* dominates *v if $v \prec w$ and every path $\mathrm{R}_T \to v$ in T contains w.*

The set of dominators of a node is nonempty and has a minimum:

Lemma 5.2. (Lengauer and Tarjan, 1979) *If $v \neq \mathrm{R}_T$, then there is a unique w dominating v such that each w' dominating v satisfies $w \preceq w'$; this w is called the* immediate dominator *of v, denoted $w = \mathrm{id}(v)$.* □

Example 5.3. In Fig. 2, the dominators of a are d, f, and h, and $\mathrm{id}(a) = d$. The dominators of b are f and h, and $\mathrm{id}(b) = f$. Note that d is not a dominator of b, since the path $h \to f \to e \to b$ does not pass through d.

The immediate dominator is interesting to us since, as we will see later, at $\mathrm{id}(v)$ we can replace L_v by g_v. The following result relates the relative position of v and w to that of their immediate dominators.

Lemma 5.4. *If $v \prec w$, then either $\mathrm{id}(v) \preceq w$ or $\mathrm{id}(w) \preceq \mathrm{id}(v)$.*

To use these definitions in an algorithmic context, we will make use of the following result:

Theorem 5.5. (Lengauer and Tarjan, 1979) *Given T, there exists an algorithm of time complexity $O(|E|)$ that finds $\mathrm{id}(v)$ for each $v \in V$.* □

6 PRELIMINARIES II: SQUAREFREE POLYNOMIAL ALGEBRAS

In this second preliminary section, we formally define the algebras in which our calculations take place. These are similar to multivariate polynomial algebras, except in every monomial every variable can have degree at most 1.

Definition 6.1. *Let X be a finite set. We define the* squarefree real polynomial algebra over X *to be the algebra $\mathcal{A}(X)$ consisting of formal sums*

$$\alpha = \sum_{Y \subseteq X} \alpha_Y \prod_{x \in Y} \mathsf{L}_x,$$

where the L_x are formal variables and $\alpha_Y \in \mathbb{R}$. Addition and multiplication are as normal polynomials, except that they are subject to the law $\mathsf{L}_x^2 = \mathsf{L}_x$ for all $x \in X$; that is,

$$(\alpha + \beta)_Y = \alpha_Y + \beta_Y,$$
$$(\alpha \cdot \beta)_Y = \sum_{\substack{Y', Y'' \subseteq X: \\ Y' \cup Y'' = Y}} \alpha_{Y'} \beta_{Y''}.$$

Example 6.2. Let $X = \{x, y\}$. Let $\alpha = 2 + \mathsf{L}_x + \mathsf{L}_y$ and $\beta = \mathsf{L}_x + 3\mathsf{L}_x\mathsf{L}_z$. Coefficientwise, α is described as

$$\alpha_X = \begin{cases} 2, & \text{if } X = \varnothing, \\ 1, & \text{if } X = \{x\} \text{ or } X = \{y\}, \\ 0, & \text{if } X = \{x, y\}. \end{cases}$$

Furthermore, $\alpha + \beta = 2 + 2\mathsf{L}_x + \mathsf{L}_y + 3\mathsf{L}_x\mathsf{L}_z$, and

$\alpha \cdot \beta = \alpha \cdot \mathsf{L}_x + \alpha \cdot (3\mathsf{L}_x\mathsf{L}_z)$

$$= (2\mathsf{L}_x + \mathsf{L}_x + \mathsf{L}_x\mathsf{L}_y) + (6\mathsf{L}_x\mathsf{L}_z + 3\mathsf{L}_x\mathsf{L}_z + 3\mathsf{L}_x\mathsf{L}_y\mathsf{L}_z)$$
$$= 3\mathsf{L}_x + \mathsf{L}_x\mathsf{L}_y + 9\mathsf{L}_x\mathsf{L}_z + 3\mathsf{L}_x\mathsf{L}_y\mathsf{L}_z.$$

Note that if $X \subseteq X'$, an $\alpha \in \mathcal{A}(X)$ can also be considered an element of $\mathcal{A}(X')$, by taking $\alpha_Y = 0$ whenever $Y \not\subseteq X$. For two sets X and Y this also allows us to add and multiply $\alpha \in \mathcal{A}(X)$ and $\beta \in \mathcal{A}(Y)$, by considering both to be elements of $\mathcal{A}(X \cup Y)$. In the rest of this paper we will do this without comment.

Besides multiplication and addition, another important operation that we need is the substitution of a formal variable by a polynomial. This works the same as with regular polynomials.

Definition 6.3. *Let X, Y be finite sets and $x \in X \setminus Y$. Let $\alpha \in \mathcal{A}(X)$ and $\beta \in \mathcal{A}(Y)$. Then the* substitution *$\alpha[\mathsf{L}_x \mapsto \beta]$ is the element of $\mathcal{A}(X \setminus \{x\} \cup Y)$ obtained by replacing all instances of L_x by β; more formally, $\alpha[\mathsf{L}_x \mapsto \beta]$ is expressed as*

$$\beta \cdot \left(\sum_{\substack{Z\subseteq X:\\ x\in Z}} \alpha_Z \prod_{x'\in Z\setminus\{x\}} \mathsf{L}_{x'} \right) + \sum_{\substack{Z\subseteq X:\\ x\notin Z}} \alpha_Z \left(\prod_{x'\in Z} \mathsf{L}_{x'} \right).$$

In terms of coefficients this is expressed as

$$\alpha[\mathsf{L}_x \mapsto \beta]_Z = \alpha_Z + \sum_{\substack{x\in Z'\subseteq X,\\ Z''\subseteq Y:\\ Z'\setminus\{x\}\cup Z''=Z}} \alpha_{Z'}\beta_{Z''}$$

where $\alpha_Z = 0$ if $Z \not\subseteq X$.

Note that in this definition the multiplication and addition are of elements of $\mathcal{A}(X \setminus \{x\} \cup Y)$.

Example 6.4. Continuing Example 6.2, the substitution $\beta[\mathsf{L}_z \mapsto \alpha]$ is equal to

$$\beta[\mathsf{L}_z \mapsto \alpha] = \mathsf{L}_x + 3\mathsf{L}_x \cdot (2 + \mathsf{L}_x + \mathsf{L}_y)$$
$$= 10\mathsf{L}_x + 3\mathsf{L}_x\mathsf{L}_y.$$

In what follows, we will need three results on calculation in $\mathcal{A}(X)$. The first result considerably simplifies the substitution operation.

Lemma 6.5. *Let x, α, β be as in Definition 6.3. Then*

$$\alpha[\mathsf{L}_x \mapsto \beta] = \alpha[\mathsf{L}_x \mapsto 1] \cdot \beta + \alpha[\mathsf{L}_x \mapsto 0] \cdot (1 - \beta).$$

The second result shows how substitution behaves with respect to addition and multiplication.

Lemma 6.6. *Let $\alpha_1, \alpha_2 \in \mathcal{A}(X)$, $\beta \in \mathcal{A}(Y)$, and $x \in X \setminus Y$. Then:*

1. $(\alpha_1 + \alpha_2)[\mathsf{L}_x \mapsto \beta] = \alpha_1[\mathsf{L}_x \mapsto \beta] + \alpha_2[\mathsf{L}_x \mapsto \beta]$.
2. *If $\beta^2 = \beta$, then furthermore $(\alpha_1\alpha_2)[\mathsf{L}_x \mapsto \beta] = \alpha_1[\mathsf{L}_x \mapsto \beta] \cdot \alpha_2[\mathsf{L}_x \mapsto \beta]$.*

The third result states that two substitution operations can be interchanged, as long as one does not substitute a variable present in the other:

Lemma 6.7. *Let $\alpha \in \mathcal{A}(X)$, $\beta_1 \in \mathcal{A}(Y_1)$, $\beta_2 \in \mathcal{A}(Y_2)$, $x_1, x_2 \in X \setminus (Y_1 \cup Y_2)$. If $x_1 \notin Y_2$ and $x_2 \notin Y_1$, then $\alpha[\mathsf{L}_{x_1} \mapsto \beta_1][\mathsf{L}_{x_2} \mapsto \beta_2] = \alpha[\mathsf{L}_{x_2} \mapsto \beta_2][\mathsf{L}_{x_1} \mapsto \beta_1]$.*

When this lemma applies and the order of substitutions does not matter, we will write expressions like $\alpha[\mathsf{L}_{x_1} \mapsto \beta_1, \mathsf{L}_{x_2} \mapsto \beta_2]$, or even $\alpha[\forall i \leq n\colon \mathsf{L}_{x_i} \mapsto \beta_i]$.

Note that as an $\mathbb{R}$-algebra, one may identify $\mathcal{A}(X)$ with K/I, where $K = \mathbb{R}[\mathsf{L}_x\colon x \in X]$ is a free polynomial algebra and I is the ideal generated by the set $\{\mathsf{L}_x^2 - \mathsf{L}_x \mid x \in X\}$. However, the substitution operation does not correspond to a 'natural' operation on on K/I.

6.1 Real-Valued Boolean Functions

We will use the elements of $\mathcal{A}(X)$ is to represent functions $\mathbb{B}^X \to \mathbb{R}$. The following result states that this can be done in a unique way. Since both elements of $\mathcal{A}(X)$ and functions $\mathbb{B}^X \to \mathbb{R}$ can be represented by $2^{|X|}$ real numbers, this should come as no surprise.

Theorem 6.8. *Let X be a finite set, and let $g\colon \mathbb{B}^X \to \mathbb{R}$ be any function. Then there exists a unique $\langle g \rangle \in \mathcal{A}(X)$ such that $g(c) = \langle g \rangle[\forall x \in X\colon \mathsf{L}_x \mapsto c_x]$ for all $\vec{c} \in \mathbb{B}^X$.*

Example 6.9. Let $X = \{x, y\}$, with $\vec{c} \in \mathbb{B}^X$ represented as $c_x c_y$. Consider the function $g\colon \mathbb{B}^X \to \mathbb{R}$ given by

$$g(00) = 3, \qquad g(01) = -2,$$
$$g(10) = 7, \qquad g(11) = 4.$$

Suppose $\langle g \rangle = k_1 + k_2\mathsf{L}_x + k_3\mathsf{L}_y + k_4\mathsf{L}_x\mathsf{L}_y$. Then, for instance,

$$g(10) = \langle g \rangle[\mathsf{L}_x \mapsto 1, \mathsf{L}_y \mapsto 0] = k_1 + k_2.$$

In a similar way we can express all $g(\vec{c})$ as sums of k_i. Thus, to find the k_i, we have to solve

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \\ -2 \\ 4 \end{pmatrix}.$$

Since this matrix is lower triangular with nonzero diagonal entries, it is invertible, so the k_i exist and are unique. In fact, we find $\langle g \rangle = 3 + 4\mathsf{L}_x - 5\mathsf{L}_y + 2\mathsf{L}_x\mathsf{L}_y$.

7 THE ALGORITHM

Using the notation of the previous two sections, we can now state our algorithm for calculating unreliability. It is presented in Algorithm 1. The algorithm works bottom-up, assigning to each node v a formal

```
input : A FT $T = (V, E, \gamma, \mathrm{p})$
output: $U(T)$
1  ToDo $\leftarrow V$;
2  while ToDo $\neq \varnothing$ do
3      Pick $v \in$ ToDo minimal w.r.t. $\preceq$;
4      ToDo $\leftarrow$ ToDo $\setminus \{v\}$;
5      if $\gamma(v) = \mathtt{BE}$ then
6          $g_v \leftarrow \mathrm{p}(v)$;
7      else
8          if $\gamma(v) = \mathtt{OR}$ then
9              $g_v \leftarrow 1 - \prod_{w \in \mathrm{ch}(v)} (1 - \mathsf{L}_w)$;
10         else
11             $g_v \leftarrow \prod_{w \in \mathrm{ch}(v)} \mathsf{L}_w$;
12         end
13         ToDo$_v \leftarrow \{w \in V \mid \mathrm{id}(w) = v\}$;
14         while ToDo$_v \neq \varnothing$ do
15             Pick $w \in$ ToDo$_v$ maximal w.r.t. $\preceq$;
16             ToDo$_v \leftarrow$ ToDo$_v \setminus \{w\}$;
17             $g_v \leftarrow g_v[\mathsf{L}_w \mapsto g_w]$;
18         end
19     end
20 end
21 return $g_{\mathrm{R}_T}$
```

Algorithm 1: The algorithm $\mathtt{SFPA}(T)$.

expression $g_v \in \mathcal{A}(\{w \in V \mid w \prec v\})$ representing the failure probability of v; the formal variables L_w present in g_v represent nodes with multiple paths from the root, which we will also encounter further in the calculation.

The algorithm works as follows: working bottom-up (lines 1–4), the algorithm first assigns a g_v of the most basic form to v (lines 5–12): for a BE this is simply its failure probability $\mathrm{p}(v)$, while for OR- and AND-gates it is the expression for the failure probability in terms of the formal variables L_w, where w ranges over $\mathrm{ch}(v)$. After obtaining this expression of g_v, the algorithm then substitutes away all formal variables that we will not encounter later in the computation (lines 13–18). These are precisely the L_w for which $\mathrm{id}(w) = v$, as for these w this is the point where we will not encounter other copies of L_w anymore. We replace each L_w with the associated expression g_w; we start with the w closest to v, as these g_w may contain other $\mathsf{L}_{w'}$ that also need to be substituted away. Finally, we return g_{R_T} (line 21). At this point all formal variables have been substituted away, so $g_{\mathrm{R}_T} \in \mathbb{R}$. Note that 'under the hood' we have determined $\mathrm{id}(v)$ for each v, which can be done in linear time by Theorem 5.5.

The main theoretical result of this paper is the validity of Alg. 1.

Theorem 7.1. *Let T be a FT. Then* $\mathtt{SFPA}(T) = U(T)$.

We will prove this theorem in Section 9. First, we introduce a slight extension to the FT formalism.

8 PARTIALLY CONTROLLABLE FAULT TREES

In this section, we slightly extend the FT formalism in a manner necessary for the proof of Theorem 7.1. The resulting objects, *partially controllable fault trees* (PCFTs), are just like regular FTs, except that certain BEs are labelled *controllable BEs*; these do not have a fixed failure probability but instead can be set to $\mathtt{0}$ or $\mathtt{1}$ at will. We emphasize that the concept of PCFTs does not correspond to an engineering reality, but is a mathematical construct needed for the proof of Theorem 7.1.

Definition 8.1. *An* partially controllable fault tree *(PCFT) is a tuple $T = (V, E, \gamma, \mathrm{p})$ where:*

- *(V, E) is a rooted directed acyclic graph;*
- *γ is a function $\gamma\colon V \to \{\mathtt{OR}, \mathtt{AND}, \mathtt{BE}, \mathtt{CBE}\}$ such that $\gamma(v) \in \{\mathtt{BE}, \mathtt{CBE}\}$ if and only if v is a leaf;*
- *p is a function $\mathrm{p}\colon \mathrm{BE}_T \to [0,1]$, where $\mathrm{BE}_T = \{v \in V \mid \gamma(v) = \mathtt{BE}\}$.*

Similar to BE_T we define $\mathrm{CBE}_T = \{v \in V \mid \gamma(v) = \mathtt{CBE}\}$. Since the failure of CBEs is not probabilistic, one can only speak of the failure probability of T once one has set the states of the CBEs. Therefore, $U(T)$ is not a fixed probability, but a function $\mathbb{B}^{\mathrm{CBE}_T} \to [0,1]$.

Definition 8.2. *1. The structure function of T is a map $V \times \mathbb{B}^{\mathrm{BE}_T} \times \mathbb{B}^{\mathrm{CBE}_T} \to \mathbb{B}$ defined by*

$$\mathrm{S}_T(v, \vec{f}, \vec{c}) = \begin{cases} f_v, & \text{if } \gamma(v) = \mathtt{BE}, \\ c_v, & \text{if } \gamma(v) = \mathtt{CBE}, \\ \bigvee_{w \in \mathrm{ch}(v)} \mathrm{S}_T(w, \vec{f}, \vec{c}), & \text{if } \gamma(v) = \mathtt{OR}, \\ \bigwedge_{w \in \mathrm{ch}(v)} \mathrm{S}_T(w, \vec{f}, \vec{c}), & \text{if } \gamma(v) = \mathtt{AND}. \end{cases}$$

2. Let $\vec{F} \in \mathbb{B}^{\mathrm{BE}_T}$ be a random variable so that F_v is Bernoulli distributed with $\mathbb{P}(F_v = \mathtt{1}) = \mathrm{p}(v)$ for each $v \in \mathrm{BE}_T$, and all F_v are independent. Then the unreliability *of T is the function $U(T)\colon \mathbb{B}^{\mathrm{CBE}_T} \to [0,1]$ given by*

$$U(T)(\vec{c}) = \mathbb{P}(\mathrm{S}_T(\mathrm{R}_T, \vec{F}, \vec{c}) = \mathtt{1}).$$

In light of Theorem 6.8, the function $U(T)\colon \mathbb{B}^{\mathrm{CBE}_T} \to \mathbb{B}$ is described by its associated polynomial

$$\langle U(T) \rangle \in \mathcal{A}(\mathrm{CBE}_T).$$

Example 8.3. Consider the PCFT $\texttt{OR}(a,b)$, where $\gamma(a) = \texttt{BE}$ and $\gamma(b) = \texttt{CBE}$ (see Fig. 3) and $\mathrm{p}(a) = 0.4$. Then $\mathbb{B}^{\mathrm{BE}_T} \cong \mathbb{B}^{\mathrm{CBE}_T} \cong \mathbb{B}$, and $\mathrm{S}_T(\mathrm{R}_T, f, c) = \mathtt{1}$ if and only if at least one of f_a, c_b equals $\mathtt{1}$. Since $\mathbb{P}(F_a = 1) = \mathrm{p}(a) = 0.4$, it follows that

$$U(T)(c) = \begin{cases} 0.4, & \text{if } c_b = \mathtt{0}, \\ 1, & \text{if } c_b = \mathtt{1}. \end{cases}$$

As a polynomial this is $\langle U(T)\rangle = 0.4 + 0.6\mathsf{L}_b$.

8.1 Quasimodular Composition

Now that we have expressed a PCFT T as a polynomial $\langle U(T)\rangle$, the next step is to relate substitution operations on such polynomials to graph-theoretic operations on PCFTs. The key concept on the PCFT side is *quasimodular composition*, which is defined as follows.

Definition 8.4. *Let* $T = (V, E, \gamma, \mathrm{p})$ *and* $T' = (V', E', \gamma', \mathrm{p}')$ *where* V *and* V' *are not necessarily disjoint, such that* $E, \gamma, \mathrm{p}, \mathrm{ch}$ *coincide with* $E', \gamma', \mathrm{p}', \mathrm{ch}$ *on* $V \cap V'$. *Let* $v \in \mathrm{CBE}_T \setminus V'$, *and assume that* $\mathrm{BE}_T \cap \mathrm{BE}_{T'} = \varnothing$ *(see Fig. 4). Then the* quasimodular composition $T[v \mapsto T']$ *of* T *and* T' *in* v *is the PCFT obtained by replacing* v *in* T *by the entire PCFT* T', *rerouting all edges originally to* v *to* $\mathrm{R}_{T'}$ *instead.*

The concept of quasimodular composition of PCFTs is closely related to modular composition of FTs (Rauzy and Dutuit, 1997). The difference is that in modular composition T and T' may not share any nodes, while in quasimodular composition they may share CBEs, as well as any internal nodes; however, due to the condition that these internal nodes must have the same children in T and T', any shared internal nodes may not have any BE descendants.

The following result states that substitution on the polynomial level precisely corresponds to quasimodular composition on the PCFT level; it is the key ingredient to the proof of Theorem 7.1.

Theorem 8.5. *Let* T, T', v *be as in Definition 8.4 and let* $T'' = T[v \mapsto T']$ *be their quasimodular composition. Then* $\mathrm{CBE}_{T''} = \mathrm{CBE}_T \setminus \{v\} \cup \mathrm{CBE}_{T'}$, *and as elements of* $\mathcal{A}(\mathrm{CBE}_{T''})$ *one has*

$$\langle U(T'')\rangle = \langle U(T)\rangle[\mathsf{L}_v \mapsto \langle U(T')\rangle].$$

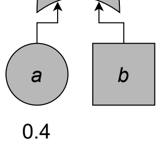
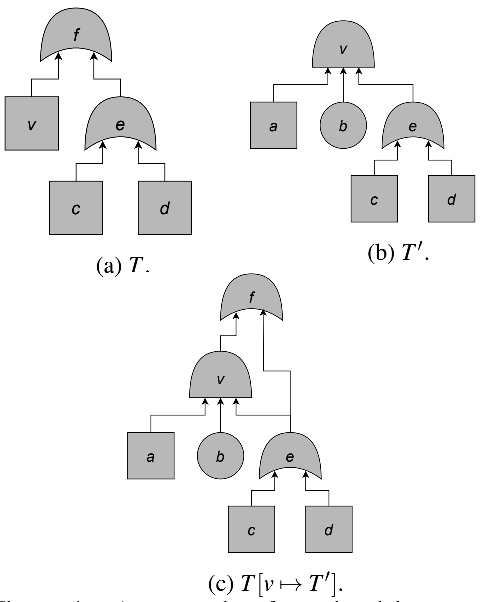


Figure 3: The PCFT of Example 8.3.



(c) $T[v \mapsto T']$.
Figure 4: An example of quasimodular composition. Square nodes are CBEs.

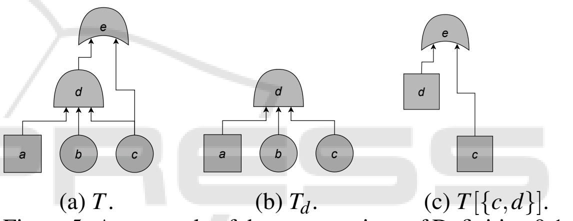


Figure 5: An example of the constructions of Definition 9.1. A PCFT T is depicted in (a) (square nodes are CBEs). The sub-FT T_d with root d is depicted in (b). The FT $T[\{c,d\}]$ obtained by turning c, d into CBEs is depicted in (c); note that this FT is also equal to $T[\{b,c,d\}]$, $T[\{a,c,d\}]$ and $T[\{a,b,c,d\}]$.

This theorem is best read 'in reverse': given a large PCFT T'', one can calculate $U(T'')$ by finding a *quasimodule* T' and its remainder T, and combining $U(T)$ and $U(T')$. In this sense, this theorem is analogous to *modular decomposition* of FTs (Rauzy and Dutuit, 1997), in which a FT's unreliability is expressed in terms of that of its *modules*. Again, the key difference is that we allow not just modular decomposition, but also quasimodular decomposition.

9 SKETCHED PROOF OF CORRECTNESS

In this section, we sketch the proof of Theorem 7.1; a full proof is presented in the appendix. Before we outline the proof, we first define two ways to construct new (PC)FTs from a FT.

Definition 9.1. *Let* $T = (V, E, \gamma, \mathrm{p})$ *be a FT.*

1. *Let $v \in V$. Then $T_v = (V_v, E_v, \gamma_v, \mathrm{p}_v)$ is the FT consisting of the descendants of v, with v as a root.*
2. *Let $I \subseteq V$. Then $T[I]$ is the PCFT obtained from T via the following procedure:*
   - *For each $v \in I$, set $\gamma(v) = \mathtt{CBE}$;*
   - *For each $v \in I$, remove all outgoing edges;*
   - *Then $T[I]$ is the PCFT consisting of all nodes reachable from the root.*

These constructions are depicted in Figure 5.

For a node v, let $g_{v,\infty}$ be the value of g_v at the end of the loop in lines 14–18 of Algorithm 1; this is the value g_v has when the algorithm ends, and which will be used to substitute L_v in line 17. Then Theorem 7.1 follows from the following result:

Theorem 9.2. *Let $v \in V$, and define*

$$I_v = \{w \in V \mid w \prec v \prec \mathrm{id}(w)\}.$$

Then $g_{v,\infty} = \langle U(T_v[I_v])\rangle$.

Theorem 7.1 is just the special case $v = \mathrm{R}_T$, as $T_{\mathrm{R}_T} = T$ and $I_{\mathrm{R}_T} = \varnothing$. The proof can be sketched as follows:

This is proven by induction on T. For BEs this is immediate. If $\gamma(v) = \mathtt{AND}$, then g_v is initialized as $\prod_{w\in \mathrm{ch}(v)} \mathsf{L}_w$. Then, for each w picked in line 15 of Algorithm 1, a L_w is substituted by its corresponding polynomial $g_w = g_{w,\infty}$. By the induction hypothesis, $g_{w,\infty}$ corresponds to the unreliability function of a certain PCFT, and by Theorem 8.5, this substitution operation corresponds to the composition of PCFTs. By keeping track of the form of the resulting PCFT, one shows that the PCFT one ends up with is exactly $T_v[I_v]$, showing the result for v. The case $\gamma(v) = \mathtt{OR}$ is, of course, completely analogous. By induction, this proves Theorem 9.2, and by consequence Theorem 7.1.

10 COMPLEXITY

The complexity of Alg. 1 can be bound in terms of graph parameters of the DAG T. To do so, we first note that we can slightly rephrase the algorithm as follows. If w is a child of v and w has only one parent, then w is a maximal element of ToDo_v. As such, in line 15 these w will be picked first. Therefore, we may as well do this replacement in lines 9 and 11 directly. This leads to Alg. 2, which has the same functionality as Alg. 1 and therefore calculates $U(T)$ correctly. Note that the condition that w has only one parent is equivalent to $\mathrm{id}(w) = v$, which leads to our definition of S_v in line 8.

Like the standard algorithm for reliability analysis for treelike FTs (Ruijters and Stoelinga, 2015), Alg. 2

```
   input : A FT $T = (V, E, \gamma, \mathrm{p})$
   output: $U(T)$
1  ToDo ← $V$;
2  while ToDo $\neq \varnothing$ do
3      Pick $v \in$ ToDo minimal w.r.t. $\preceq$;
4      ToDo ← ToDo $\setminus \{v\}$;
5      if $\gamma(v) = \mathtt{BE}$ then
6          $g_v \leftarrow \mathrm{p}(v)$;
7      else
8          $S_v = \{w \in \mathrm{ch}(v) \mid \mathrm{id}(w) = v\}$;
9          if $\gamma(v) = \mathtt{OR}$ then
10             $p_1 \leftarrow \prod_{w\in S_v}(1 - g_w)$;
11             $p_2 \leftarrow \prod_{w\in \mathrm{ch}(v)\setminus S_v}(1 - \mathsf{L}_w)$;
12             $g_v \leftarrow 1 - p_1 p_2$;
13         else
14             $p_1 \leftarrow \prod_{w\in S_v} g_w$;
15             $p_2 \leftarrow \prod_{w\in \mathrm{ch}(v)\setminus S_v} \mathsf{L}_w$;
16             $g_v \leftarrow p_1 p_2$;
17         end
18         ToDo$_v$ ← $\{w \in V \setminus \mathrm{ch}(v) \mid \mathrm{id}(w) = v\}$;
19         while ToDo$_v \neq \varnothing$ do
20             Pick $w \in$ ToDo$_v$ maximal w.r.t. $\preceq$;
21             ToDo$_v$ ← ToDo$_v \setminus \{w\}$;
22             $g_v \leftarrow g_v[\mathsf{L}_w \mapsto g_w]$;
23         end
24     end
25 end
26 return $g_{\mathrm{R}_T}$
```

Algorithm 2: The algorithm $\mathtt{SFPA2}(T)$.

works bottom-up. However, it is more complicated due to the fact that our main objects of interest are squarefree polynomials rather than real numbers, and operating on these induces a larger complexity. To describe this complexity, we introduce the following notation:

$$X = \{v \in V \mid v \text{ has multiple parents}\}.$$

Since in Alg. 2 line 8, the set S_v consists of all children of v with a single parent, the only L_x that are introduced satisfy $x \in X$. Thus each g_v is an element of $\mathcal{A}(X)$, and as such has at most $2^{|X|}$ terms. Multiplying two such polynomials has complexity $O(4^{|X|})$, and since substitution is just multiplication by Lemma 6.5, substitution has the same complexity. Next, we count the number of multiplications and substitutions. The FT T has $|X|$ nodes with more than 1 parent and 1 node with 0 parents, so in total $|V| - |X| - 1$ nodes have exactly one parent and are used in multiplications in lines 10 and 14 of Alg. 2; in particular, there are at most $|V|$ multiplications. Furthermore, for each v one has $|\mathsf{ToDo}_v| \leq |X|$, hence at each v there are at most $|X|$ substitutions in line 22. We conclude:

Table 1: Summary of the results: the minimum, median and maximum of the computation times (in seconds) of the two algorithms are displayed. *Timeout at 60 seconds, attained 3 times.

		SFPA	Storm
Benchmark 1	minimum	0.60	0.89
	median	3.51	1.81
	maximum	60*	7.91
Benchmark 2	minimum	0.30	0.41
	median	0.68	1.42
	maximum	6.73	47.29

Theorem 10.1. *Let X be as above. Then Alg. 2 has time complexity $O(|V|(|X|+1)4^{|X|})$.*

If $X = \varnothing$, then T is treelike. In this case, Theorem 10.1 tells us that Alg. 2 has time complexity $O(|V|)$. Indeed, in this case $S_v = \mathrm{ch}(v)$ for all v, and each g_v is a real number. Hence Alg. 2 reduces to the standard bottom-up algorithm for treelike FTs, which is known to have linear time complexity. In fact, Theorem 10.1 generalizes this result: it shows that for bounded $|X|$, time complexity of Alg. 2 is linear. This makes Alg. 2 a useful tool if the non-tree topology of an FT is concentrated in only a few nodes.

The complexity bound of Theorem 10.1 can be sharpened by realizing that a variable L_w only occurs in the computation of g_v if $w \prec v \preceq \mathrm{id}(w)$. Using this as a bound we get the following result:

Theorem 10.2. *Define*

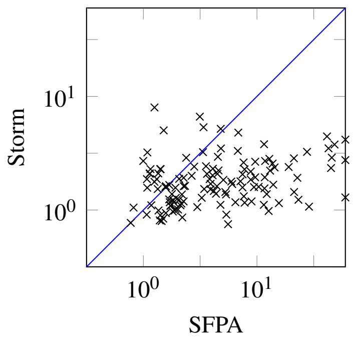
$$c = \max_{v \in V} |\{w \in X \mid w \prec v \preceq \mathrm{id}(w)\}|.$$

Then Alg. 2 has time complexity $O(|V|(c+1)4^c)$.

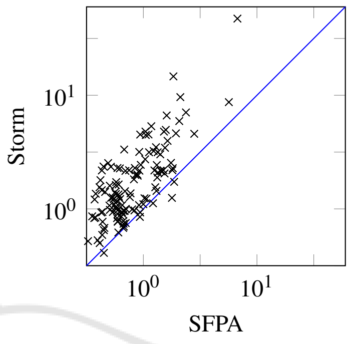
As far as I am aware, a comparable analysis that bounds the time complexity of BDD-based methods in terms of multiparent nodes does not exist in the literature. Such an analysis can be complicated by variable ordering heuristics for the BDD construction, and is beyond the scope of this paper. Thus the existence of a provable complexity bound in terms of the number of multiparent nodes is a major advantage of the SFPA method.

11 EXPERIMENTS

We perform experiments to test `SFPA`'s performance, as implemented in Python. All experiments are performed on a Ubuntu virtual machine with 6 cores and 8GB ram, running on a PC with an Intel Core i7-10750HQ 2.8GHz processor and 16GB memory. We compare the performance to that of Storm-dft, a state-of-the-art model checker that calculates FT unreliability via a BDD-based approach with modularization



(a) Unreliability on benchmark set 1.



(b) Unreliability on benchmark set 2.

Figure 6: Timing comparison of SFPA and Storm for calculating unreliability. Times are in seconds, timeout at 60s.

(Basgöze et al., 2022b). We compare performance on two benchmark sets of FTs:

1. A collection of 128 randomly generated FTs used as a benchmark set in (Basgöze et al., 2022a). These FTs have, on average, 89.3 nodes, of which 10.3 have multiple parents.
2. A new randomly generated collection of 128 FTs, created using SCRAM (Rakhimov, 2019). These FTs have, on average, 123.8 nodes, 4.1 of which have multiple parents.

The second benchmark set was created in order to validate the theoretical results of Section 10, where it was shown that the complexity of SFPA depends on the number of nodes with multiple parents. We compute both the unreliability of each FT and measure the time of both computations (timeout: 60 seconds). The results are given in Table 1 and Figure 6. As one can see, Storm largely outperforms SFPA on benchmark set 1, with lower computation times on 76% of the unreliability calculations. On benchmark set 2, SFPA fares considerably better, outperforming Storm on 95% of all FTs for unreliability calculation: on average SFPA takes only 54% of the computation time of Storm. The fact that SFPA is more efficient on this benchmark set can be understood from Theorem 10.1, which shows that computational complexity of SFPA is low when the number of multiparent nodes is low. By contrast, for a BDD-based approach the presence of *any* multiparent nodes means one cannot use the bottom-up algorithm and has to rely on creating the BDD, which is usually slower than the bottom-up approach. Furthermore, modularization may only be of limited use depending on the position of the multiparent nodes.

Overall, we can conclude that for calculating unreliability SFPA is competitive with the state-of-the-art, and is significantly faster on an FT benchmark set with fewer multiparent nodes.

12 CONCLUSION

In this paper, we have introduced SFPA, a novel algorithm for calculating fault tree unreliability based on squarefree polynomial algebras. We have proven its validity and given complexity bounds in terms of the number of multiparent nodes. Experiments show that it is significantly faster than the state of the art on FTs with few multiparent nodes.

There are several directions for future work. First, our proof-of-concept Python implementation of SFPA can undoubtedly be improved, leading to faster computation. Such improvements can be done on the theoretical side as well. For example, one could introduce a new formal variable U_v for $1-\mathsf{L}_v$; this would decrease the number of terms in the expression of g_v when v is an $\texttt{OR}$-gate from $2^{|\mathrm{ch}(v)|}-1$ to 2, hopefully leading to faster computation. In this case, new computation rules such as $\mathsf{L}_v\mathsf{U}_v = 1$ need to be introduced.

Second, our experimental results show that a BDD-based method works best for FTs with more multiparent nodes, while SFPA works best for FTs with fewer multiparent nodes. It would be interesting to see a more extensive experimental evaluation that investigates what the break-even point is. Such an experimental evaluation can be augmented by incorporating real-world case studies, to test the effectiveness of SFPA in practice.

On the other hand, it would be interesting to see to what extent SFPA-like methods can be applied to other problems in FT analysis, such as the analysis of dynamic FTs, which also consider time-dependent gates and behaviour. A good candidate is the analysis of attack trees (ATs), the security counterpart of FTs. Quantitative analysis of (non-dynamic) ATs is also done using BDDs (Lopuhaä-Zwakenberg et al., 2022), which has the same issues as BDD-based FT analysis. We expect that SFPA-like methods can be extended to ATs as well.

ACKNOWLEDGEMENTS

This research has been partially funded by ERC Consolidator grant 864075 CAESAR and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

REFERENCES

Basgöze, D., Volk, M., Katoen, J.-P., Khan, S., and Stoelinga, M. (2022a). Artifact for "BDDs Strike Back - Efficient Analysis of Static and Dynamic Fault Trees".

Basgöze, D., Volk, M., Katoen, J.-P., Khan, S., and Stoelinga, M. (2022b). Bdds strike back: efficient analysis of static and dynamic fault trees. In *NASA Formal Methods Symposium*, pages 713–732. Springer.

Bobbio, A., Egidi, L., and Terruggia, R. (2013). A methodology for qualitative/quantitative analysis of weighted attack trees. *IFAC Proceedings Volumes*, 46(22):133–138.

Bouissou, M., Bruyere, F., and Rauzy, A. (1997). Bdd based fault-tree processing: a comparison of variable ordering heuristics. In *Proceedings of European Safety and Reliability Association Conference, ESREL'97*.

IsoTree (2023). FaultTree+. available online at https://www.isograph.com/software/reliability-workbench/fault-tree-analysis-software/.

Lê, M., Weidendorfer, J., and Walter, M. (2014). A novel variable ordering heuristic for bdd-based k-terminal reliability. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 527–537. IEEE.

Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141.

Lopuhaä-Zwakenberg, M., Budde, C. E., and Stoelinga, M. (2022). Efficient and generic algorithms for quantitative attack tree analysis. *IEEE Transactions on Dependable and Secure Computing*.

Lopuhaä-Zwakenberg, M. (2023a). Fault tree reliability analysis via squarefree polynomials.

Lopuhaä-Zwakenberg, M. (2023b). Fault tree reliability analysis via squarefree polynomials (full version). available online at https://arxiv.org/abs/2312.05836.

Pandey, M. (2005). Fault tree analysis. *Lecture notes, University of Waterloo, Waterloo*.

Prosser, R. T. (1959). Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138.

Rakhimov, O. (2019). Scram. available online at https://github.com/rakhimov/scram.

Rauzy, A. (1993). New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211.

Rauzy, A. and Dutuit, Y. (1997). Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering & System Safety*, 58(2):127–144.

Reliotech (2023). TopEvent FTA. available online at https://www.fault-tree-analysis.com/free-fault-tree-analysis-software.

Ruijters, E., Budde, C. E., Nakhaee, M. C., Stoelinga, M. I., Bucur, D., Hiemstra, D., and Schivo, S. (2019). Ffort: a benchmark suite for fault tree analysis.

Ruijters, E. and Stoelinga, M. (2015). Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer science review*, 15:29–62.

Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *siam Journal on Computing*, 8(3):410–421.

Watson, H. A. (1961). Launch control safety study. *Bell labs*.