Fault Trees from Data: Efficient Learning with an Evolutionary Algorithm

Alexis Linard[1,3(✉)], Doina Bucur[2], and Mariëlle Stoelinga[1,2]

[1] Institute for Computing and Information Science, Radboud University, Nijmegen, The Netherlands
a.linard@cs.ru.nl
[2] University of Twente, Enschede, The Netherlands
{d.bucur,m.i.a.stoelinga}@utwente.nl
[3] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. Cyber-physical systems come with increasingly complex architectures and failure modes, which complicates the task of obtaining accurate system reliability models. At the same time, with the emergence of the (industrial) Internet-of-Things, systems are more and more often being monitored via advanced sensor systems. These sensors produce large amounts of data about the components' failure behaviour, and can, therefore, be fruitfully exploited to learn reliability models automatically. This paper presents an effective algorithm for learning a prominent class of reliability models, namely fault trees, from observational data. Our algorithm is evolutionary in nature; i.e., is an iterative, population-based, randomized search method among fault-tree structures that are increasingly more consistent with the observational data. We have evaluated our method on a large number of case studies, both on synthetic data, and industrial data. Our experiments show that our algorithm outperforms other methods and provides near-optimal results.

Keywords: Fault tree induction · Safety-critical systems · Cyber-physical systems · Evolutionary algorithm

1 Introduction

Reliability engineering is an important field that provides methods, tools and techniques to evaluate and mitigate the risks related to complex systems such as drones, self-driving cars, production plants, etc. Fault tree analysis is one of the most prominent technique in this field. It is widely deployed in the automotive, aerospace and nuclear industry, by companies and institutions like NASA, Ford, Honeywell, Siemens, the FAA, and many others.

This research is supported by the Dutch Technology Foundation (STW) under the Robust CPS program (project 12693), the EU project SUCCESS, the Smart Industries program (project SEQUOIA 15474), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

N. Guan et al. (Eds.): SETTA 2019, LNCS 11951, pp. 19–37, 2019.
https://doi.org/10.1007/978-3-030-35540-1_2

Fault trees [32] (FTs) belong to analytical techniques for safety, security, and dependability. They are graphical models that represent how component failures arise and propagate through the system, leading to system-level failures. Component failures are modelled in the leaves of the tree as *basic events*. Fault tree *gates* model how combinations of basic events lead to a system failure, represented by the top event in the FT. The analysis of such FTs [29] is multifold: they can be used to compute dependability metrics such as system reliability and availability; understand how systems can fail; identify the best ways to reduce the risk of system failure, etc. A key bottleneck in fault tree analysis is, however, the effort needed to construct a faithful fault tree model. FTs are usually built manually by domain experts. Given the complexity of today's systems, industrial FTs often contain thousands of gates. Hence, their construction is a very intricate task, and also error-prone, since their soundness and completeness largely depends on domain expertise. With the emergence of the industrial Internet-of-Things, Cyber-physical systems are more and more equipped with smart sensor systems, monitoring whether a system component is in a failed state or not. Even though such a monitoring system is often designed to detect failures during operations, their data can be very fruitfully deployed to learn reliability models. Such data can be crucial for the engineers to build an FT [14]. Recent work focused on learning FTs from observational data, identifying causalities from data [25].

In this paper, we focus on FT generation from data, using an evolutionary algorithm (EA). EAs approximate stochastic learning by mimicking biological evolution, and have been successfully applied to a wide plethora of applications; examples include the scheduling of flexible manufacturing systems [11], automata learning [10], induction of Boolean functions [28], and many more. In our case, each stage of the EA keeps a population of candidate FTs. New fault trees are generated by mimicking biological evolution. That is, new FTs are created by reproduction (e.g., adding or deleting FT gates), crossover (e.g. swapping FT branches), and mutation (e.g. changing an AND gate into an OR gate). In total, we have identified seven (parametric) generation rules, which are equally applied. Finally, we select the new population by only keeping those FTs with the best *fitness*, i.e. FTs that best fit to the observational data. We have experimentally verified the applicability of our algorithm, on synthetic data, an industrial case study, and a benchmark of FTs previously studied in the literature. Our experiments show that the algorithm is fast and accurate ($>99\%$). Further, we have investigated the robustness of our method to noisy data. Here we found that our EA handles noisy records. We also developed a variation of our EA in order to take expert knowledge into account. When domain experts partially know the structure of the FT, then the task is reduced to evolve *sub*-Fault Trees, given the known *skeleton* of the FT.

Being a first step, our algorithm focuses on *static* fault trees, featuring only Boolean gates. An important topic for future work is the extension to dynamic fault trees. These come with additional gates, catering for common dependability patterns like spare management and functional dependencies. Static fault trees, however, have appeal as relatively simple yet powerful formalism and are

often used in practice. Furthermore, dynamic fault trees strongly depend on the temporal order in which failures occur, and their learning will, therefore, require more complex data, such as time series.

This paper is organized as follows. Sections 2 and 3 review related work on learning FTs from data as well as preliminary definitions. We present then in Sect. 4 our technique to infer an FT using an EA. In Sect. 5, the variation of our EA that takes expert knowledge into account. In Sect. 6, we show the results we achieved. Finally, we discuss and conclude about further research. We refer to the full version of our paper for missing technical details.[1]

2 Related Work

Related work on learning fault trees spans three areas of research: the synthesis of fault trees from other graphical models of the system under study; recent work on the generation of fault trees from observational data describing the system; and, since fault trees are in essence Boolean functions, literature on learning Boolean functions from observational data.

Model-Based Synthesis. While state-of-the-art fault tree design is often performed manually by domain specialists [15], several methods have been proposed to synthesize FTs automatically from other models of the system [3,30]. Thus, these methods require the pre-construction of a system model in a suitable model description language, which varies with each method for FT synthesis. For example, the HiP-HOPS framework [27] synthesizes an FT from a system model describing transactions among the system components, annotated with failure information. Similar synthesis methods were developed from the AltaRica system description language, which models the causal relations between system variables and events using transitions [20]. Specific system control models in the form of directed graphs have also been shown to suit the synthesis of fault trees [1,12], as well as Go models [33]. Furthermore, system models described in the model language NuSMV also enable the synthesis of FTs. A limitation of this method is, however, the fact that the resulting FTs show the relation between top events and basic events, but do not show how failure propagates in the system via system components [4]. Static FTs can also be synthesized from models in the Architectural Analysis and Design Language AADL [21].

As a special case, FT generation has been attempted so that the learning method includes explicit reasoning about the causal relations between events in the system. For this type of FT generation, [18] requires a probabilistic system model, from which a model-checking step obtains a set of probabilistic counterexamples. When the system is concurrent, the order of events in these counterexamples does not necessarily signify causality, so logical combinations of events are separately validated for causality. Similarly, in [22] a cause-effect graph (and from that, an FT) is extracted by model checking a process already modelled by a finite-state machine, against safety and liveness requirements, using failure

[1] See http://arxiv.org/abs/1909.06258.

injection. Since model-based FT learning requires prior modelling of the system under study, these methods do not *adapt* well in applications where the systems evolve and thus need to be remodelled, e.g., components are replaced, or the interactions between components change, thus changing the failure modes and their probability of occurrence.

Learning Causal Models from Data. Supervised *automated learning* of dependability models using *system data*, unlike the model-based methods described above, will adapt to system change, under the assumption that all the system components remain monitored by sensors throughout their lifetime, also after a change of components. Here, we take "learning" to mean broadly any autonomous computational intelligence method able to infer (or even approximate) high-level models of knowledge from data. Causal Bayesian Networks [19] are standard graphical models which have been learnt from data examples. These models have straightforward translations into FTs, but are themselves NP-hard or require exponential time to synthesize accurately [8,16]. These networks will model a limited form of causality, namely global causal relationships, rather than a sequence of causal relationships among events local to the components of a system.

LIFT [25] is a recent approach for learning static FTs with Boolean event variables, n-ary AND/OR gates, annotated with event failure probabilities. The input to the algorithm is untimed observational data, i.e., a dataset where each row is a single observation over the entire system, and each column records the value of a system event. All intermediate events to be included in the FT must be present in the dataset, but not all may be needed in the FT, and a small amount of noise in the dataset can be tolerated. LIFT also includes a causal validation step (the Mantel-Haenszel statistical test) to filter for the most likely causal relationships among system events, but the worst-case complexity is exponential in the number of system events in the data. Its main advantage is that of being one of the few automated FT-learning methods which validate causality.

Learning Boolean Formulas and Classifiers from Data. Before LIFT, observational data were used to generate FTs with the IFT algorithm [24] based on standard decision-tree statistical learning. The advantage of learning a graphical decision tree out of data is the inherent interpretability of decision-tree models and their ease of translation into other graphical models. Boolean formulas or networks were also machine-learnt using a similar tree-based method [16,26]. The classic C4.5 learning algorithm yields a Boolean decision tree that is easily translatable into a Boolean formula by constructing the conjunction of all paths leading to a leaf modelling a True value (i.e., system failure), and then simplifying the Boolean function. The resulting models encode the same information as a decision tree (i.e., a classifier for the observational data), so lack the validation of causal relations, but are expected to preserve their predictive power about the system. This retained our attention: indeed, static FTs (in opposition to dynamic FTs, where time-dependence of events is considered) can be seen as Boolean functions. Furthermore, Boolean formulas were also

machine-learnt using black-box classifiers (namely, classifiers not easily interpretable as a graphical model). Such methods include SVMs, Logistic Regression and Naive Bayes.

We propose a novel algorithm to learn an FT that best (most accurately) classifies records in a tabular dataset composed of observational tuples, in which values (failures) for each Boolean basic event and Boolean top event in the system are known. We compare it with these existing learning algorithm, in terms of its performance when fitting data and robustness to noisy data.

3 Background

In this section, we first define the structure of a static FT (consisting of logic gates, and also of intermediate events) and a dataset from which an FT is then inferred. The formulations below follow definitions from [25].

FTs [32] are trees that model how component failures propagate to system failures. Since subtrees can be shared, FTs are in fact directed acyclic graphs (DAGs) rather than trees. Essentially, *intermediate events* in the FT are logical combinations of other intermediate events, with only *basic events* (BE) as the leaves of the tree, and one special intermediate event called the *top event* as root. Gates model how BE failures lead to system failures. Standard fault trees feature two types of gates: AND, and OR.

Definition 1. *A **gate** G is a tuple $(t, \boldsymbol{I}, O)$ such that:*

- *t is the type of G with $t \in \{And,\ Or\}$.*
- *$\boldsymbol{I}$ is a set of $n \geq 2$ intermediate events $\{i_1, \ldots, i_n\}$ that are inputs of G.*
- *O is the intermediate event that is the output of G.*

We denote by $I(G)$ the set of intermediate events in the input of G and by $O(G)$ the intermediate event in the output of G.

Definition 2. *An **AND** gate is a gate $(And, \boldsymbol{I}, O)$ where output O occurs (i.e. O is True) if and only if every $i \in \boldsymbol{I}$ occurs.*

Definition 3. *An **OR** gate is a gate $(Or, \boldsymbol{I}, O)$ where output O occurs (i.e. O is True) if and only if at least one $i \in \boldsymbol{I}$ occurs.*

Definition 2 requires that all system components modelled by the events in the input of the **AND** gate must fail in order for the system modelled by the event in the output to fail. Similarly, Definition 3 requires that one of the system components modelled by the events in the input of the **OR** gate must fail in order for the system modelled by the event in the output to fail.

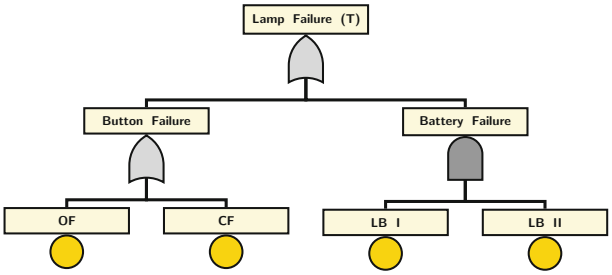
Definition 4. *A **basic event** B is an event with no input and one intermediate event as output. We denote by $O(B)$ the intermediate event in the output of B.*

Sometimes other gates are considered, like the XOR (exclusive OR), the voting gate and the NOT gate [17,32]. For the sake of simplicity, we focus on the AND and OR gates; other gates can be treated in a similar fashion. The root of the tree is called the *top event* (T). The *top event* represents the failure condition of interest, such as the stranding of a train, or the unplanned unavailability of a satellite. Thus, a FT fails if its *top event* fails.

Definition 5. *A* ***fault tree F*** *is a tuple* $(\boldsymbol{BE}, \boldsymbol{IE}, T, \boldsymbol{G})$ *where:*

- $\boldsymbol{BE}$ *is the set of basic events;* $O(B) \in \boldsymbol{IE},\ \forall B \in \boldsymbol{BE}$*. A basic event may be annotated with a probability of occurrence* p*.*
- $\boldsymbol{IE}$ *is the set of intermediate events.*
- T *is the top event,* $T \in \boldsymbol{IE}$*.*
- $\boldsymbol{G}$ *is the set of gates;* $I(G) \subset \boldsymbol{IE} \cup \boldsymbol{BE},\ O(G) \in \boldsymbol{IE},\ \forall G \in \boldsymbol{G}$*.*
- *The graph formed by* $\boldsymbol{G}$ *should be connected and acyclic, with the top event* T *as unique root.*

*We denote by IE(**F**) the set of intermediate events in* $\boldsymbol{F}$ *and by IE(G) the intermediate event corresponding to gate* G*.*



(a) Example Fault Tree.

OF	CF	LB I	LB II	T	count
0	0	0	0	0	900
0	0	0	1	0	15
0	0	1	0	0	5
0	0	1	1	1	25
0	1	0	1	1	5
0	1	1	0	1	5
1	0	0	0	1	35
1	0	1	0	1	5
1	1	0	0	1	3
1	1	1	0	1	2

(b) Example dataset.

Fig. 1. Example of Fault Tree and learning dataset.

We now define a data format from which we learn an FT, as a collection of records. Each record is a valuation for the set of BEs (variable that models the state of one basic, indivisible system component), indicating whether a *failure* was observed for that BE. We assume that our dataset is *labeled*, i.e., also indicates whether the top event T has failed, yielding the predicted outcome of the FT.

Definition 6. *A* record R *over the set of variables* $\boldsymbol{V}$ *is a list of length* $|\boldsymbol{V}|$ *containing tuples* $[(V_i, v_i)], 1 \leq i \leq |\boldsymbol{V}|$ *where* V_i *is a variable,* $V_i \in V$ *and* v_i *is a Boolean value of* V_i*.*

Definition 7. *A* dataset $\boldsymbol{D}$ *is a set of* $|\boldsymbol{D}|$ *records, all over the same set of variables* $\boldsymbol{V}$*. Each variable in* $\boldsymbol{V}$ *forms a column in* $\boldsymbol{D}$*, and each record forms a row. When* k *identical records are present in* $\boldsymbol{D}$*, a single such record is shown, with a new count column for the value* k*.*

Figure 1a shows a FT modeling a lamp failure. The top OR-gate shows that a lamp fails if there is either a button failure or a battery failure. A button failure happens if either an operator (**OF**) or a cable (**CF**) fails. The AND-gate indicates that battery failure happens if both batteries are low (**LB I** and **LB II**). Figure 1b shows a corresponding dataset.

4 Learning Fault Trees with Nature-Inspired Stochastic Optimization

Evolutionary algorithms (EAs) were among the earliest artificial intelligence methods, first envisioned by Alan Turing in 1950 [31]. EAs are heuristics that mimic biological evolution: one starts with an initial population, and iteratively generates new individuals through modification and recombination, where only the best individuals are kept in the next generation – mimicking survival of the fittest. EAs are particularly suitable to automatically learn models of some kind, such as trees, graphs or matrix structures, free-form equations, sets and permutations, and synthetic computer programs, etc. Evolutionary algorithms have been very successfully applied in several domains, ranging from antenna designs for spacecrafts [13], graph-like network topologies [5] and matrix-like robot designs [7].

The main challenges in devising EAs are (a) formalizing what is a syntactically correct *solution* to the problem (in our case, a well-formed fault tree), and (b) formalizing what makes, semantically, a solution better than another, i.e., writing a *fitness function* which takes any proposed solution and returns a numerical "goodness" for that solution. In our case, we want the fault tree to be consistent with the observational data. Hence the fitness function is the proportion of records correctly classified. Then, the EA will aim to *maximize* the fitness as close as possible to its optimal value of 100%. The EA consists of an iterative optimization process, which maintains a *population* of candidate solutions at each iteration, and randomly mutates and combines (i.e., applies *genetic operators* to) solutions from a population, such that the fitness of the *best solution* per population improves in time.

1. *Initialization*: The initial population contains two simple FTs; all variables in the input dataset **D** are represented as BEs in these FTs.
2. *Mutation and recombination*: Genetic operations are performed on the FTs and generate new FTs.
3. *Evaluation*: The fitness of each new FT is evaluated.
4. *Selection*: High-fitness FTs from the new generation replace low-fitness FTs from the previous generation.

5. *Termination*: Steps 2 to 4 are repeated until a given termination criterion is met. This can be if at least one solution in the population exceeds a given fitness bound, or if a given maximum number of iterations is reached. Upon termination, the best solution or solutions in the population are returned.

We describe these steps in more detail below.

4.1 Initialization

Our EA takes as input a dataset $\mathbf{D}$ and aims at computing an FT with maximal fitness to this dataset. The dataset yields the set of BEs, as well as the top level event T. We start with an initial population consisting of the following two FTs, where all BEs are connected to T via an AND and an OR gate respectively $F_1 = (\mathbf{BE}, \{T\}, T, \{(And, \mathbf{BE}, T)\})$ and $F_2 = (\mathbf{BE}, \{T\}, T, \{(Or, \mathbf{BE}, T)\})$, where $\mathbf{BE} \cup \{T\} = \mathbf{V}$, so all the basic events and the top event must be in the dataset $\mathbf{D}$. These two FTs are the simplest structures including all the observational variables in the data, with an AND gate and an OR gate, respectively, at the top of the FT, and all BEs as inputs of this gate. These two individuals act as a seed population; later populations are larger in size. The population size is a setting depending on the nature of the FT to learn (namely, the number of BEs). Since the time complexity of the algorithm depends on the population size, increasing the population size may lead to scalability issues. It has also been shown in [6] that increasing the population size does not always perform as well as expected. In our experiments, we limit the population to hundreds of FTs.

4.2 Mutation and Recombination

We define seven stochastic genetic operators (one binary and six unary) which apply to FTs. For each iteration, each of the genetic operators operates on all individuals in the population with a given probability, to create new individuals. The order in which they are applied is randomized at each iteration.

G-create. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, create a gate $G \notin \mathbf{G}$, randomly select its nature (AND or OR), then randomly select a gate $G' \in \mathbf{G}$. Randomly select inputs events I' of $I(G')$ to become inputs of G such that $I(G) = I'$ and $I(G') = I(G') \setminus I'$. Then, add $O(G)$ to the input events of G' such that $I(G') = I(G') \cup O(G)$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \cup O(G), T, \mathbf{G} \cup \{G\})$ with $G \notin \mathbf{G}$.

G-mutate. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ and change its nature (AND to OR, or OR to AND).

G-delete. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ such that $O(G) \neq T$ and delete it. We set then, $I(G_p) = \bigcup_{i \in I(G)} IE_i$ such that $O(G) \in G_p$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \setminus O(G), T, \mathbf{G} \setminus \{G\})$.

BE-disconnect. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE}$, disconnect B from its intermediate event $G = O(B)$. The new FT if $\mathbf{F} = (\mathbf{BE} \setminus \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \notin I(G)$.

Note here that a gate G left with 0 or 1 input will not be removed: this is to preserve the solution search space and enable the connection of BEs to this gate. Only the genetic operator *G-delete* can remove such a gate.

BE-connect. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a basic event $B \notin \mathbf{BE}$ and $B \in \mathbf{V} \setminus T$, randomly choose a gate $G \in \mathbf{G}$ and connect B to the input of G. The new FT is $\mathbf{F} = (\mathbf{BE} \cup \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \in I(G)$. Note that this operator is essentially the inverse of **BE-disconnect**. The relevance of this operator lies in the fact that some FTs in the population may not contain as many BEs as variables $\mathbf{V}$ in the dataset. Also, our definition of this operator implies that no BE will be input to 2 different gates. However, the connection of the same BE to 2 different gates can occur within a *crossover* operation.

BE-swap. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE}$ and a randomly chosen gate $G \in \mathbf{G} \setminus O(B)$, disconnect B from $O(B)$ and connect B to G.

Crossover. The crossover operator takes two FTs as input and swaps at random two of their subtrees. This leads to two new FTs, where the first fault tree contains the selected subtree of the second fault tree and vice versa. More precisely, one selects at random an intermediate event $IE_1 \in \mathbf{IE_1}$ from the first fault tree $\mathbf{F_1}$, and one also selects at random an intermediate event $IE_2 \in \mathbf{IE}_2$ from the section fault tree. Then one replaces in $\mathbf{F_1}$ the subtree under $\mathbf{IE_1}$ by the subtree under $\mathbf{IE_2}$. Similarly, one replaces in $\mathbf{F_2}$ the subtree under $\mathbf{IE_2}$ by the subtree under $\mathbf{IE_1}$. Given two input FTs $\mathbf{F_1} = (\mathbf{BE_1}, \mathbf{IE_1}, T_1, \mathbf{G_1})$ and $\mathbf{F_2} = (\mathbf{BE_2}, \mathbf{IE_2}, T_2, \mathbf{G_2})$, randomly select an $IE_1 \in \mathbf{IE_1}$ and $IE_2 \in \mathbf{IE_2}$. Then, we set $I(O(IE_1)) = I(O(IE_1)) \setminus IE_1 \cup IE_2$ and $I(O(IE_2)) = I(O(IE_2)) \setminus IE_2 \cup IE_1$. Finally, $O(IE_1) = O(IE_2)$ and vice versa.

4.3 Evaluation

We define the *fitness* of an FT as the number of records in the dataset for which the value of the top event, given the values of the BEs, is correctly computed. The count of a record, i.e. the number of appearances, give more weight in the fitness function to records that occur often. In this way, noisy data can be better handled, which often happen in real life applications.

Definition 8. *The fitness of a fault tree is its accuracy w.r.t. the dataset* $\boldsymbol{D}$ *s.t.*

$$f = \frac{\sum\limits_{r \in D} x}{\sum\limits_{r \in D} k} \text{ where } \begin{cases} x = k \text{ if } V[T] = P[T] \\ x = 0 \qquad \text{otherwise} \end{cases}$$

where $P[T]$ stands for the predicted value of the top event given the dataset $\boldsymbol{D}$ for a given FT, $V[T]$ the real value of the top event and k the number of occurrences of the record r.

4.4 Selection

The selection strategy of the best individuals to undergo genetic operations is essential to increase the improvement rate of the fitness of the population. Commonly used strategies are roulette wheel, stochastic universal sampling, tournament and random selections. In all our experiments, we use an elitist strategy. The nature of the individuals motivates this choice: best-fitted FTs are the closest in the population to the optimal solution. They consist of Boolean gates and BEs, which means that a least fitted solution needs more genetic operations to become optimal. We thus hope that mutating the best FTs will be less costly in terms of iterations of the EA in order to converge towards the right solution.

4.5 Termination

The decision of whether to return the best FTs in the actual population or to continue the evolutionary process follows the termination criteria. In our experiments, we used the standard termination criteria, which are:

1. at least one solution in the population achieves an accuracy of 1, which means a perfect fitness to the data.
2. a maximum number of allowed iterations is reached.
3. convergence: no improvement of the best FT in the population has been observed for a given number of iterations.

Note that several runs of our EA may return different FTs with the same fitness, especially in terms of structure. Indeed, two FTs with a different structure may be semantically equivalent; FTs, like Boolean formulas, can be factorized to Disjunctive Normal Form (DNF, where a Boolean formula is standardized as a disjunction of conjunctive clauses) or to Conjunctive Normal Form (CNF, where a Boolean formula is standardized as a conjunction of disjunctive clauses). As a result, it may happen that all variables in the dataset do not appear in the FT since they are either not needed or not relevant. In our experiments, we chose to compute the CNF of the best-fitted FT. The transformation of an FT into a CNF is based on the following rules: the double negative law, De Morgan's laws, and the distributive law.

5 Learning of Partial Fault Trees

A fruitful application of learning fault trees is the learning of partial models, where domain experts partially know the structure of the FT, and other parts need to be inferred from data. In this way expert knowledge and data-driven approaches are aggregated. To accommodate this approach, we propose here

a variation of our EA. We parameterize our EA with such a partially known structure as input to the algorithm. The task for the EA is then to evolve *sub*-Fault Trees, given the known *skeleton* of the FT. The initial population becomes then the partial structures given as input. Genetic operators are slightly modified to ensure the given skeletons to remain unmodified in each mutated FT. We gather at any moment of the evolutionary process a population composed of FTs containing the allowed skeleton. In this section, we detail the initialization and the genetic operators to take into account the expert knowledge provided as a *skeleton* FT.

5.1 Initialization

In the same way as the procedure described in Sect. 4, our evolutionary algorithm takes as input a dataset $\mathbf{D}$, as well a known FT-*skeleton* F_o. We start then with an initial population consisting of this FT-*skeleton*:

$$F_o = (\mathbf{BE_o}, \mathbf{IE_o}, T_o, \mathbf{G_o})$$

where $\mathbf{BE_o}$ are BEs contained in the skeleton, $\mathbf{IE_o}$ are the intermediate events of the skeleton, T_o is the top event of the skeleton and $\mathbf{G_o}$ is the set of gates contained in the skeleton. This is this structure given as input that will remain in all mutated FTs all along the evolutionary process.

5.2 Mutation and Recombination

In order to preserve the FT-*skeleton* during mutation and recombination operations, we have to adapt the following genetic operators.

G-create-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, create a gate $G \notin \mathbf{G}$, randomly select its nature (AND or OR), then randomly select a gate $G' \in \mathbf{G}$. Randomly select inputs events I' of $I(G') \setminus \mathbf{IE_o}$ to become inputs of G such that $I(G) = I'$ and $I(G') = I(G') \setminus I'$. Then, add $O(G)$ to the input events of G' such that $I(G') = I(G') \cup O(G)$. The new FT is $\mathbf{F} = (\mathbf{BE}, \mathbf{IE} \cup O(G), T, \mathbf{G} \cup \{G\})$ with $G \notin \mathbf{G}$. In that way, we allow the creation of a gate such that its input events are not in the gates of the skeleton, that is, the skeleton remains unchanged.

G-mutate-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G} \setminus \mathbf{G_o}$ and change its nature (AND to OR, or OR to AND).

G-delete-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G} \setminus \mathbf{G_o}$ such that $O(G) \neq T$ and delete it. We set then, $I(G_p) = \bigcup_{i \in I(G)} IE_i$ such that $O(G) \in G_p$.

BE-disconnect-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE} \setminus \mathbf{BE_o}$, disconnect B from its intermediate event $G = O(B)$. The new FT if $\mathbf{F} = (\mathbf{BE} \setminus \{B\}, \mathbf{IE}, T, \mathbf{G})$, where $B \notin I(G)$.

BE-swap-o. Given the input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$ and a randomly chosen basic event $B \in \mathbf{BE} \setminus \mathbf{BE_o}$ and a randomly chosen gate $G \in \mathbf{G} \setminus O(B)$, disconnect B from $O(B)$ and connect B to G.

Crossover-o. Given two input FTs $\mathbf{F_1} = (\mathbf{BE_1}, \mathbf{IE_1}, T_1, \mathbf{G_1})$ and $\mathbf{F_2} = (\mathbf{BE_2}, \mathbf{IE_2}, T_2, \mathbf{G_2})$, randomly select an $IE_1 \in \mathbf{IE_1} \setminus \mathbf{IE_o}$ and $IE_2 \in \mathbf{IE_2} \setminus \mathbf{IE_o}$. Then, we set $I(O(IE_1)) = I(O(IE_1)) \setminus IE_1 \cup IE_2$ and $I(O(IE_2)) = I(O(IE_2)) \setminus IE_2 \cup IE_1$. Finally, $O(IE_1) = O(IE_2)$ and vice versa.

Note that the scenario where expert guidance is used to lead to faster convergence of the FT learning algorithm is realistic: indeed, this variation is helpful to refine existing FTs, or checking if a handmade model is accurate given real-world measurements.

6 Experimental Evaluation

We have evaluated the efficiency and effectiveness of our EA method using a large number of cases. We compared our methods with six other learning techniques: five approaches from the literature, and the variant of our own EA technique for learning partial fault trees. For these methods, we investigated both the accuracy and as well as runtime. Our comparisons were performed for a set of synthetic cases (Sects. 6.2 and 6.3), as well as for industrial benchmarks (Sects. 6.4 and 6.5).

6.1 Experimental Set Up

The first three methods in our evaluation are: (1) Support Vector Machine (abbreviated svm in the figures), (2) Logistic Regression (abbreviated log) and (3) Naive Bayes Classifier (nba). These methods are Boolean classifiers that, given the values of the BEs, predict the value of the top event T. Being classifiers, methods (1)–(3) do not yield FT models, only a prediction for the value of T.

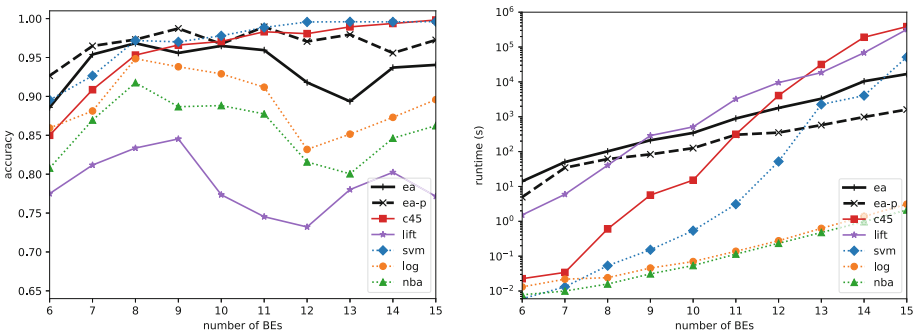
Then, we have used three methods that do learn fault tree models: (4) We have compared our results to the well-known C4.5 algorithm for learning decision trees (abbreviated c45). Decision trees can be transformed to FTs, by first computing in the decision tree the conjunction of all paths leading to failure leaves, and then simplifying the conjunction to CNF. (5) We have also compared to the earlier LIFT approach, which re returns an FT. (6) Finally, we used the variation of the EA for learning partial fault trees (ea-p). Here, we assumed that the two upper layers of the fault trees were fixed.

To compare these methods, the observational data was divided into two sets: one training set, used as input to the EA (with an average of 2/3 of all possible observations), and a test set containing all observational variables (complete boolean table), used to evaluate the solution returned by our algorithms. The parameters of the EA were set as follows: we used a population size of 100. As termination criteria, we used either a maximum number of 100 iterations or

an observed convergence (i.e. no improvement of the best individual's fitness) over 10 iterations or an FT with fitness 1 (optimal solution) in the population. Each genetic operator was applied with probability 0.9 in order to increase the mutation rate of the population. The selection and replacement strategy were elitists, to systematically replace least-fitted individuals in the old population by the best individuals in the union of the old population and the set of newly generated individuals. Finally, to homogenize several runs of the EA, the conjunctive normal form (CNF) of the best FT was returned in the termination step. Note that our Python implementations and dataset are available[2], and that we used state-of-the-art implementations of the scikit-learn library for techniques (1)–(5).

6.2 Synthetic Dataset: Accuracy and Runtime

We have first used a large synthetic case. We considered 100 randomly generated fault trees with 6 to 15 BEs, and for each FT, a randomly generated data set, with 200 to 230k records. Figures 2a and b present respectively the average accuracy and the average runtime, both as functions of the number of BEs. Figure 2a shows that the svm and c45 methods have the highest accuracy. However, the svm method only provides a classifier, not a fault tree. Further, the c45 method does not perform well in terms of runtime, see Fig. 2b. Our methods EA and EA-p perform reasonably well in terms of accuracy, as well as in term of run time. Finally, the log and nba method are fast but provide low accuracy. We also see that LIFT obtains less good results. An exponential complexity can explain this, and the fact LIFT requires data about intermediate events.



(a) Accuracy of the learnt FTs.

(b) Runtime of different algorithms.

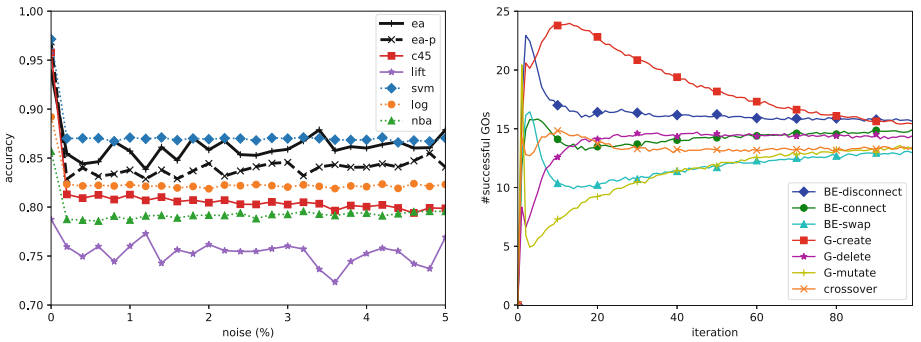
Fig. 2. Comparison of different learning algorithms.

We can also see that the more the FTs contain BEs, the more it is complicated to gather a solution with perfect fitness w.r.t. the training set. This is due to the

[2] https://gitlab.science.ru.nl/alinard/learning-ft.

significant number of iterations needed to converge to an optimal solution when dealing with a large number of BEs. However, we can see that expert knowledge is extremely beneficial in the case of ea-p, where the skeleton of the FT is given. It enables us to learn more accurate FTs (accuracy $> 95\%$) and faster (up to 10 times faster than the baseline EA).

6.3 Synthetic Dataset: Other Statistics



(a) Effect of noise on the learned FTs.

(b) Number of successful GOs per type.

Fig. 3. Statistics on genetic algorithm.

We also carried out an experiment where noise is added in the dataset, in order to test the robustness of the different algorithms. The noise varies from 0 to 5% of noisy records. We call a *noisy* record one where the value of at least one variable has been changed, i.e. measured incorrectly in real life. In the results shown in Fig. 3a, we see that our methods are relatively robust against noise compared to other methods. However, we see that the accuracy of the learned FTs drops whenever noise is present in the dataset.

Further, we also investigated which genetic operations were successful, as a function of the number of iterations, shown in Fig. 3(b). The latter is computed by looking at, for each iteration, the number of individuals issued from the same genetic operator who survived in the next generation, i.e. whose fitness was good enough to be kept in the population. We see that the success of most operations depends on the stage of the EA: this is the case for *BE-disconnect*, and *G-create*, which provide satisfying new individuals during the first iterations of the algorithm. An explanation is that *G-create* will increase the size of the FT, i.e. its complexity. Then, the search space of the solution is increased. In opposition to these gates, *G-mutate* seems to be a less good operator since the number of individuals issued from it tends not to survive in the population. This is mainly due to the change of semantics this operator implies: indeed, when the depth of the mutated gate is small (i.e. close to the top event), the meaning

of the resulting FT may drastically change. Hence a small number of successful operations of this type.

6.4 Case Study with Industrial Dataset

We present here an industrial case study based on the dataset from [23]. We consider here a component called the nozzle. The system containing the nozzles records large amounts of data about the state of the components over time, among them the failing of nozzles and nozzle-related factors. The dataset is composed of 9,000 records, 8 basic events being nozzles-related factors and a Boolean top event, standing for nozzle failure. We ran our genetic algorithm 10 times, with a maximum allowed iterations of 100 (convergence criterion of 10), and the fitness of the best FT learnt was of 0.997 (split ratio for train/test set of 80/20). The resulting FT has been validated by domain experts. Even in a practical context, multiple runs of the EA may return different (possibly equivalent) FTs, with different structures. Depending on the applications, expert knowledge can figure out whether one or the other returned FT is the most relevant to the case study. To help the selection process, one can place additional constraints on the FT, such as the number of children.

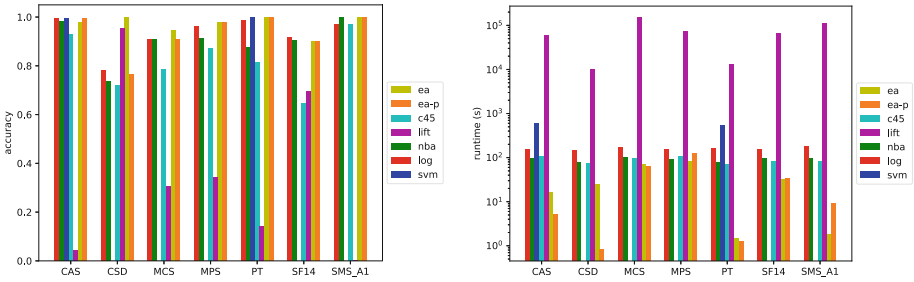
6.5 Fault Tree Benchmark

We present here the results we obtained for a set of publicly available benchmark suite[3], consisting of industrial fault trees from the literature, containing from 6 to 14 BEs and 4 to 10 gates. The FTs used are Cardiac Assist System (CAS), Container Seal Design Example (CSD), Multiprocessor Computing System (MCS), Monopropellant Propulsion System (MPS), Pressure Tank (PT), Sensor Filter Network (SF14) and Spread Mooring System (SMS_A1). Whereas the fault tree models were given in the literature, no data sets were available. Therefore, we have randomly generated these data sets, containing 10M records per case, in order to cope with low failure rates. Since the benchmark does provide failure probabilities per BEs, we have used those probabilities: If p_e is the failure probability of BE e in the benchmark, then we set, in each data record, $R[e] = 1$ with probability p_e. Figure 4a and b present the accuracy and runtime, respectively. Missing bars stand for experiments for which no result could have been obtained within 1 week of running time. We can see that for all case studies, our method is either the most or the second most efficient. We also see that in all cases, our method is among the most accurate methods.

7 Discussion

Extensions. The definition of gates and genetic operators can be extended. We show how to deal with **K/N** gates, which are gates of type $(k/N, \mathbf{I}, O)$ where

[3] https://dftbenchmarks.utwente.nl/.



(a) Results of FT Benchmark (accuracy). (b) Results of FT Benchmark (runtime).

Fig. 4. Results of Fault Tree Benchmark.

output O occurs (i.e. O is True) if at least k input events $i \in \mathbf{I}$ occurs, with $|\mathbf{I}| = N$. The cardinality of a k/N gate is said to be the number k. Note that this gate can be replaced by the OR of all sets of k inputs, but the use of k/N gates is much more compact for the representation of a FT. We can then define new genetic operators, such as **k-n-change** where, given an input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a k/N gate $G \in \mathbf{G}$ and change its cardinality, such that $k \in [1, N-1]$. We also extend the mutate gate operator of **G-mutate**, as follows: Given an input FT $\mathbf{F} = (\mathbf{BE}, \mathbf{IE}, T, \mathbf{G})$, randomly select a gate $G \in \mathbf{G}$ and change its nature (AND to OR or k/N; OR to AND or k/N; k/N to OR or AND). Similarly, we can redefine the create gate operator **C-create** such that the randomly selected nature of the new gate is chosen among AND, OR or k/N.

Our formalism can also handle **NOT** gates so that the FTs can become non-monotonic.

Limitations. While we can accurately learn small fault trees, the main limitation of our method at the moment is scalability. While other techniques, especially naive Bayesian classifiers, score well, techniques that learn models experience slower performance. Therefore, a solution may be to combine both methods. We can also use better heuristics on which GO to deploy, and with what parameters. Such ideas were also the key to the success of EAs in other application domains. The result obtained by the EA does not ensure a perfect fitness of the FT with regards to the data. This is the case when a maximal number of iteration has been reached, and the best FT in the population returned. Hence the near-optimality of our algorithm. In addition, multiple runs of the EA may return different (possibly equivalent) FTs, with different structures. We leave for further work the discovery of which of the returned FT is right, based on the data, figuring out causal relationships between variables using Mantel-Haenszel Partial Association score [2]. Another limitation lies in the growing size of the FTs after iterations: this may lead to overgrown FTs. However, we think that the best fit individuals may be compacted when returned: indeed, some gates in the FTs may contain none of only one input, and some factorization can be applied. We thus recommend performing FT reduction on the returned FTs,

such as the calculation of CNFs or DNFs. A first alternative would be to reduce to CNF or DNF the FTs in the population at each iteration of the EA. However, this would drastically reduce the search space of the EA (e.g. by making fewer mutations/recombinations possible, hence leading to a lower fitness) and go against genetic programming principles. A second alternative would be to take into account the size of the solutions as a second fitness function for the selection step. The implementation of such a multi-objective EA [9] is left for further work.

In all cases, learning FTs from an already known *skeleton* FT may suffer less from overgrown FTs after several iterations: the *skeleton* may give, indeed, already enough information on the structure of the FT. Thus, it helps the algorithm to converge faster to a solution.

8 Conclusion and Future Work

We presented an evolutionary algorithm for the automated generation of FTs from Boolean observational data. We defined a set of genetic operators specific to the formalism of FTs. Our results show the robustness and scalability of our algorithm. Our future research will focus on the learning of dynamic FTs, and especially trying to learn their specific gates such as PAND, FDEP and SPARE gates. We will also further look into Bayesian Inference and translating rules from Bayesian Networks to FTs. We also hope to take into account different failure modes of components thanks to INHIBIT gates. Finally, since there are many possible (i.e. logically equivalent) alternatives to an FT, we would like to investigate further what are the features of a good FT. In other words, we think that we need to characterize how much better a particular FT structure is compared to another.

References

1. Allen, D.J.: Digraphs and fault trees. Ind. Eng. Chem. Fundam. **23**(2), 175–180 (1984)
2. Birch, M.: The detection of partial association, i: the 2 × 2 case. J. R. Stat. Soc. Ser. B (Methodological) **26**, 313–324 (1964)
3. Bozzano, M., Bruintjes, H., Cimatti, A., Katoen, J.-P., Noll, T., Tonetta, S.: COMPASS 3.0. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 379–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_25
4. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA safety analysis platform. Int. J. Softw. Tools Technol. Transf. **9**(1), 5 (2007)
5. Bucur, D., Iacca, G., Squillero, G., Tonda, A.: The impact of topology on energy consumption for collection tree protocols: an experimental assessment through evolutionary computation. Appl. Soft Comput. **16**, 210–222 (2014)
6. Chen, T., Tang, K., Chen, G., Yao, X.: A large population size can be unhelpful in evolutionary algorithms. Theor. Comput. Sci. **436**, 54–70 (2012)

7. Cheney, N., MacCurdy, R., Clune, J., Lipson, H.: Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, pp. 167–174. ACM (2013)
8. Chickering, D.M., Heckerman, D., Meek, C.: Large-sample learning of Bayesian networks is NP-hard. J. Mach. Learn. Res. **5**, 1287–1330 (2004)
9. 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)
10. Dupont, P.: Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In: Carrasco, R.C., Oncina, J. (eds.) ICGI 1994. LNCS, vol. 862, pp. 236–245. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58473-0_152
11. Geiger, C.D., Uzsoy, R., Aytuğ, H.: Rapid modeling and discovery of priority dispatching rules: an autonomous learning approach. J. Sched. **9**(1), 7–34 (2006)
12. Henry, J., Andrews, J.: Computerized fault tree construction for a train braking system. Qual. Reliab. Eng. Int. **13**(5), 299–309 (1997)
13. Hornby, G., Globus, A., Linden, D., Lohn, J.: Automated antenna design with evolutionary algorithms. In: Space 2006, p. 7242 (2006)
14. Joshi, A., Gavriloiu, V., Barua, A., Garabedian, A., Sinha, P., Khorasani, K.: Intelligent and learning-based approaches for health monitoring and fault diagnosis of RADARSAT-1 attitude control system. In: 2007 IEEE International Conference on Systems, Man and Cybernetics, pp. 3177–3183 (2007)
15. Kabir, S.: An overview of fault tree analysis and its application in model based dependability analysis. Expert Syst. Appl. **77**, 114–135 (2017)
16. Kearns, M., Li, M., Valiant, L.: Learning boolean formulas. J. ACM **41**(6), 1298–1328 (1994)
17. Lee, W.S., Grosh, D.L., Tillman, F.A., Lie, C.H.: Fault tree analysis, methods, and applications: a review. IEEE Trans. Reliab. **34**(3), 194–203 (1985)
18. Leitner-Fischer, F., Leue, S.: Probabilistic fault tree synthesis using causality computation. Int. J. Crit. Comput. Based Syst. **4**(2), 119–143 (2013)
19. Li, J., Shi, J.: Knowledge discovery from observational data for process control using causal bayesian networks. IIE Trans. **39**(6), 681–690 (2007)
20. Li, S., Li, X.: Study on generation of fault trees from Altarica models. Procedia Eng. **80**, 140–152 (2014)
21. Li, Y., Zhu, Y., Ma, C., Xu, M.: A method for constructing fault trees from AADL models. In: Calero, J.M.A., Yang, L.T., Mármol, F.G., García Villalba, L.J., Li, A.X., Wang, Y. (eds.) ATC 2011. LNCS, vol. 6906, pp. 243–258. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23496-5_18
22. Liggesmeyer, P., Rothfelder, M.: Improving system reliability with automatic fault tree generation. In: Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, pp. 90–99 (1998)
23. Linard, A., Bueno, M.L.P.: Towards adaptive scheduling of maintenance for cyber-physical systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 134–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_9
24. Madden, M.G., Nolan, P.J.: Generation of fault trees from simulated incipientfault case data. WIT Trans. Inf. Commun. Technol. **6** (1994)
25. 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

26. Oliveira, A.L., Sangiovanni-Vincentelli, A.: Learning complex Boolean functions: algorithms and applications. In: Advances in Neural Information Processing Systems, pp. 911–918 (1994)
27. Papadopoulos, Y., McDermid, J.: Safety-directed system monitoring using safety cases. Ph.D. thesis, University of York (2000)
28. Park, M.S., Choi, J.Y.: Logical evolution method for learning Boolean functions. In: 2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace, vol. 1, pp. 316–321 (2001)
29. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15–16**, 29–62 (2015)
30. Sharvia, S., Kabir, S., Walker, M., Papadopoulos, Y.: Model-based dependability analysis: state-of-the-art, challenges, and future outlook. In: Software Quality Assurance, pp. 251–278. Elsevier (2016)
31. Turing, A.M.: Computing machinery and intelligence. In: Epstein, R., Roberts, G., Beber, G. (eds.) Parsing the Turing Test, pp. 23–65. Springer, Dordrecht (2009). https://doi.org/10.1007/978-1-4020-6710-5_3
32. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault tree handbook. Technical report, Nuclear Regulatory Commission Washington DC (1981)
33. Zhang, Y., Ren, Y., Liu, L., Wang, Z.: A method of fault tree generation based on go model. In: 2015 First International Conference on Reliability Systems Engineering (ICRSE), pp. 1–5. IEEE (2015)