SATMCS: An Efficient SAT-Based Algorithm and Its Improvements for Computing Minimal Cut Sets

Weilin Luo, Ou Wei, and Hai Wan

***Abstract*—Fault tree analysis (FTA) is a prominent reliability analysis method, which is widely used in safety-critical industries. Computing the minimal cut sets (MCSs) of a fault tree, i.e., finding all the smallest combinations of the basic events that cause system failures, is a fundamental step in FTA. Since coherent fault trees are the most common in industrial systems in practice, they are the focus of this article. Computing MCSs is a computationally hard problem. Classical methods have been proposed based on manipulation of Boolean expressions and binary decision diagrams. However, given the inherent intractability of computing MCSs in practice, there are still limitations on time and memory in these methods. Therefore, developing new methods over different paradigms remains to be an interesting research direction. In this article, motivated by recent progress on modern Boolean satisfiability problem (SAT) solvers, we present a new method for computing MCSs based on SAT, namely SATMCS. Specifically, given a fault tree, we iteratively search for a cut set based on the conflict-driven clause learning framework. By exploiting local propagation graph, which characterizes the partial failure propagation based on the cut set, we provide efficient algorithms for extracting an MCS. The new MCS is learned as a block clause for SAT solving, and the conflict clauses in iterations are incrementally recorded, which helps to prune search space and ensures completeness of the results. Moreover, we adopt a jump-chronological backtracking strategy to prepare the next iteration, which allows for reusing the same search steps in SAT solving. We compare SATMCS with state-of-the-art commercial tools on practical fault trees. Although SATMCS is only a prototype, it shows comparable performance in time consumption with one tool (XFTA), and in various cases, it outperforms the others (FaultTree+ and Commander). Besides, SATMCS exhibits much better performance on memory usage than these tools. Specifically, SATMCS consumes about one order of magnitude less memory usage in most instances.**

***Index Terms*—Boolean satisfiability problem, conflict-driven clause learning (CDCL), fault tree analysis (FTA), minimal cut sets (MCSs), reliability analysis.**

Manuscript received September 29, 2019; revised March 3, 2020 and June 2, 2020; accepted July 18, 2020. Date of publication August 14, 2020; date of current version June 1, 2021. This work was supported in part by the Guangdong Province Science and Technology Plan projects under Grant 2017B010110011 and Grant 2016B030305007, in part by the National Natural Science Foundation of China under Grant 61976232 and Grant 61573386, in part by the National Key R&D Program of China under Grant 2018YFC0830600, in part by the Guangdong Province Natural Science Foundation under Grant 2016A030313292, Grant 2017A070706010 (soft science), and Grant 2018A030313086, and in part by the Guangzhou Science and Technology Project under Grant 201804010435. Associate Editor: S. Liu. *(Corresponding author: Hai Wan.)*

Weilin Luo and Hai Wan are with the School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510275, China (e-mail: luowlin3@mail2.sysu.edu.cn; wanhai@mail.sysu.edu.cn).

Ou Wei is with the Department of Computer Science, University of Toronto, Toronto, ON M5S, Canada (e-mail: owei@cs.toronto.edu).

Digital Object Identifier 10.1109/TR.2020.3014012

I. Introduction

FAULT tree analysis (FTA) [1] is a prominent reliability analysis method widely used in safety-critical industries, such as aerospace [2] and power plants [3]. It computes a large class of system reliability properties and measures based on the fault tree that models failure propagation in a system. A cut set is a combination of component failures (*basic events*) such that the occurrence of the events leads to a system failure (*top event*) of the fault tree. A minimal cut set (MCS), i.e., the smallest combination of the basic events leading to the system failure, provides better characterization of the cause of the failure than a cut set. In practice, coherent[1] fault trees are the most common in industrial systems [4], so we focus on computing MCSs of coherent fault trees in this article.

Finding all the MCSs of a fault tree plays a fundamental role in FTA—both qualitative analysis, e.g., detection of common cause failures, and quantitative analysis, e.g., calculation of failure probabilities and importance measures, depend on MCSs. However, the computation of MCSs is a hard task, since it is exponential with respect to the number of basic events.

Classical methods to compute MCSs are based on manipulation of Boolean expressions [5], [6] and binary decision diagrams (BDDs) [7], [8]. The methods based on Boolean expressions use Boolean algebra laws to transform the Boolean formula defined by a fault tree into a set of MCSs, which potentially increases the size of the formula exponentially. The methods based on BDDs represent a fault tree with a BDD. When the fault tree exhibits a high degree of redundancy and an appropriate variable ordering is available, the size of the BDD can be greatly reduced. Efficient minimization algorithms then are applied to the BDD to encode all MCSs.

Recently, a number of improvements have been proposed, e.g., based on zero-suppressed BDD [9]–[12], symmetric structures [13], and ternary decision diagrams (TDDs) [14]. In practice, commercial FTA tools based on these methods have been widely used in industries. However, these methods often suffer from the exponential increment of memory usage because they need to construct a complete model of a fault tree. In the worst case, they fail to produce any results [15]. Therefore, given the inherent intractability of computing MCSs, developing new methods over different paradigms remains to be an interesting research direction.

[1]Fault trees are classified as coherent and noncoherent based on the monotonicity of Boolean formulas. Computing all MCSs of noncoherent fault trees is beyond the scope of this article.

The Boolean satisfiability problem (SAT) [16] determines whether there exists a model for a given Boolean formula. Although the problem is proved to be NP-complete [17], modern SAT solvers can handle large problems with tens of thousands of variables and hundreds of thousands of clauses and have been widely applied in industrial practice, such as bounded model checking [18], planning [19], and software verification [20]. Surprisingly, there is lack of works on computing MCSs of practical fault trees based on SAT solving.

In this article, motivated by recent progress on modern SAT solvers, we present a new method, namely SATMCS, for computing MCSs based on SAT. To this end, we reduce the computation of a cut set to a SAT problem and, then, extract an MCS from the cut set. Meanwhile, based on the conflict-driven clause learning (CDCL) framework [21], a depth-first search algorithm with effective pruning strategies for SAT solving, we design a framework to iteratively search for MCSs of a fault tree.

In this article, we present the following main features of SATMCS.

1) *Tseitin encoding of fault trees:* CDCL requires a Boolean formula in conjunctive normal form (CNF) for efficient search. The structure function of a fault tree, however, typically is a nonclausal[2] formula. Therefore, in order to use CDCL for computing MCSs, we first need to define an appropriate representation for encoding fault trees. The appropriate representation not only shall reduce the complexity of conversion from a nonclausal formula to a clausal formula, but also provide useful information for the subsequent effective calculation of MCSs. For this purpose, in this article, we consider encoding a fault tree in CNF using *Tseitin encoding* [22]. This encoding allows for a linear increase in the size of the formula, and more importantly, we are able to derive structural characteristics of the fault tree to extract an MCS efficiently.
2) *Extraction of MCSs using local propagation graph (LPG):* Extracting an MCS from a cut set is a key step with the goal of removing unnecessary basic events. A straightforward procedure for this can be done by direct calls to an external SAT solver. However, since SAT is in NP-complete, multiple SAT queries are still time-consuming. In our work, we provide a novel MCS extraction algorithm by exploiting the structural characteristics of a fault tree. We define an LPG induced by a cut set, which characterizes the local failure propagation within the events in the cut set. Testing necessity of basic events is then performed in an incremental way over the LPG that keeps being updated with the progress of testing, which improves the efficiency of MCS extraction.
3) *Clause learning with dynamic deletion:* This is the main challenge for guaranteeing efficiently finding *all* MCSs. To this end, we design a mechanism to continuously search for different MCSs and share learnt clauses during all solving. Meanwhile, we employ a clause deletion strategy, which periodically shrinks the size of the set of clauses, avoiding reduced CDCL performance and high memory usage caused by a large number of clauses.
4) *Jump-chronological backtracking:* At the end of each iteration, we prepare for searching for the next MCS by backtracking. The nonchronological backtracking, i.e., starting directly from the beginning, is simple, but the information of similarity between MCSs is lost. Considering the similarity between MCSs, we optimize this process by selecting a heuristic search depth to backtrack rather than going to the origin directly, which makes full use of the common information in the different iterations to search in the space nearby. Thanks to the strategy, the iterations are connected more closely for each other and the efficiency is improved.

[2]Throughout the article, the term nonclausal is used to denote propositional formulas not necessarily represented as CNF or DNF.

We have implemented an FTA tool for computing MCSs, SatFTA, which is powered by the SATMCS. We assess the performance of SATMCS in this article through experiments. We first evaluate SatFTA against three state-of-the-art commercial FTA tools (FaultTree+, XFTA, and Commander) over benchmarks from real-world applications. The experimental results show that, although SATMCS is only a implemented in a prototype, it dominates FaultTree+ and Commander in time consumption and is comparable with XFTA; moreover, SATMCS performs substantially better on memory usage than all tools.

We also discuss the capabilities of different SATMCS implementations in extracting MCSs and backtracking strategy. The experimental results show that the LPG and jump-chronological backtracking greatly improve the efficiency of SATMCS.

The rest of this article is organized as follows. In Section II, we introduce the basic concepts of the satisfiability problem and FTA. In Section III, we then present the main features of our method in detail. Section IV further discusses the impact of jump-chronological backtracking on SATMCS. We introduce our SATMCS-based fault analysis tools—SatFTA—in Section V and report the experiments for performance evaluation. In Section VII, we discuss related work. Finally, Section VIII concludes this article.

II. Preliminaries

This section introduces the notations and the backgrounds about the Boolean satisfiability problem and FTA.

A. Boolean Satisfiability Problem

Given a set of Boolean variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, a *literal* is either a Boolean variable $v_i \in \mathcal{V}$, called a *positive* literal, or its *negation* $\neg v_i$. v_i and $\neg v_i$ are *opposite*. A *product* π is a conjunction of literals, which is denoted as a set of literals. A product does not contain both a literal and its opposite. $|\pi|$ is the size of π, i.e., the number of literals in π. A product π *covers* another product π', iff $\pi' \supseteq \pi$. An *assignment* over $\mathcal{V}$ is denoted by a product π such that, for any $v \in \mathcal{V}$, v is assigned to true, iff $v \in \pi$, and is assigned to false, iff $\neg v \in \pi$. If all the variables in $\mathcal{V}$ are assigned, i.e., $|\pi| = |\mathcal{V}|$, π is called a *full* assignment over $\mathcal{V}$, and *partial* otherwise. Given a Boolean formula f defined over $\mathcal{V}$, π is a *model* of f, denoted by $\pi \models f$, iff f evaluates to true under π. A *clause* is a disjunction of literals. A Boolean

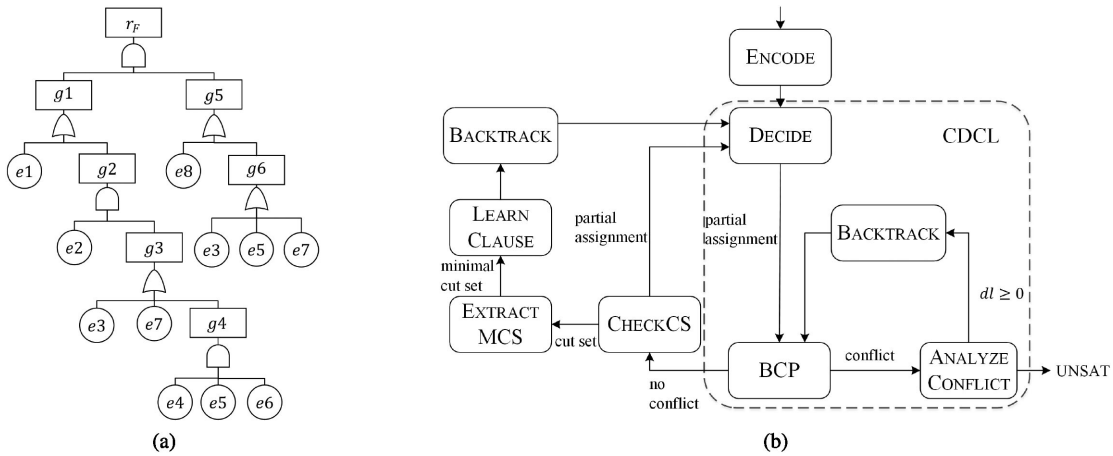

Fig. 1. (a) Fault tree $F = (e1 \vee (e2 \wedge (e3 \vee e7 \vee (e4 \wedge e4 \wedge e6)))) \wedge (e8 \vee (e3 \vee e5 \vee e7))$, where $E_F = \{e1, e2, e3, e4, e5, e6, e7\}$ and $G_F = \{r_F, g1, g2, g3, g4, g5, g6\}$. (b) Framework of SATMCS.

formula is in CNF if it is a conjunction of clauses, which is donated as a set of clauses.

Tseitin encoding [22] transforms a nonclausal Boolean formula into a clausal Boolean formula by adding auxiliary variables. Overall, Tseitin encoding introduces an auxiliary variable for every subformula and several clauses to constrain the value of this variable to be equal to the subformula it represents, in terms of the inputs to this subformula. The nonclausal Boolean formula is satisfiable if and only if the conjunction of these clauses together with the new variable associated with the topmost operator is satisfiable.

Research on the SAT problem studies the satisfaction of a Boolean formula [16]. Specifically, given a Boolean formula f, a SAT solver answers whether there exists an assignment π such that $\pi \models f$. If it exists, solver returns true and π, and false otherwise.

The main deterministic algorithm for SAT is based on the CDCL, whose framework is shown in the dashed box of Fig. 1(b). Main components of CDCL include DECIDE, BCP, ANALYZECONFLICT, and BACKTRACK. Given a Boolean formula f in CNF, CDCL searches for a full assignment satisfying f iteratively. During each iteration, DECIDE uses heuristics to pick an unassigned variable v of f for assigning and branching search. BCP then keeps applying the *unit clause rule* over the current partial assignment until either there is no more implication or a conflict is encountered. In the former case, CDCL calls DECIDE for deeper search until it obtains a full assignment, which means the instance is satisfiable. The depth of the search is indicated by a *decision level*. The decision level starts at 0 and is increased by 1 for subsequent decisions until a backtracking occurs. ANALYZECONFLICT reports a subset of the current partial assignment (its negative is called *learnt clause*) as a reason of the discovered conflict and returns a decision level to backtrack to. The information of the subset of the current partial assignment is recorded as a conflict clause that prevents the same conflict occurring again. BACKTRACK enters the decision level -1 only when some variable is implied by the formula to be both true and false, i.e., the instance is unsatisfiable.

B. Fault Tree Analysis

A fault tree F is a directed acyclic graph consisting of two types of nodes: *events* and *gates*. An event can be considered as a Boolean variable representing the failure status of a subsystem (*internal* event) or an individual component (*basic* event), where events occur (respectively, do not occur), i.e., corresponding variables are assigned as `true` (respectively, `false`). Especially, the *top* event indicates the final state of the system. Events are connected through gates; each gate is associated with an internal event as its output and several events as its inputs. Following the logical structures in F, the semantics of F is a Boolean formula built over basic events, called the *structure function*. An example is shown in Fig. 1(a).

Formally, a fault tree F is a triple $\langle E_F, G_F, \mathcal{Q} \rangle$, where E_F and G_F are sets of basic events and internal events, respectively, and $r_F \in G_F$ is top event; $\mathcal{Q}$ is a set of *Boolean equations*. Each Boolean equation $Q_{\hat{g}} \in \mathcal{Q}$, describing the gate $\hat{g}$ associated with the internal event $g \in G_F$, is a tuple $\langle Op,\ In,\ Out \rangle$, where $Op \in \{\mathsf{AND}, \mathsf{OR}\}$ is the type of $\hat{g}$, $In \subseteq E_F \cup G_F$ is the set of input events of $\hat{g}$, and $Out = g$ is the output event. We use an example to illustrate the Boolean equations.

Example 1: The fault tree shown in Fig. 1(a) can be expressed as a set of Boolean equations: $\mathcal{Q} = \{\langle \mathsf{AND}, \{g1, g5\}, r_F \rangle, \langle \mathsf{OR}, \{e1, g2\}, g1 \rangle, \langle \mathsf{AND}, \{e2, g3\}, g3 \rangle, \langle \mathsf{OR}, \{e3, e7, g4\}, g3 \rangle, \langle \mathsf{AND}, \{e4, e5, e6\}, g4 \rangle, \langle \mathsf{OR}, \{e8, g6\}, g5 \rangle, \langle \mathsf{OR}, \{e3, e5, e7\}, g6 \rangle\ \}$.

Let π be a product corresponding to the basic events E_F. If $\pi \models F$, π is called a *cut set* of F; π is said to be *minimal* iff there does not exist another cut set π' such that $\pi' \subset \pi$. In practice, most fault trees have *coherent* behavior, and they are called coherent fault trees. Formally, for a coherent fault tree F, if $\pi \subseteq \pi' \wedge \pi \models F$, then $\pi' \models F$. Intuitively, if the occurrence of an event set leads to system failure, then the occurrence of any of its supersets also leads to system failure. In this article, we focus on computing MCSs of coherent fault trees, because they are the most common in industrial systems. Besides, we assume that fault trees are expressed in negation normal form

(NNF), i.e., NOT gate ($\neg$) only appears at the level of basic events; therefore, it is enough to consider coherent fault trees with AND ($\wedge$) and OR ($\vee$) gates ($Op \in \{\mathsf{AND}, \mathsf{OR}\}$).

The failure propagation of the coherent fault tree is the process as follows: 1) some basic events occur, i.e., there are some failures in basic events; and 2) the failures are transmitted through internal events upward until triggering the top event. Note that the basic events that do not occur cannot lead to failure propagation for coherent fault trees. In other words, for a coherent fault tree, it is enough to consider the positive literal in a cut set for extracting an MCS from the cut set [8].

III. SAT-Based Computation of MCSs

In this section, we propose the algorithm of SAT-based computing MCSs—SATMCS. We first give the overview of the algorithm and then introduce the details.

A. Overview

In this article, we reduce the computation of a cut set into a SAT problem. In a nutshell, given a fault tree F, we query the SAT solver whether F is satisfiable. If there is a model, we extract a cut set from the model. However, there are two challenges for computing all MCSs via SAT queries. First, a modern SAT solver only accepts a CNF as input, whereas F is nonclausal. We will discuss this challenge in Section III-B. Second, it would be very expensive by naively querying SAT solver to obtain all MCSs, since one SAT query only returns one model. Therefore, we design a new framework shown in Fig. 1(b) by extending CDCL [23] to make it capable of computing all MCSs efficiently. Besides, the extension leverages the advantage of CDCL, i.e., effective pruning strategies, so it helps reduce the search time.

The framework of SATMCS includes CDCL parts, CheckCS, ExtractMCS, LearnClause, as well as Backtrack. SATMCS takes a fault tree F represented as a set of clauses $\mathcal{F}$ as an input, and its output is all the accurate MCSs of F. SATMCS maintains sets of clauses $\mathcal{H} = \mathcal{F} \cup \mathcal{B} \cup \mathcal{C}$ and $\mathcal{M}$, where $\mathcal{F}$ encodes F in CNF; using block clauses, $\mathcal{B}$ blocks already computed MCSs recorded by $\mathcal{M}$; $\mathcal{C}$ contains the conflict clauses from CDCL. In a iteration, on the one hand, SATMCS uses $\mathcal{C}$ to continually shrink the search space in which it is certain that there is no model of F. On the other hand, SATMCS uses $\mathcal{B}$ to block the models that has been found. Eventually, $\mathcal{B}$ prunes all the search space covered by all MCSs, i.e., all the MCSs of F have been found. At this point, $\mathcal{H}$ is unsatisfiable, and the algorithm terminates.

A complete iteration of SATMCS is shown as follows.

1) According to CDCL, SATMCS searches for a cut set of F not covered by any already computed MCSs. Note that there are two differences from CDCL. First, we keep updating $\mathcal{C}$ in all iterations, which avoids the same conflict clauses coming from different iterations. Second, when there is no conflict detected for a partial assignment π in BCP, CheckCS is called to determine whether π is a model of $\mathcal{F}$. It ensures that a shorter cut set can be obtained at the first time, which helps the extraction of MCSs.
2) If π is not a cut set, SATMCS will continue searching; otherwise, ExtractMCS, a procedure for testing necessity of the literals in π, is called to extract an MCS from π.
3) The information of a new MCS is learned by LearnClause by adding a block clause into $\mathcal{B}$, preventing the same MCSs being computed again.
4) The updated $\mathcal{H}$ is sent to Decide for next iteration after Backtrack computes the decision level to backtrack.

SATMCS guarantees to obtain all MCSs, because it is based on the techniques of search space pruning. It is obvious that SATMCS terminates ($\mathcal{H}$ is unsatisfiable), which means that all models of $\mathcal{F}$, i.e., cut sets, have been blocked by the clauses in $\mathcal{B}$ corresponding to all MCSs. The proof is provided in the Appendix. Besides, using constantly updated $\mathcal{B}$ and $\mathcal{C}$, SATMCS can incrementally prune the search space. However, as the number of clauses increases, SATMCS encounters a performance bottleneck, because the performance of BCP is limited by the number of clauses. We will discuss this challenge in Section III-D. Based on this framework, we leverage the information of similarity between MCSs to speed up the computation of MCSs. We will introduce this method in Section IV.

B. Tseitin Encoding of Fault Trees

The modern SAT solvers based on the CDCL allow Boolean formula in CNF as their input, which takes full advantage of the unit clause rule to imply new assignments in BCP. Therefore, SATMCS needs a fault tree F encoded in CNF as input.

To encode F in CNF, we need to address the following two issues. The first one comes from the fact that if a nonclausal formula is transformed into an equivalent formula in CNF only using the variables in E_F, it will potentially increase the size of the formula exponentially. The second one is that we expect that the encoding method can bring out information, such as structural characteristics of the fault tree, to optimize the computation of MCSs.

Fortunately, at the price of introducing new variables for subformulas, Tseitin encoding [22] enables a transformation with only a linear increase in the size of the formula. In our work, given a set of Boolean equations $\mathcal{Q}$ of F, we directly encode F as the CNF formula $\mathcal{F}$ by Tseitin encoding. According to the structures of fault trees in $\mathcal{Q}$, the auxiliary variables correspond naturally to the internal events G_F. Formally, for each Boolean equation $Q_{\hat{g}} = \langle Op, \{x_1, \ldots, x_n\}, g\rangle$, $Q_{\hat{g}}$ is encoded as a set of clauses $U_{\hat{g}} = L_{\hat{g}} \cup R_{\hat{g}}$, where $L_{\hat{g}}$ is equal to $g \models Op(x_1, \ldots, x_n)$, and $R_{\hat{g}}$ to $Op(x_1, \ldots, x_n) \models g$. If $Op = \mathsf{OR}$, then $L_{\hat{g}} = \{\neg g \vee \bigvee_{i=1}^{n} x_i\}$ and $R_{\hat{g}} = \bigcup_{i=1}^{n}\{\neg x_i \vee g\}$; if $Op = \mathsf{AND}$, then $L_{\hat{g}} = \bigcup_{i=1}^{n}\{\neg g \vee x_i\}$ and $R_{\hat{g}} = \{g \vee \bigvee_{i=1}^{n} \neg x_i\}$. Finally, we obtain $\mathcal{F} = \bigcup_{g \in G_F} U_{\hat{g}} \cup \{r_F\}$ by Tseitin encoding. We use an example to illustrate the Tseitin encoding.

Example 2 (Example 1 continued): Considering $Q_{\hat{g3}} = \langle \mathsf{OR}, \{e3, e7, g4\}, g3\rangle$, Tseitin encoding first introduces auxiliary variables $g3$ and $g4$; then, it obtains $L_{\hat{g3}} = \{\neg g3 \vee e3 \vee e7 \vee g4\}$ and $R_{\hat{g3}} = \{\{\neg e3 \vee g3\}, \{\neg e7 \vee g3\}, \{\neg g4 \vee g3\}\}$; finally, $U_{\hat{g3}} = L_{\hat{g3}} \cup R_{\hat{g3}}$.

Note that every Boolean formula can be transformed in linear time to NNF, so we transform a fault tree F to a set of clauses

$\mathcal{F}$ in linear time using Tseitin encoding. In addition, Tseitin encoding can be optimized by considering only $L_{\hat{g}}$ for each $Q_{\hat{g}}$. We include $R_{\hat{g}}$ in our encoding, since it helps to force assignments to auxiliary variables in BCP, which simplifies the extraction of MCSs later based on the structural characteristics of the fault tree.

Definition 1: A structural characteristic is a tuple $\langle O, I\rangle$, where $O \in G_F$ and $I \in 2^{E_F \cup G_F}$. $\langle O, I\rangle$ fulfills that $\bigwedge_{l_i \in I} l_i \models O$.

Intuitively, $\langle O, I\rangle$ means that I is a reason why O occur. We build the structural characteristics ζ by following step.

1) For each $\langle \mathsf{AND}, In, l\rangle \in \mathcal{Q}$, we add $\langle l, In\rangle$ into ζ.
2) For each $\langle \mathsf{OR}, In, l\rangle \in \mathcal{Q}$, we, for each $l' \in In$, add $\langle l, \{l'\}\rangle$ into ζ.

Property 1: Let $\pi^+ = \pi^+_{E_F} \cup \pi^+_{G_F}$ be a model of $\mathcal{H}$, where $\pi^+_{E_F}$ (respectively, $\pi^+_{G_F}$) is a set of positive literals corresponding to basic (respectively, internal) events. For each $l \in \pi^+_{G_F}$, there are at least one structural characteristic $\langle l, \hat{I}\rangle$, such that $\hat{I} \subseteq \pi^+$.

Property 1 shows that, thanks to Tseitin encoding, we can trace several structural characteristics in SAT solving. In other words, given a model of $\mathcal{H}$, we can find all the reasons why the internal events occur. It is the key to build an LPG (see Section III-C). We use an example to illustrate why Property 1 is right in SAT solving.

Example 3: For the fault tree [see Fig. 1(a)], where $\langle g3, \{e3\}\rangle \in \zeta$, suppose $e3 \in \pi^+$, $g3$ must be in π^+ because the clause $\{\neg e3 \vee g3\}$ in $R_{\hat{g3}}$ is unit under $e3$ is `true`.

C. Extracting MCSs Using LPG

According to the framework of SATMCS, we exploit CDCL to search for a cut set and extract an MCS from the cut set. In order to improve the performance, we propose the LPG for extracting MCSs. The LPG is built by a partial model of $\mathcal{H}$, which means that it only characterizes the partial failure propagation. The proof that it is enough to extract an MCS from a partial failure propagation is provided in the Appendix (see Lemma 3). In this subsection, we first introduce how to obtain a partial model, i.e., a cut set with fewer literals, in CDCL. Then, we provide our method to build an LPG and extract MCSs from a cut set based on the LPG.

1) Checking of Cut Sets: As mentioned before, in order to find a cut set with fewer literals, if there are no conflicts detected in BCP, SATMCS calls CHECKCS to determine whether the current partial assignment π is a model of $\mathcal{F}$; if true, π is guaranteed to be a cut set of F and be not covered by any MCSs in $\mathcal{M}$; otherwise, SATMCS continues to search for a cut set of F. Note that, due to Tseitin encoding, there exist internal events in cut sets.

Detecting whether a partial assignment π is a cut set can be achieved by checking the satisfiability of $\mathcal{F}$ under π by traversing all clauses. However, it is unavoidable that a large number of redundant traversals for $\mathcal{F}$ would reduce the efficiency through CHECKCS significantly.

To address this problem, note that, in CDCL, the two partial assignments π_{dl-1} and π_{dl}, obtained in decision level $dl-1$ and dl, respectively, without backtracking, satisfy $\pi_{dl-1} \subset \pi_{dl}$, which denoted as the monotonic satisfiability of CNF. Therefore, we efficiently implement CHECKCS based on incrementally checking a cut set. In our implementation, we keep a list for the clauses in $\mathcal{F}$ and introduce the first unresolved clause for each decision level. The first unresolved clause u_{dl} is defined as the first unresolved clause in the list for decision level dl in CDCL. For decision level dl, CHECKCS watches the first unresolved clause u_{dl-1} and, instead of always checking the list from the beginning, checks whether the new assignments in dl can satisfy more clauses than that in the previous decision level, i.e., it starts checking from u_{dl-1}. Finally, the first unresolved clause for decision level dl is updated unless all the clauses are satisfied. Thanks to the monotonic satisfiability of CNF, this watching mechanism improves the efficiency of CHECKCS by avoiding unnecessary clause checking.

2) Building an LPG: After π is identified as a cut set of F, EXTRACTMCS is called to extract an MCS from π. We introduce the concepts of the *necessary and unnecessary* literal for extracting an MCS.

Definition 2: A literal $l \in \pi$ is an *unnecessary literal* of π, iff $\pi' = \pi \setminus l$ s.t. $\pi' \models f_F$; otherwise, l is *necessary*.

In short, the extraction of MCS can be regarded as testing the necessity of the literals in π, removing unnecessary literals, and finally obtaining an MCS. Let $\pi^+_{E_F}$ (respectively, $\pi^+_{G_F}$) be the set of positive literals in π corresponding to E_F (respectively, G_F). Because F is a coherent fault tree and the internal events in G_F are constrained by the basic events in E_F, the MCSs covering π must be a subset of $\pi^+_{E_F}$. Therefore, EXTRACTMCS conducts a sequence of queries on $\mathcal{F}$; each query is optimized to test the necessity for a literal in $\pi^+_{E_F}$. These tests can be done by direct calls to a SAT solver in naive implementation.

However, SAT solving is time-consuming, since the SAT problem is in NP-complete. Based on the observation that the SAT solving a transformed CNF formula from a fault tree did not take advantage of the structural characteristics of the fault tree, to improve efficiency, we exploit the structural characteristics of F, providing an incremental extraction algorithm based on an LPG induced by a cut set.

We first introduce the concept of the LPG.

Definition 3: Let π^+ be a partial model of $\mathcal{H}$ that only contains positive literals. The *LPG* over π^+ is a directed hypergraph $P_{\pi^+} = \langle V, R\rangle$, where $V = \pi^+$ is a set of events and $R \subseteq 2^V \times V$ is a set of hyperedges. An edge $(S, v) \in R$ denotes that occurrence of all the events in S causes the event v to occur. Specifically, there are only two types of hyperedges in LPG: one of which holds $|S| = 1$ defined by an OR gate; another holds $|S| > 1$ defined by an AND gate.

Intuitively, because π^+ is a partial model of H, an LPG can be considered as a projection of F over π^+, characterizing a partial failure propagation between the corresponding events in π^+ based on the structural characteristics of fault tree. Note that the top event r_F must be in V, since π is a cut set of F.

Algorithm 1 shows the procedure of building LPG P_{π^+}, where SEPARATE aims to get $\pi^+_{E_F}$ and $\pi^+_{G_F}$ from π^+; GETALLREASON is called to obtain a set of structural characteristics about an internal event l_{out}. Based on π^+, we trace all the reasons why the events in $\pi^+_{G_F}$ occur (lines 3–7). We use an example to illustrate the LPG.

Algorithm 1: BUILDLPG.

INPUT : A partial model π^+, and a set of structural characteristics ζ
OUTPUT : An LPG $\langle V, R\rangle$

```
1 V ← π+
2 π+_{E_F}, π+_{G_F} ← SEPARATE(π+)
3 for each l_out ∈ π+_{G_F} do
4     ζ_{l_out} ← GETALLREASON(l_out, ζ)
5     for each ⟨l_out, Î⟩ ∈ ζ_{l_out} do
6         if Î ⊆ π+ then
7             R ← R ∪ (Î, l_out)
8 return ⟨V, R⟩
```

Algorithm 2: EXTRACTMCS.

INPUT : A partial model π and a set of structural characteristics ζ
OUTPUT : An MCS mcs extracted from π

```
1 mcs ← ∅
2 (V, R) ← BUILDLPG(π+, ζ)
3 for each v ∈ π+_{E_F} do
4     if ISNECESSARY(R, v) then
5         mcs ← mcs ∪ {v}
6 return mcs
```

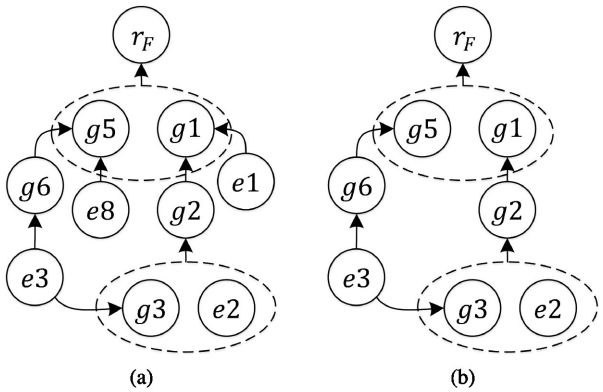

Fig. 2. (a) LPG. (b) Reduced LPG.

Example 4: Given the fault tree shown in Fig. 1(a), consider $\pi^+ = \{r_F, g1, g5, g2, g3, g6, e1, e2, e3, e8\}$ derived from a cut set π. The LPG over π^+ is shown in Fig. 2(a). For the event $g1$, if $e1$ occurs, $g1$ must be triggered according to $\langle \mathsf{OR}, \{g2, e1\}, g1\rangle$; therefore, P_{π^+} contains the hyperedge $(\{e1\}, g1)$. For the event $g2$, $\langle \mathsf{AND}, \{g3, e2\}, g2\rangle$ implies that $g1$ occurs iff both $g3$ and $e2$ occur; hence, P_{π^+} contains the hyperedge $(\{g3, e2\}, g2)$.

3) Extracting an MCS Based on LPG: We first introduce some concepts to illustrate the necessary literals in LPG and then propose the method to keep all the necessary literals in LPG for extracting an MCS.

Definition 4: Let $P_{\pi^+} = \langle V, R\rangle$ be an LPG and $v, v' \in V$. There is an *influence edge* between v and v', denoted as $v \dashrightarrow v'$, iff $(S, v') \in R$, where $v \in S$. If $\nexists (S,' v') \in R$, the edge is a *unit*, denoted as $v \mapsto v'$. Specially, if $v \in E_F$, then $v \mapsto v$.

We illustrate the influence edge and the unit influence edge via the following example.

Example 5 (Example 4 continued): In the LPG [see Fig. 2(a)], $e1 \dashrightarrow g1$ holds, while $e1 \mapsto g1$ does not hold, because of $g2 \dashrightarrow g1$.

Property 2: If $v \mapsto v'$ holds, then removing v from the LPG will lead to removing v' from the LPG.

Intuitively, Property 2 illustrates the key step to remove a literal (v) and its influence (v'). Specifically, v influences v' in two cases. Case 1, $|S| > 1$: by the construction of P_{π^+}, v' is associated with an AND gate. Hence, v' will be influenced by v. Case 2, $\neg(\exists \tilde{v} \neq v \cdot (\{\tilde{v}\}, v') \in R)$: $|S| = 1$ implies that v' corresponds to an OR gate. In this case, if v is the only event in π^+ that causes v' to occur ($v \mapsto v'$), v' is influenced by v.

We extend the concept $\mapsto$ into an *influence path* over a sequence $v_0, v_1, \ldots, v_n$ where $v_i \in V$: $v_0 \mapsto v_n$ iff $v_{i-1} \mapsto v_i$ holds for $i \in (0, n]$. Based on the above concepts, we have the following lemma. The proof of this lemma is provided in the Appendix.

Lemma 1: If $v \mapsto r_F$ holds, then l is a necessary literal; otherwise, v is unnecessary.

Based on Lemma 1, EXTRACTMCS tests the necessity of each basic event $v \in \pi^+_{E_F}$ by analyzing the influence paths in P_{π^+} triggered by v. This procedure is based on the main idea of iteratively searching for a fix point in a graph. If removing those paths triggered by v from P_{π^+} does not cause the top event r_F to be removed, v is unnecessary for causing r_F, i.e., a fix point is found, and P_{π^+} is reduced for next test; otherwise, $v \mapsto r_F$ holds, and v is included into an MCS. Because P_{π^+} only characterizes the failure propagation relevant to π^+, and P_{π^+} continues to be reduced with the detection, the efficiency of the sequence of queries in EXTRACTMCS is improved. Algorithm 2 shows the pseudocode of EXTRACTMCS. First, BUILDLPG constructs the LPG $P_{\pi^+} = \langle V, R\rangle$ based on Algorithm 1 (line 2). Then, ISNECESSARY (see Algorithm 3) tests the necessity of a basic event v in $\pi^+_{E_F}$ linearly (lines 3–5).

In every testing, ISNECESSARY creates a copy of R for backtracking and computes $eventSet$ through chaotic iteration to find a fix point. $eventSet$ is initialized with $\{v\}$ (line 2), used to record the influenced events along the influence paths triggered by v in LPG, that is, for every l' in $eventSet$, $v \mapsto l'$ holds. Lines 4–11 show the process of searching for a fix point based on Property 2. In this process, we try to remove the literal $l \in eventSet$ from LPG. Based on the hyperedges R', we keep finding the literal $l \mapsto l'$ (lines 5–11). If we find that l' influenced by v is the top event r_F (line 8), i.e., $v \mapsto r_F$, v will be necessary, and ISNECESSARY returns $TRUE$. Otherwise, l' is added into $eventSet$ (line 10) and the edge (S, l') is removed (line 11). If r_F is not detected during the iteration (a fix point), which means v is unnecessary for r_F, then R will be reduced for the next test, and ISNECESSARY returns $FALSE$ (lines 12 and 13).

The proof that there is an MCS in the final reduced LPG is provided in the Appendix (see Lemma 3). We introduce

Algorithm 3: IsNecessary.

INPUT : A set of hyper-edges R of an LPG and a testing literal v

OUTPUT : The necessity status of v and update R

```
1  R' ← R
2  eventSet ← {v}
3  while eventSet ≠ ∅ do
4      l ← eventSet.pop()
5      for each (S, l') ∈ R' do
6          if l ∈ S then
7              if (|S| > 1) or ¬(∃l̃ ≠ l · ({l̃}, l') ∈ R')
               then
8                  if l' is r_F then
9                      return TRUE
10                 eventSet ← eventSet ∪ {l'}
11             R' ← R' \ {(S, l')}
12 R ← R'
13 return FALSE
```

the computation of ExtractMCS via an example, shown as follows.

Example 6 (Example 4 continued): In the LPG [see Fig. 2(a)], note that different MCSs can be extracted from the set of basic events $\{e1, e2, e3, e8\}$, depending on the testing order. Suppose we test in the order of the index of the events. Because of the existence of the hyperedge $(\{g2\}, g1)$, $e1$ does not influence $g1$. Therefore, the edge $(\{e1\}, g1)$ can be removed from the graph. In the reduced graph, $e2$ becomes a necessary event because of the influence path $e2 \mapsto g2 \mapsto g1 \mapsto r_F$. Subsequent tests show that $e3$ is necessary because of the path $e3 \mapsto g3 \mapsto g2 \mapsto g1 \mapsto r_F$, while $e8$ is unnecessary. The final reduced LPG is shown in Fig. 2(b), and $\{e2, e3\}$ is a new MCS by retaining all the basic events in reduced LPG.

D. Clause Learning With Dynamic Deletion

With the framework shown in Fig. 1(b), we are able to enumerate all MCSs of a fault tree. Next, we design a new mechanism to share the learnt clauses in different iterations to prune the search space. Meanwhile, we employ a clause deletion strategy that periodically shrinks the size of the set of clauses, avoiding reduced CDCL performance and high memory usage caused by a large number of clauses.

To find all the MCSs and improve efficiency, the CNF formula $\mathcal{H} = \mathcal{F} \cup \mathcal{B} \cup \mathcal{C}$ keeps being updated in AnalyzeConflict and LearnClause. After a new MCS is extracted, a block clause consisting of the opposite to the literals in the MCS is added to $\mathcal{B}$, which ensures that the same MCS will not be discovered again. Moreover, similar to CDCL, AnalyzeConflict also updates $\mathcal{C}$ to learn the learnt clauses generated from conflicts, which helps BCP to prune the search space incrementally. $\mathcal{C}$ guarantees that learnt clauses learned by AnalyzeConflict can be used by BCP in this iteration. More importantly, in future iterations, these previously learned learnt clauses will be shared, which avoids repeated calculations about these clauses. By pruning of $\mathcal{B}$ and $\mathcal{C}$, SATMCS is tuned to search for MCSs until all the search spaces covered by MCSs are pruned by $\mathcal{B}$. At the same time, there is no model of $\mathcal{H}$. The proof of the following theorem (the correctness of SATMCS) is provided in the Appendix.

Theorem 1: SATMCS terminates, iff it obtains all the MCSs of the coherent fault tree F.

As the computation progresses, however, a large number of clauses in $\mathcal{H}$ will not only increase memory usage, but also increase the burden on BCP and reduce the efficiency of CDCL. To address this issue, we propose a clause deletion strategy to dynamically shrink the size of $\mathcal{H}$ in LearnClause. Note that the clauses in $\mathcal{F}$ and $\mathcal{B}$ are essential to ensure the correctness, whereas the clauses in $\mathcal{C}$ can be deleted without affecting the correctness. Moreover, the size of $\mathcal{C}$ grows much faster than that of $\mathcal{B}$, since finding one MCS typically requires several iterations over CDCL and multiple learnt clauses can be discovered by AnalyzeConflict. In fact, the number of MCSs could reach millions, while the number of learnt clauses is often tens of millions. It is clear that the memory usage of $\mathcal{C}$ is significantly higher than that of $\mathcal{B}$. Therefore, SATMCS reduces the clauses in $\mathcal{C}$ dynamically by a shrinking condition.

It is key to compute the shrinking condition because frequently reducing learnt clauses will lose the information of learnt clauses learned by the incremental pruning. Therefore, we consider the ratio of the number of the clauses in $\mathcal{C}$ to that in $\mathcal{H}$, i.e., $|\mathcal{C}|/|\mathcal{H}|$; when the ratio exceeds a certain threshold η, LearnClause deletes the clauses in $\mathcal{C}$. Note that, in practice, we set $\eta = 0.7$ for better performance.

IV. Jump-Chronological Backtracking

At each iteration, SATMCS prepares to search for a new MCS after blocking the last MCS in Backtrack. A straightforward method is to restart the solver under only keeping $\mathcal{H}$. However, it will lose some information during the searching process, such as the similarity between MCSs. In this section, we first discuss the different methods to backtrack. Then, we define the similarity between MCSs. Finally, we design a method to choose a reasonable decision level to backtrack based on the similarity between MCSs.

There are two straightforward ideas to backtrack. The first one, one by one, backtracks to the previous decision level, namely *chronological backtracking*. It possibly leads to an inconsistent $\mathcal{H}$. We explain the inconsistent case via an example.

Example 7 (Example 6 continued): Suppose that CDCL discovers a satisfying assignment $\pi^+ = \{r_F, g1, g5, e1, g2, g3, e2, e3, g6, e8\}$ through the following process: at decision level 0, r_F, $g1$, and $g5$ are implied; Decide makes a decision for $e1$ at decision level 1 where there is no implication; then, at decision level 2, Decide makes a decision for $g2$ and BCP propagates $g3$ and $e2$; after deciding $e3$, $g6$ is implied at decision level 3; finally, Decide makes a decision for $e8$ at the decision level 4. After computing a new MCS $\{e2, e3\}$, if it backtracks to decision level 3, obviously, there would be a conflict in the partial assignment $\{r_F, g1, g5, e1, g2, g3, e2, e3, g6\}$, and a

Algorithm 4: BACKTRACK.

```
   INPUT  : An MCS π
   OUTPUT : NULL
1  level_1st, level_2nd ← 0
2  for each l ∈ π do
3      if DLEVEL(l) > level_1st then
4          level_1st ← DLEVEL(l)
5          level_2nd ← level_1st
6      else if level_1st > DLEVEL(l) > level_2nd then
7          level_2nd ← DLEVEL(l)
8  if level_1st == level_2nd then
9      CDCL.BACKTRACK(0)
10 else
11     CDCL.BACKTRACK(level_2nd)
```

block clause $\neg e2 \vee \neg e3$, corresponding to the new MCS, would be discovered.

Another way to backtrack is to go directly backtrack to the decision level 0, namely *nonchronological backtracking*, which can be seen as restarting solver under keeping $\mathcal{H}$. Although nonchronological backtracking avoids the inconsistency caused by chronological one, the "rude" way makes the computation of MCSs in each iteration likely to be independent, and the information obtained from the computed MCSs, such as the similarity between MCSs, cannot be fully used in further iterations.

We introduce the similarity between MCSs.

Definition 5: Let mcs_i and mcs_j be two MCSs. A product $P_{i,j}$ is a *model cover* of mcs_i and mcs_j, iff it fulfills the two following requirements: 1) $mcs_i \models P_{i,j}$; and 2) $mcs_j \models P_{i,j}$. $P_{i,j}$ is *minimal*, iff there does not exist $P'_{i,j} \supset P_{i,j}$, such that $P'_{i,j}$ is a model cover of mcs_i and mcs_j. The *similarity* between mcs_i and mcs_j is equal to $|P_{i,j}|$, if $P_{i,j}$ is the minimal model cover of them.

Intuitively, the similarity between MCSs essentially reflects the number of common literals in two MCSs, which means that there are several same search steps between the iterations, such as the same branchings and the same BCP based on these branchings. We illustrate the similarity between MCSs through an example.

Example 8: The MCSs of the fault tree [see Fig. 1(a)] contain $\{e1, e8\}$ and $\{e1, e3\}$. Because $e1 \wedge e8 \models e1$, $e1 \wedge e3 \models e1$ and $\{e1\}$ is minimal, $\{e1\}$ is their minimal model cover. Therefore, the similarity between $\{e1, e8\}$ and $\{e1, e3\}$ is 1.

We observe that, in practice, a fault tree often has millions of MCSs, and each MCS covers lots of cut sets; therefore, it potentially contains a large number of MCSs with a great similarity. Based on the above observation, we propose *jump-chronological backtracking*, which selects a reasonable decision level, called *pivot level*, to backtrack. Thanks to the great similarity between MCSs, it can reduce the repeat search steps.

The key to jump-chronological backtracking is to compute the pivot level. The pseudocode of this process is shown in Algorithm 4, where DLEVEL returns the decision level of a literal, and CDCL.BACKTRACK exploits the function to backtrack in CDCL. In our optimization, we empirically set the pivot level as the second highest decision level of literals in the last MCS (line 11). If the decision levels are the same, we backtrack to the decision level 0 (line 9). We illustrate the jump-chronological backtracking via an example.

Example 9 (Example 7 continued): After extracting an MCSs $\{e2, e3\}$, SATMCS backtracks to the decision level 2 because the decision level of $e2$ (2) is smaller than that of $e3$ (3). Obviously, the new iteration begins with the partial assignment $\{r_F, g1, g5, e1, g2, g3, e2\}$ at decision level 2 via jump-chronological backtracking, instead of $\{r_F, g1, g5\}$ at decision level 0 via nonchronological backtracking.

Compared with nonchronological backtracking, jump-chronological backtracking improves the efficiency for the cases that have lots of MCSs with great similarity. However, for the cases that have few MCSs with great similarity, jump-chronological backtracking may eventually go back to the decision level 0, which slightly increases the cost. We discuss the effect of jump-chronological backtracking in the experiment (see Section VI-C).

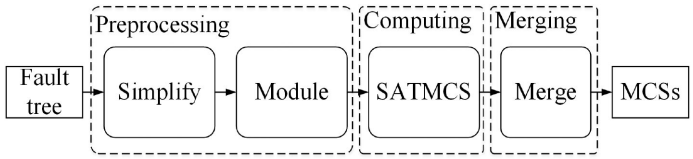

Fig. 3. Framework of SatFTA.

V. TOOL FOR COMPUTING MCSS BASED ON SATMCS

We have implemented a tool for computing all MCSs of a fault tree—SatFTA—that is powered by SATMCS, whose framework is shown in Fig. 3. The core of computation in SatFTA is SATMCS. SatFTA includes additional two parts for preprocessing and merging. In the whole process, SatFTA first reduces and decomposes the given fault tree to several relatively independent modules, i.e., subtrees, in the part of preprocessing. Then, SATMCS is used to compute the MCSs of every modules. In the end, SatFTA combines the MCSs of each modules in the part of merging to generate all the MCSs of the given fault tree.

The main purpose of preprocessing is to obtain the simplified and multimodule fault tree from the given one. The primary purpose of fault tree simplification is to simplify the fault tree structure by pruning redundant nodes or subtrees. The simplification reduces the size of the large fault tree, which reduces the calculation complexity. SatFTA relies on the following two structures [3] for simplification.

1) $\exists Q_{\hat{g}_i}, Q_{\hat{g}_j} \in \mathcal{Q}$ s.t. $g_j \in In_{\hat{g}_i}$ and $Op_{\hat{g}_i} = Op_{\hat{g}_j}$.
2) $\exists Q_{\hat{g}} \in \mathcal{Q}$ s.t. $|In_{\hat{g}}| = 1$.

For structure 1, SatFTA merges $\hat{g}_i$ and $\hat{g}_j$ into $\hat{g}_i$, i.e., $Q_{\hat{g}_i} = \langle Op_{\hat{g}_i}, In_{\hat{g}_i} \cup In_{\hat{g}_j}, g_i \rangle$. Subsequent gates of the same type are contracted to form a single gate. This reconstructs the fault tree as an alternating sequence of AND and OR gates. For structure 2, SatFTA directly replaces $\hat{g}$ with $In_{\hat{g}}$ in $\mathcal{Q}$. Based on

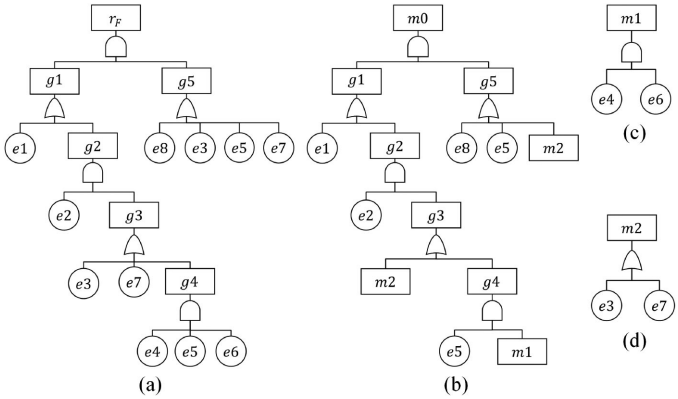

Fig. 4. (a)–(d) Example of preprocessing.

the algorithm [24], the fault tree is decomposed into relatively independent modules, i.e., the events in each module will not affect other modules. As a result of the decomposition, new module variables need to be introduced. In the part of computing, these module variables are considered as basic events in the module; thus, the MCSs of modules contain the module variables. To illustrate the part of preprocessing, consider the following example.

Example 10: The fault tree shown in Fig. 1(a) is reduced as Fig. 4(a). Due to $Op_{\hat{g5}} = Op_{\hat{g6}}$, $g6$ is replaced with $\{e3, e5, e7\}$ as input of $\hat{g5}$. Then, the fault tree is decomposed into $m0$, $m1$, and $m2$ modules, shown in Fig. 4(b)–(d), in which $m0$ is the main module including the top event r_F; $m1 = e4 \wedge e6$ and $m2 = e3 \vee e7$ are the new modules. Obviously, for the original fault tree shown in Fig. 1(a), $e4$ and $e6$ only occur in $m1$; $e3$ and $e7$ only occur in $m2$.

After obtaining all MCSs of each module, SatFTA directly replaces the module variables with the MCSs corresponding to the module starting from the main module. Recalling Example 10, we illustrate the parts of computing and merging.

Example 11: By SATMCS, the MCSs of $m0$ are $\{e1, e5\}$, $\{e2, e5, m1\}$, $\{e1, e8\}$, $\{e1, m2\}$, and $\{e2, m2\}$, in which $m1$ will be replaced by the MCSs of $m1$ – $\{e4, e6\}$, and $m2$ will be replaced by the MCSs of $m2$ – $\{e3\}$ or $\{e7\}$. After replacing the module variables, all MCSs of the original fault tree are $\{e1, e5\}$, $\{e2, e5, e4, e6\}$, $\{e1, e8\}$, $\{e1, e3\}$, $\{e1, e7\}$, $\{e2, e3\}$, and $\{e2, e7\}$.

VI. Experiment and Result

To evaluate the performance SATMCS, we compare SatFTA with a variety of commercial FTA tools over benchmarks from real-world applications and discuss the effects of different MCS extraction and backtracking methods. Specifically, we aim to answer the following research questions.

Question 1: Does SatFTA have advantages over other commercial tools?

Question 2: What is the effect of different algorithms for extracting MCSs in SATMCS?

Question 3: What is the effect of different backtracking strategies for SATMCS?

We answer the questions through three classes of experiments. The first one compares SatFTA, FaultTree+,[3] XFTA,[4] and RAM Commander (Commander for short)[5] on the efficiency of computing MCSs. In the second one, we study the difference between our algorithm for extracting MCSs and other alternatives. In the last one, we implement the variant based on nonchronological backtracking and compare it with jump-chronological backtracking.

The details of the benchmarks for the experiments are shown in Table I, which consist of 33 coherent fault trees from real industrial applications [25]. The first four columns present the information of the fault trees, including the name, the number of basic events (#E), gates (#G), and the number of MCSs (#MCS). The benchmarks are organized into different groups based on the prefix of their names that indicate the sources, e.g., *das** comes from aerospace industry, and *edf** comes from nuclear power industry.

For fairness, we use the following experimental settings.

1) Because the commercial tools equip preprocessing methods that are not published, we only exploit the basic preprocessing method [24].
2) The number of MCSs of some fault trees is large; therefore, for all tools, we do not consider IO time.

All experiments are performed on an Intel Xeon E5-1603 2.8 GHz, with 16 GB of memory. In the three classes experiments, we set the CPU time limit to 1 h and the memory limit to 8 GB. There are some special marks in the experiment results, i.e., "T. out" and "M. out" indicate the results out of the CPU time and memory limit, respectively; "Can't B." means that the case cannot be built; "−" denotes that the data cannot be counted. We mark the winner in **bold** in all tables.

A. Comparison Between SatFTA and Commercial Tools

FaultTree+, XFTA, and Commander are popular FTA commercial tools that have been used worldwide on safety-critical industries for analyzing large-scale fault trees. We compare SatFTA with these commercial tools on 33 coherent fault tree benchmarks to answer Question 1.

Table I shows the experimental results. SatFTA can solve 31 cases of the 33 fault trees, which include all the 21 cases solved by FaultTree+ and the ten cases solved by Commander. Compared with XFTA, XFTA can solve 28 cases, in which only $edf9204$ cannot be successfully computed by SatFTA. Although XFTA can solve $edf9204$ that SatFTA fails to compute, SatFTA works well in four cases ($das9209$, $edfpa14b$, $edfpa14o$, and $edfpa14q$). XFTA fails to compute the four cases because of the memory limit. Note that for the cases solved by SatFTA and the three commercial tools, they produce exactly the same set of MCSs.

1) Comparison With FaultTree+: It is observed that SatFTA generally computes faster than FaultTree+ for the

[3][Online]. Available: https://www.isograph.com/software/reliability-workbench

[4][Online]. Available: http://altarica-association.org/contents/xfta.html

[5][Online]. Available: http://www.reliability-safety-software.com/products/ram-comm-ander

TABLE I
EXPERIMENTAL RESULTS ON COMPUTING MCSS

Benchmark	#E	#G	#MCS	SatFTA		FaultTree+		XFTA		Commander	
				Time (sec.)	Mem. (MB)	Time (sec.)	Mem. (MB)	Time (sec.)	Mem. (MB)	Time (sec.)	Mem. (MB)
baobab3	80	107	24,386	5.27	14	M. out	M. out	**0.24**	**12**	Can't B.	Can't B.
chinese	25	36	392	**0.03**	**3**	0.80	89	**0.03**	4	0.04	30
das9201	122	82	14,217	**0.06**	**3**	1.30	91	0.07	7	10.45	26
das9202	49	36	27,778	**0.04**	**3**	1.68	88	0.14	7	75.31	66
das9203	51	30	16,200	**0.02**	**3**	0.77	86	0.07	7	455.91	74
das9204	53	30	16,704	**0.02**	**3**	0.67	89	0.09	7	67.32	46
das9205	51	20	17,280	**0.03**	**3**	0.85	86	0.07	7	0.04	44
das9206	121	112	19,518	0.15	**3**	2.10	92	**0.08**	7	219.31	85
das9207	276	324	25,988	1.03	**4**	3.53	100	**0.14**	9	Can't B.	Can't B.
das9208	103	145	8,060	0.56	**4**	1.50	96	**0.07**	5	Can't B.	Can't B.
das9209	109	73	8.2E10	**0.06**	**3**	0.70	96	M. out	M. out	T. out	T. out
edf9201	183	132	579,720	**0.30**	**4**	3,279.00	93	1.50	95	Can't B.	Can't B.
edf9202	458	435	130,112	0.72	**3**	Can't B.	Can't B.	**0.45**	26	Can't B.	Can't B.
edf9203	362	475	20,807,446	164.27	**168**	M. out	M. out	**80.56**	2,782	Can't B.	Can't B.
edf9204	323	375	32,580,630	T. out	T. out	M. out	M. out	**160.45**	**4,447**	Can't B.	Can't B.
edf9205	165	142	21,308	0.20	**3**	376.43	248	**0.10**	8	Can't B.	Can't B.
edf9206	240	362	–	T. out	T. out	M. out	M. out	M. out	M. out	Can't B.	Can't B.
edfpa14b	311	290	105,955,422	**1,851.83**	**746**	M. out	M. out	M. out	M. out	Can't B.	Can't B.
edfpa14o	311	173	105,927,244	**1,348.04**	**644**	M. out	M. out	M. out	M. out	Can't B.	Can't B.
edfpa14p	124	101	415,500	2,012.91	751	M. out	M. out	**6.94**	**120**	Can't B.	Can't B.
edfpa14q	311	194	105,950,670	**1,558.90**	**660**	M. out	M. out	M. out	M. out	M. out	M. out
edfpa14r	106	132	380,412	1,802.63	722	M. out	M. out	**6.70**	**119**	M. out	M. out
edfpa15b	283	249	2,910,473	**12.01**	**30**	M. out	M. out	12.61	402	Can't B.	Can't B.
edfpa15p	276	324	27,870	12.57	31	1,115.12	212	**0.50**	**12**	Can't B.	Can't B.
edfpa15r	88	110	26,549	12.42	24	89.68	142	**0.37**	**12**	T. out	T. out
elf9601	145	242	151,348	270.02	28	Can't B.	Can't B.	**0.91**	**26**	Can't B.	Can't B.
ftr10	175	94	305	0.12	**3**	1.16	91	**0.02**	5	0.45	98
isp9602	116	122	5,197,647	**0.24**	**4**	16.68	94	31.72	1,033	M. out	M. out
isp9603	91	95	3,434	0.12	**3**	1.61	93	**0.02**	4	Can't B.	Can't B.
isp9604	215	132	746,574	**0.18**	**3**	1.14	88	2.01	103	Can't B.	Can't B.
isp9606	89	41	1,776	0.05	**3**	0.98	90	**0.02**	4	0.11	38
isp9607	74	65	150,436	**0.08**	**3**	1.80	94	0.60	42	T. out	T. out
jbd9601	533	315	14,007	1.20	**4**	2.87	109	**0.13**	9	1,616.00	358

21 cases solved by both of them. In particular, for $edf9201$, $edf9205$, and $edfpa15p$, SatFTA is significantly faster than FaultTree+ about two orders of magnitude. Moreover, the memory used by SatFTA is also significantly less than that used by FaultTree+ in these 21 cases—SATMCS consumes about one order of magnitude less memory than FaultTree+ in 19 cases.

2) Comparison With XFTA*:* The performance of XFTA is the best among the three commercial tools. Overall, XFTA and SatFTA are comparable in time used, where XFTA wins in 15 instances and SatFTA wins in 18 instances. In memory used, it is obvious that SatFTA has much better performance in most cases (25 instances)—it consumes at least one order of magnitude less memory than XFTA in some cases, such as $edf9201$, $edf9203$, $edfpa15b$, and $isp9602$. Besides, although SatFTA takes over 1000-s CPU time to compute $edfpa14b$, $edfpa14o$, and $edfpa14q$, their memory used is not large (all less than 0.8 GB). XFTA fails to compute the three instances due to large memory usage (out of 8 GB). Consequently, SatFTA uses less memory than XFTA, thanks to the SAT-based method.

3) Comparison With Commander*:* For other tools, as can be seen from Table I, most of the cases cannot be solved by Commander because it is unable to construct a valid structure for these cases. Moreover, for all the ten cases solved by Commander, SatFTA uses significantly less CPU time and memory than Commander.

In general, the experimental results look encouraging, which shows that SatFTA has excellent ability in computing MCSs compared with the commercial tools. It can solve large-scale fault trees without significantly sacrificing efficiency with respect to CPU time and memory usage.

B. Evaluation of Extracting MCSs

The features of EXTRACTMCS include the following: (i) EXTRACTMCS tests the necessity of a basic event by incrementally analyzing failure propagation in an LPG induced by the cut set; and (ii) it conducts a linear number of tests for deciding the necessity of basic events in a cut set. In this experiment, we design two variants of SATMCS about the features to evaluate the effect of Extracting MCSs (Question 2).

Considering feature (i), such a test can be done by a direct call to a SAT solver. We implement a variant of SATMCS, namely SATMCS-Call, which replaces the LPG-based test in EXTRACTMCS with direct calling ZChaff [26]. Specifically, given a fault tree F and a cut set $\pi^+ = \pi^+_{E_F} \cup \pi^+_{G_F}$, if

TABLE II
EVALUATION OF EXTRACTING MCSS

Benchmark	SATMCS		SATMCS-Call		SATMCS-QX	
	Time (sec.)	Mem. (MB)	Time (sec.)	Mem. (MB)	Time (sec.)	Mem. (MB)
baobab3	**5.27**	**14**	29.36	15	59.29	**14**
chinese	0.03	**3**	**0.02**	**3**	0.03	**3**
das9201	**0.06**	**3**	0.07	**3**	0.08	**3**
das9202	0.04	**3**	**0.03**	**3**	0.05	**3**
das9203	0.02	**3**	**0.01**	**3**	**0.01**	**3**
das9204	**0.02**	**3**	**0.02**	**3**	**0.02**	**3**
das9205	0.03	**3**	**0.01**	**3**	**0.01**	**3**
das9206	**0.15**	**3**	0.41	**3**	0.59	4
das9207	**1.03**	**4**	5.49	**4**	4.02	**4**
das9208	**0.56**	**4**	2.72	**4**	2.98	**4**
das9209	0.06	**3**	**0.05**	**3**	**0.05**	**3**
edf9201	**0.30**	**4**	0.83	**4**	1.41	**4**
edf9202	**0.72**	**3**	1.15	4	0.98	4
edf9203	**164.27**	168	1,022.76	**141**	2,155.82	160
edf9204	T. out	T. out	T. out	T. out	T. out	T. out
edf9205	**0.20**	**3**	0.41	4	0.61	4
edf9206	T. out	T. out	T. out	T. out	T. out	T. out
edfpa14b	**1,851.83**	**746**	T. out	T. out	T. out	T. out
edfpa14o	**1,348.04**	**644**	3,067.42	649	T. out	T. out
edfpa14p	**2,012.91**	**751**	T. out	T. out	T. out	T. out
edfpa14q	**1,558.90**	**660**	3,146.99	667	T. out	T. out
edfpa14r	**1,802.63**	**722**	3,562.77	736	T. out	T. out
edfpa15b	**12.01**	**30**	92.90	**30**	121.81	**30**
edfpa15p	**12.57**	**31**	81.53	32	110.93	32
edfpa15r	**12.42**	**24**	88.39	25	124.69	25
elf9601	**270.02**	**28**	392.80	29	462.38	29
ftr10	**0.12**	**3**	0.13	**3**	0.12	**3**
isp9602	**0.24**	**4**	0.64	**4**	1.30	**4**
isp9603	**0.12**	**3**	0.29	**3**	0.31	**3**
isp9604	**0.18**	**3**	0.32	**3**	0.45	**3**
isp9606	0.05	**3**	**0.03**	**3**	**0.03**	**3**
isp9607	**0.08**	**3**	0.17	**3**	0.31	**3**
jbd9601	**1.20**	**4**	8.24	**4**	4.10	**4**

$(\bigwedge_{l_i \in \pi^+_{E_F} \wedge l_i \neq l} l_i) \wedge (\neg F)$ is satisfiable (respectively, unsatisfiable), then l is necessity (respectively, unnecessary) for the cut set π.

We compare SATMCS and SATMCS-Call on the benchmarks as above. Table II shows the results. SATMCS-Call computes 29 cases within the limits. SATMCS successfully computes two cases, $edfpa14b$ and $edfpa14p$, more than SATMCS-Call. For the 29 cases that both solvers can solve, SATMCS takes about less than half of the time used by SATMCS-Call in most cases, which shows the advantage of the LPG-based approach. Particularly, for *edf9203*, *edfpa15b*, *edfpa15p*, and *edfpa15r*, the CPU time used by SATMCS is less than the one-sixth of that by SATMCS-Call. In contrast, SATMCS consumes slightly less memory than SATMCS-Call in most cases, but not significant. SATMCS-Call needs to repeatedly call the SAT solver in every testing, the search space of which is not reduced. Therefore, the cost of calling the SAT solver and the huge search space seriously affect the efficiency of SATMCS-Call.

These results indicate that the method to extract MCSs based on LPG focuses on the partial constraints of the fault tree, while the method based on SAT calling considers all constraints each time. Moreover, the incrementally updating LPG can explicitly leverage the information produced in the process although this information can theoretically be implicitly captured by incremental SAT callings.

An alternative to feature (ii) is to use the QuickXplain algorithm [27], [28], which, based on recursively splitting the cut set, requires exponentially less tests in optimal case than the linear approach. However, when most of the events in a cut set are necessary, QuickXplain may require more tests than the linear one. QuickXplain requires making $\mathcal{O}((|\pi'|-1)+|\pi'|\lg\frac{|\pi|}{|\pi'|})$ queries to the necessary testing to extract an MCS, where π is a cut set and π' is an MCS. Note that QuickXplain is an asymptotically optimal algorithm for the problem. However, if $\frac{|\pi|}{|\pi'|} \in [1,2]$, EXTRACTMCS only needs $\mathcal{O}(|\pi|)$ queries, which is better than QuickXplain.

In our case, SATMCS uses CDCL to search for cut sets. It turns out that the cut sets found by CDCL are very close to the MCSs extracted from them—according to our statistics from the benchmarks, the average ratio of the number of basic events in a cut set to that in the corresponding MCS is 1.33. In this case, we expect better performance of extracting MCS using the linear approach implemented in EXTRACTMCS than using QuickXplain.

We implement another variant of SATMCS using QuickXplain, called SATMCS-QX, and compare with SATMCS. Table II illustrates that, with comparable memory usage, SATMCS takes about one-tenth of the CPU time used by SATMCS-QX in most cases, which confirms our analysis above. Particularly, $edfpa14o$, $edfpa14q$, and $edfpa14r$ are even unsolved by SATMCS-QX within the limit.

Through this experiment, it can be concluded that, for fault trees, the LPG-based necessity testing and the linear approach exhibit better performance.

C. Influence of Backtracking

In this experiment, we study the effect of the two backtracking methods on SATMCS. SATMCS-JC uses jump-chronological backtracking, which selects a pivot level to backtrack. In contrast, SATMCS-NC equips with nonchronological backtracking, i.e., direct backtracking to the decision level 0. We answer Question 3 by comparing their results (see Table III) of benchmarks as above.

Overall, SATMCS-JC not only successfully computes the 26 cases that SATMCS-NC can solve within the limits, but also works well for $edfpa14b$, $edfpa14o$, $edfpa14p$, $edfpa14q$, and $edfpa14r$, where the number of MCSs is more than 30 million. Particularly, $edfpa14b$, $edfpa14o$, and $edfpa14q$ even include more than 100 million MCSs. In addition to being able to solve more cases, jump-chronological backtracking has greatly improved in the speed for computing $edf9203$, which is 73% higher than that for the nonchronological one, but has no significant improvement for other cases. We believe that this is related to the similarity between MCSs.

Let mcs_i be an MCS that is computed in the ith iteration and mcs_{i+1} be an MCS that is computed in the $i+1$th iteration. In order to study the similarity of MCSs, in SATMCS-JC, we

TABLE III
INFLUENCE OF BACKTRACKING

Benchmark	SATMCS-NC		SATMCS-JC		ψ
	Time (sec.)	Mem. (MB.)	Time (sec.)	Mem. (MB.)	
baobab3	8.87	16	**5.27**	**14**	0.92
chinese	**0.02**	**3**	0.03	**3**	0.73
das9201	**0.05**	**3**	0.06	**3**	0.74
das9202	**0.02**	**3**	0.04	**3**	0.58
das9203	**0.01**	**3**	0.02	**3**	0.80
das9204	**0.02**	**3**	**0.02**	**3**	0.83
das9205	**0.01**	**3**	0.03	**3**	0.00
das9206	**0.16**	**3**	0.15	**3**	0.75
das9207	**0.92**	**4**	1.03	**4**	0.67
das9208	0.58	**4**	**0.56**	**4**	0.83
das9209	**0.05**	**3**	0.06	**3**	0.00
edf9201	0.33	**4**	**0.30**	**4**	0.77
edf9202	0.74	**3**	**0.72**	**3**	0.51
edf9203	604.62	191	**164.27**	**168**	0.95
edf9204	T. out	T. out	T. out	T. out	–
edf9205	**0.20**	**3**	**0.20**	**3**	0.76
edf9206	T. out	T. out	T. out	T. out	–
edfpa14b	T. out	T. out	**1,851.83**	**746**	0.97
edfpa14o	T. out	T. out	**1,348.04**	**644**	0.96
edfpa14p	T. out	T. out	**2,012.91**	**751**	0.97
edfpa14q	T. out	T. out	**1,558.90**	**660**	0.97
edfpa14r	T. out	T. out	**1,802.63**	**722**	0.97
edfpa15b	20.93	**28**	**12.01**	30	0.94
edfpa15p	22.53	**31**	**12.57**	**31**	0.93
edfpa15r	21.70	30	**12.42**	**24**	0.95
elf9601	284.51	34	**270.02**	**28**	0.96
ftr10	**0.10**	**3**	0.12	**3**	0.53
isp9602	0.26	**4**	**0.24**	**4**	0.97
isp9603	0.13	**3**	**0.12**	**3**	0.69
isp9604	0.20	**3**	**0.18**	**3**	0.82
isp9606	**0.02**	**3**	0.05	**3**	0.65
isp9607	0.09	**3**	**0.08**	**3**	0.87
jbd9601	1.23	**4**	**1.20**	**4**	0.67

introduce the ratio of the similarity between mcs_i and mcs_{i+1} to the size of mcs_i, denoted as ρ. In the experiment, we assume that if $\rho \geqslant 0.5$, mcs_i and mcs_{i+1} are similar. We report the ratio of the number of cases where $\rho \geqslant 0.5$ to the total number of iterations (ψ in Table III). On the one hand, the values of ψ are greater than 0.5 in most of the cases computed by SATMCS-JC. That means there is at least 50% chance of computing a new MCS that is similar to the previous MCS. Particularly, in $edf9203$, $edfpa14b$, $edfpa14o$, $edfpa14p$, $edfpa14q$, and $edfpa14r$, the ψ even reaches 90%, for which SATMCS-JC outperforms SATMCS-NC significantly. On the other hand, for the cases whose ψ is less than 90%, ψ even is 0 for $das9205$ and $das9209$; though SATMCS-JC slightly increases backtracking cost, SATMCS-JC is still comparable with SATMCS-NC.

In conclusion, the jump-chronological backtracking can significantly improve the solution efficiency of the cases with the great similarity between MCSs, while it does not have a greatly poor impact on the cases with other cases.

VII. RELATED WORK

The classical methods for computing MCSs are based on Boolean manipulation, BDD, and others. The methods based on Boolean manipulation include the top-down [6] and bottom-up [5] algorithms. This class exploits operations of Boolean algebra to compile nonclausal Boolean formulas of fault trees into its DNF. Because of the complex relationship of variables, the items of DNF are only the cut sets. The minimization of cut sets is inevitable to obtain MCSs, which is often computationally intense.

The BDD-based methods encode fault trees using BDDs that can simply represent fault trees and then use the minimization method to extract MCSs. Note that, sometimes, the building BDD fails because of the exponential memory cost. The BDD-based methods are first introduced by Coudert and Madre [7] as well as Rauzy and Dutuit [8], where they proposed metaproduct to encode a fault free into the MCSs directly.

A number of alternative approaches have been proposed over the years, also for either coherent or noncoherent fault trees. Tang and Dugan [9] proposed using zero-suppressed BDD (ZBDD) to compute MCSs. A large number of MCSs can be encoded in a compact space. Codetta-Raiteri [13] introduced parametric fault tree, which provides a compact models for the fault trees with a large number of shared subtrees. This method performs qualitative and quantitative analysis without repeating the analysis for each shared subtrees. Jung *et al.* [10], [11] provided a truncation method for coherent fault trees, which considers about significant MCSs when building ZBDD and obtains TZBDD. Contini and Matuzas [12] have extended the same approach to noncoherent fault trees. Remenyte-Prescott and Andrews [14] introduced the TDDs to compute MCSs for noncoherent fault trees. Note that this work concludes that the TDD-based method is more efficient compared with some BDD-based methods. Maio *et al.* [29] proposed a hierarchical differential evolution algorithm to identify MCSs, which transforms the problem into a hierarchical optimization problem based on genetic algorithm. Although our method has a good performance for the coherent fault trees currently, an ingenious method can make our method work in noncoherent fault trees, which is the focus of our future research.

Some optimization methods have been proposed for special fault trees. The algorithm provided by Carrasco and Sune [30] is useful for the fault trees with large scale size but few cut set. If the fault trees include the voting gates with a number of inputs, Xiang *et al.* [31] introduced the concept of minimal cut vote in MCSs to describe the special relationship of elements. The complexity of this method is linear with the number of inputs of voting gates, and it has a well-visible representation of MCSs.

Computing MCSs of fault trees can be considered as a special case of enumerating prime implicants of Boolean formulas [6]. Enumerating prime implicants is the foundation of minimization of Boolean formulas and has been applied in many fields, such as bioinformatics [32], knowledge representation and reasoning [33], multiagent systems [34], etc. Due to the significance and intractability of this problem, the methods is widely researched, such as 0–1 integer linear programming [35], the modification of modern SAT solvers [36], well-known DPLL procedure [37], etc. Among them, Jabbour *et al.* [38] focused on the idea of reformulating original Boolean formulas with "minimize" constraints. A prime implicant of original Boolean formulas directly corresponds to a model of reformulating

constraints. It should be pointed out that these methods are only computing prime implicants of the clausal Boolean formulas. For nonclausal Boolean formulas converted into a CNF, the prime implicants obtained by these methods contain redundant elements.

Based on this work, Previti *et al.* [39] propose algorithms for finding prime implicants of nonclausal formulas based on iterative SAT solving, which is the closest to ours. Their approach uses dual-rail encoding [40], [41] to ensure the computation of the complete set of prime implicants and can be adapted for computing MCSs of fault trees. This approach, however, uses a SAT solver as a black box; therefore, it has to operate on full satisfying assignments returned from the solver. By contrast, our approach extends CDCL with the algorithms for computing MCSs; hence, we are able to work on partial assignments within the CDCL procedure and discover MCSs earlier. Moreover, the approach in [39] relies on the QuickXplain algorithm for extracting prime implicants from satisfying assignments. This algorithm, as discussed in Section VI-B, may require more number of satisfiability queries than our approach.

Our work is related to the All-SAT problem [42], [43] that computes all the satisfying assignments of a given Boolean formula. This problem has been investigated in the context of model checking and predicate abstraction. Main algorithms for solving the problem are based on SAT [42]–[45] and BDD [46], [47]. Although the All-SAT problem is similar to the problem of computing MCSs, which finds all the assignments that satisfy constraints, it is distinct that the former is computationally less expensive than the latter, since the satisfying assignments are not necessary to be minimized.

Since both the All-SAT problem and the computation of MCSs involve enumerating all the results, we can learn the optimizations from the All-SAT problem. Yu *et al.* [43] obtain a special block clause covering more satisfiable assignments by extending a satisfiable solution. Obviously, in order to cover the most satisfiable assignments, this special block clause is obtained from the MCSs. However, taking the cost of the extracting MCSs into account, the All-SAT problem often adopts some compromise methods. Furthermore, Yu *et al.* [43] propose two solution of the special block clause—minimal satisfying cubes and decision-based minimal satisfying cubes, which can be considered as an approximate MCS. The idea of iteratively refining the MCSs through an approximate solution can be used for computing MCSs, which may balance cost of partial and global CPU time. Huang and Darwiche [48] encode BDD of the input exploiting the DPLL's decision process, from which it is easy to decode all the models of the input. The disadvantage, however, is that the operations of the cut set cache need to be called frequently. The cut set cache is used to conduct weaker equivalence test, which reduces the cost caused by the merging operations of the nodes in the BDD. Based on the cut set cache, Toda and Tsuda [47], learning more information contained in DPLL, propose the finer cut set cache, which cuts down the frequency of cache operations. Due to BDD with the efficient coding Boolean formulas and DPLL with the flexible search, the comprehensive application of BDD and DPLL is also worth thinking for computing MCSs.

VIII. Conclusion

Computing all MCSs is an important problem in FTA, but it is still a challenge to *efficiently* compute *all* MCSs. Given the inherent intractability of computing MCSs, developing new methods over different paradigms remains to be an interesting research direction. In this article, we presented SATMCS, a novel method for computing MCSs based on SAT, which searches for MCSs by continuous iteration extending CDCL. The major features of SATMCS include LPG-based MCS extraction, clause learning with dynamic deletion, as well as jump-chronological backtracking based on the similarity between MCSs, which significantly speed up the searching process. Furthermore, we implemented a tool for computing MCSs of a fault tree—SatFTA that can be powered by SATMCS. We assessed the performance of SATMCS in this article through three classes of experiments. Thanks to the dedicated encoding, extension of the CDCL and efficient analysis algorithms that we developed for fault trees, SATMCS possesses excellent ability to compute all MCSs. Experimental results show that SATMCS has better performance, specifically, on memory usage, though SatFTA is only a prototype. To the best of our knowledge, this is the first report that is targeted for computing MCSs in FTA based on SAT.

Currently, our method is only applicable to coherent fault trees. We would like to extend SATMCS for handling noncoherent ones in the future. We also notice that our approach can greatly benefit from the development of the SAT solver. Combined with the most efficient SAT solver based on the CDCL, such as MiniSAT [21] and Maple [49], we believe that SATMCS will perform better. Furthermore, heuristic methods to enumerate prime implicants and All-SAT problems will be applied for SATMCS optimization. For instance, the number of literals making up prime implicants is different, which quantifies the degree of contribution from an assigned variable to the progress of searching. With preferentially branch literals that possibly appear in prime implicants, it may be helpful for searching and extraction MCSs.

Appendix
Proof of SATMCS

In this appendix, we prove that SATMCS can compute all MCSs of a fault tree. Before that, we first need to prove that IsNecessary (Algorithm 3) aims to test the necessity of the $v \in \pi^+_{E_F}$. Second, we show that ExtractMCS (Algorithm 2) can return an MCS from an LPG.

Lemma 2: In IsNecessary, if v is a necessary literal of $\pi^+_{E_F}$, then IsNecessary returns `true`; otherwise, it returns `false`.

Proof: In order to prove this lemma, we prove following claims.

1) If $v \mapsto r_F$ holds, then v is a necessary literal; otherwise, v is unnecessary (see Lemma 1).
2) $\forall v \mapsto l$, l will add to $eventSet$.

We first prove Claim 1. Because the LPG is built based on the structural characteristics, all the influence edges from $\pi^+_{E_F}$ to r_F are in the LPG. If $v \mapsto r_F$, then removing v from the LPG leads to removing r_F from the LPG (see Property 2), which means that

there is not a influence edge from $\pi^+_{E_F}$ to r_F, i.e., $\pi^+_{E_F} \setminus v \models F$ is `false`. Therefore, v is a necessary literal.

Now, we prove Claim 2 by inductive hypothesis as follows.

1) *eventSet* is initialized to $\{v\}$ (line 2) and $v \in E_F$. Therefore, $v \mapsto v$ holds.
2) In the first iteration (lines 3–11), for every $(S, l_0) \in R$, where $v \in S$, if $|S| \geq 2$, which means v is one of input of `AND`, then $v \mapsto l_0$ is true; else if v is a input of `OR` where if v is the only input of `OR`, then $v \mapsto l_0$; otherwise, there is no literal influenced by v. In the first two cases, l_0 is added to *eventSet*; otherwise, there is only element v in *eventSet*. Claim 2 has been proved.
3) We suppose l_k is added to *eventSet* in the kth iteration, which means $v \mapsto l_k$. Obviously, Claim 2 holds at this time.
4) Without loss of generality, l_k is considered in the $(k+1)$th iteration. Proceeding as in the first iteration, we can show that if $l_k \mapsto l_{k+1}$, l_{k+1} is added to *eventSet*, which means $v \mapsto l_{k+1}$; otherwise, because of the hypothesis and *eventSet* is not changed, Claim 2 has been proved.

Consequently, if $v \mapsto r_F$ holds, which means v is a necessary literal (Claim 1), then r_F will be add to $eventSet$ (Claim 2) and IsNecessary return `true` (line 9); otherwise, IsNecessary reduces the LPG and return `false` (line 13). ■

Lemma 3: ExtractMCS (see Algorithm 2) extracts an MCS from the final LPG.

Proof: In order to prove this lemma, we prove following claims.

1) There is at least a necessary literal in the final LPG.
2) All literals in the final LPG are necessary.

For Claim 1, our purpose is to show that the invariant is $\exists v \in \pi^+_{E_F}, v \mapsto r_F$ in the updating LPG (V, R'). Based on the building of LPG, there at least exists $v_e \mapsto r_F$ in LPG, where $v_e \in \pi^+_{E_F}$.

1) Initially, we consider the necessity of literal $v_0 \in \pi^+_{E_F}$. If $v_0 \mapsto r_F$, then IsNecessary does not update LPG, where there at least exist $v_0 \mapsto r_F$ and $v_e \mapsto r_F$ in original LPG; otherwise, $v_e \mapsto r_F$ holds after updating (V, R').
2) Suppose after calling IsNecessary to test the necessity of $v_k \in \pi^+_{E_F}$, there at least exists $v_e \mapsto r_F$ in (V, R').
3) Considering the necessity of $v_{k+1} \in \pi^+_{E_F}$, if $v_{k+1} \mapsto r_F$, the LPG is not updated, where $v_{k+1} \mapsto r_F$ and $v_e \mapsto r_F$ hold; otherwise, IsNecessary updates (V, R'), where $v_e \mapsto r_F$ holds.

Therefore, there is at least $v_e \mapsto r_F$ in all updating LPGs.

For Claim 2, without loss of generality, we suppose that we have $v_i \mapsto r_F$ in the ith iteration. Every literal v_j in the path $v_i \mapsto r_F$ will not be removed from the updating LPG, since $v_j \mapsto r_F$ always holds in the updating LPG. Therefore, $v_i \mapsto r_F$ still holds in the final LPG.

Consequently, ExtractMCS does not remove the necessary literals (Claim 1) and leave unnecessary literals (Claim 2). ■

Now, we prove Theorem 1: `SATMCS` terminates, iff it obtains all the MCSs of the coherent fault tree F.

Proof: On the one hand, we show that `SATMCS` continues until it obtains all the MCSs. Without loss of generality, we suppose that all the MCSs have been obtained but an MCS $\pi = \{x_1, x_2, \ldots, x_m\}$. We extend π to a full assignment $p = \{x_1, x_2, \ldots, x_m, \neg x_{m+1}, \neg x_{m+2}, \ldots, \neg x_n\}$. `SATMCS` will continue, if we show that $p \models \mathcal{H}$ holds. Because π is an MCS of F and F is coherent, it follows that $p \models \mathcal{F}$. For $\forall m \in \mathcal{M}$, there at least exists a positive literal l s.t. $l \in m \wedge l \notin \pi$, which follows that $\neg l \in p$, so $p \models \mathcal{B}$ holds. Clearly, $p \models \mathcal{C}$ is true. In summary, p is a satisfiable assignment for $\mathcal{H}$, so `SATMCS` does not terminate.

On the other hand, we show that if all the MCSs are computed, `SATMCS` terminates. It is obvious that the model of $\mathcal{H}$ is finite, so we only show that the results of every iterations are different. Suppose an MCS $\pi_i = \{x_1, x_2, \ldots, x_m\}$ is obtained in the ith iteration and the block clause $b = \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_m$ is added to $\mathcal{B}$. In the jth iteration after ith, without loss of generality, we consider a partial assignment $p = \{x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m\}$. Because of the unit clause rule, `SATMCS` must set $x_k =$ `false` to satisfy b. It can be shown that the MCS obtained in the jth iteration must be different from that obtained from the ith, and their corresponding block clauses are also different. To sum up, there will be no assignment to make $\mathcal{F} \wedge \mathcal{B}$ satisfiable at the same time at the end, where `SATMCS` terminates. ■

Acknowledgment

The authors would like to thank Jian Zhang for discussion on this article, Junhua Song for her help with the experiments, and anonymous referees for helpful comments.

References

[1] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," Nuclear Regulatory Commission, Washington, DC, USA, Tech. Rep. NUREG-0492, 1981.
[2] M.-A. Esteve, J.-P. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, "Formal correctness, safety, dependability, and performance snalysis of a satellite," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1022–1031.
[3] S. Chen, J. Wang, J. Wang, F. Wang, and L. Hu, "Efficient reduction and modularization for large fault trees stored by pages," *Ann. Nucl. Energy*, vol. 90, pp. 22–25, 2016.
[4] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Comput. Sci. Rev.*, vol. 15, pp. 29–62, 2015.
[5] S. N. Semanderes, "'ELRAFT' A computer program for the efficient logic reduction analysis of fault trees," *IEEE Trans. Nucl. Sci.*, vol. NS-18, no. 1, pp. 481–487, Feb. 1971.
[6] J. B. Fussell and W. E. Vesely, "A new methodology for obtaining cut sets for fault trees," *Trans. Amer. Nucl. Soc.*, vol. 15, no. 1, pp. 324–326, 1972.
[7] O. Coudert and J. C. Madre, "Fault tree analysis: 10^{20} prime implicants and beyond," in *Proc. 39th Annu. Rel. Maintainability Symp.*, 1993, pp. 240–245.
[8] A. Rauzy and Y. Dutuit, "Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia," *Rel. Eng. Syst. Saf.*, vol. 58, no. 2, pp. 127–144, 1997.
[9] Z. Tang and J. B. Dugan, "Minimal cut set/sequence generation for dynamic fault trees," in *Proc. 50th Annu. Rel. Maintainability Symp.*, 2004, pp. 207–213.
[10] W. S. Jung, S. H. Han, and J. Ha, "A fast BDD algorithm for large coherent fault trees analysis," *Rel. Eng. Syst. Saf.*, vol. 83, no. 3, pp. 369–374, 2004.
[11] W. S. Jung, S. H. Han, and J.-E. Yang, "Fast BDD truncation method for efficient top-event probability calculation," *Nucl. Eng. Technol.*, vol. 40, no. 40, pp. 571–580, 2008.
[12] S. Contini and V. Matuzas, "Reduced ZBDD construction algorithms for large fault tree analysis," in *Reliability, Risk and Safety—Back to the Future*. Abingdon, U.K.: Taylor & Francis, 2010, pp. 898–906.

[13] D. Codetta-Raiteri, "BDD based analysis of parametric fault trees," in *Proc. 52th Annu. Rel. Maintainability Symp.*, 2006, pp. 442–449.

[14] R. Remenyte-Prescott and J. Andrews, "Analysis of non-coherent fault trees using ternary decision diagrams," *Proc. Inst. Mech. Eng. O, J. Risk Rel.*, vol. 222, no. 2, pp. 127–138, 2008.

[15] A. Rauzy, "Mathematical foundations of minimal cutsets," *IEEE Trans. Rel.*, vol. 50, no. 4, pp. 389–396, Dec. 2001.

[16] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability*, vol. 185. Amsterdam, The Netherlands: IOS Press, 2009.

[17] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. ACM Symp. Theory Comput.*, 1971, pp. 151–158.

[18] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 1999, pp. 193–207.

[19] C. Castellini, E. Giunchiglia, and A. Tacchella, "SAT-based planning in complex domains: Concurrency, constraints and nondeterminism," *Artif. Intell.*, vol. 147, nos. 1/2, pp. 85–117, 2003.

[20] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI-C programs using SAT," *Formal Methods Syst. Des.*, vol. 25, nos. 2/3, pp. 105–127, 2004.

[21] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proc. 6th Int. Conf. Theory Appl. Satisfiability Testing*, 2003, pp. 502–518.

[22] G. S. Tseitin, "On the complexity of derivations in the propositional calculus, " in, *Automation of Reasoning*. New York, NY, USA: Springer, 1983.

[23] J. P. Marques-Silva, and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, May 1999.

[24] T. Kohda, E. J. Henley, and K. Inoue, "Finding modules in fault trees," *IEEE Trans. Rel.*, vol. 38, no. 2, pp. 165–176, Jun. 1989.

[25] "Aralia faulttrees," [Online]. Available: https://www.itu.dk/research/cla/externals/clib/, Accessed: Jul. 11, 2019.

[26] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient SAT solver," in *Proc. 7th Int. Conf. Theory Appl. Satisfiability Testing*, 2004, pp. 360–375.

[27] U. Junker, "Preferred explanations and relaxations for over-constrained problems," in *Proc. 19th Int. Conf. Artif. Intell.*, 2004, pp. 167–172.

[28] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Proc. 7th Int. Conf. Formal Methods Comput.-Aided Des.*, 2007, pp. 173–180.

[29] F. D. Maio, S. Baronchelli, and E. Zio, "Hierarchical differential evolution for minimal cut sets identification: Application to nuclear safety systems," *Eur. J. Oper. Res.*, vol. 238, no. 2, pp. 645–652, 2014.

[30] J. A. Carrasco and V. Sune, "An algorithm to find minimal cuts of coherent fault-trees with event-classes, using a decision tree," *IEEE Trans. Rel.*, vol. 48, no. 1, pp. 31–41, Mar. 1999.

[31] J. Xiang *et al.*, "Efficient analysis of fault trees with voting gates," in *Proc. 22nd Int. Symp. Softw. Rel. Eng.*, 2011, pp. 230–239.

[32] V. A. Na, P. V. Milreu, L. Cottret, A. Marchetti-Spaccamela, L. Stougie, and M. F. Sagot, "Algorithms and complexity of enumerating minimal precursor sets in genome-wide metabolic networks," *Bioinformatics*, vol. 28, no. 19, pp. 2474–2483, 2012.

[33] M. Cadoli and F. M. Donini, *A Survey on Knowledge Compilation*. Amsterdam, The Netherlands: IOS Press, 1997.

[34] M. Slavkovik, "A judgment set similarity measure based on prime implicants," in *Proc. Int. Conf. Auton. Agents Multi-Agent Syst.*, 2014, pp. 1573–1574.

[35] C. Pizzuti, "Computing prime implicants by integer programming," in *Proc. 8th IEEE Int. Conf. Tools Artif. Intell.*, 1996, pp. 332–336.

[36] D. Deharbe, P. Fontaine, D. L. Berre, and B. Mazure, "Computing prime implicants," in *Proc. Formal Methods Comput.-Aided Des.*, 2013, pp. 46–52.

[37] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *Proc. 10th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2004, pp. 31–45.

[38] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi, "Enumerating prime implicants of propositional formulae in conjunctive normal form," in *Proc. 14th Eur. Workshop Logics Artif. Intell.*, 2014, pp. 152–164.

[39] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva, "Prime compilation of non-clausal formulae," in *Proc. 24th Int. Joint Conf. Artif. Intell.*, 2015, pp. 1980–1987.

[40] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A compiled simulator for MOS circuits," in *Proc. 24th ACM/IEEE Des. Autom. Conf.*, 1987, pp. 9–16.

[41] J.-W. Roorda and K. Claessen, "A new SAT-based algorithm for symbolic trajectory evaluation," in *Proc. 13th Adv. Res. Work. Conf. Correct Hardware Des. Verification Methods*, 2005, pp. 238–253.

[42] O. Grumberg, A. Schuster, and A. Yadgar, "Memory efficient all-solutions SAT solver and its application for reachability analysis," in *Proc. 5th Int. Conf. Formal Methods Comput.-Aided Des.*, 2004, pp. 275–289.

[43] Y. Yu, P. Subramanyan, N. Tsiskaridze, and S. Malik, "All-SAT using minimal blocking clauses," in *Proc. 27th Int. Conf. VLSI Des. 13th Int. Conf. Embedded Syst.*, 2014, pp. 86–91.

[44] A. Morgado and J. Marques-Silva, "Good learning and implicit model enumeration," in *Proc. 17th IEEE Int. Conf. Tools Artif. Intell.*, 2005, pp. 6–11.

[45] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set enumeration," in *Proc. Int. Conf. Log. Program. Nonmonotonic Reason.*, 2007, pp. 136–148.

[46] T. Toda and K. Tsuda, "BDD construction for all solutions SAT and efficient caching mechanism," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, 2015, pp. 1880–1886.

[47] T. Toda and T. Soh, "Implementing efficient all solutions SAT solvers," *J. Exp. Algorithmics*, vol. 21, pp. 1–12, 2016.

[48] J. Huang and A. Darwiche, "Using DPLL for efficient OBDD construction," in *Proc. Int. Conf. Theory Appl. Satisfiability Testing*, 2004, pp. 157–172.

[49] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Proc. 19th Int. Conf. Theory Appl. Satisfiability Testing*, 2016, pp. 123–140.