WAP: SAT-based Computation of Minimal Cut Sets

Weilin Luo, Ou Wei[†]
Department of Computer Science, Nanjing University of Aeronautics and Astronautics, China.
{luoweilin,owei}@nuaa.edu.cn

***Abstract*—Fault tree analysis (FTA) is a prominent reliability analysis method widely used in safety-critical industries. Computing minimal cut sets (MCSs), i.e., finding all the smallest combination of basic events that result in the top level event, plays a fundamental role in FTA. Classical methods have been proposed based on manipulation of boolean expressions of fault trees and Binary Decision Diagrams. However, given the inherent intractability of computing MCSs, developing new methods over different paradigms remains to be an interesting research direction. In this paper, motivated by recent progress on modern SAT solver, we present a new method for computing MCSs based on SAT solving. Specifically, given a fault tree, we iteratively search for a cut set based on the DPLL framework. By exploiting local failure propagation paths in the fault tree, we provide efficient algorithms for extracting an MCS from the cut set. The information of a new MCS is learned as a blocking clause for SAT solving, which helps to prune search space and ensures completeness of the results. We compare our method with a popular commercial FTA tool on practical fault trees. Preliminary results show that our method exhibits better performance on time and memory usage.**

***Index Terms*—Fault Tree Analysis, Minimal Cut Sets, SAT Solving, DPLL**

I. INTRODUCTION

Fault tree analysis (FTA) [16] is a prominent reliability analysis method widely used in safety-critical industries, such as aerospace and power plants. It computes a wide class of system reliability properties and measures over a fault tree that models failure propagation through a system. A minimal cut set (MCS) of a fault tree is the smallest combination of component failures, called basic events, that leads to a system failure represented by the top event. Finding all the MCSs 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.

Computing MCSs of a fault tree is a problem that is exponential with respect to the number of basic events in the tree. Classical methods for determining MCSs are based on manipulation of boolean expressions [6] and Binary Decision Diagrams (BDDs) [4], [13]. The boolean manipulation methods use boolean algebra laws to transform the boolean formula defined by a fault tree into disjunction normal form such that each conjunction is an MCS, 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 given, the size of the BDD can be greatly reduced. Efficient minimization algorithms then can be applied to the BDD to compute MCSs. In practice, commercial FTA tools based on these methods have been widely used in industries. Given the inherent intractability of computing MCSs, however, developing new methods over different paradigms remains to be an interesting research direction.

Motivated by recent progress on modern solver on Boolean Satisfiability Problem (SAT) [11], we present a new method for computing MCSs based on SAT solving. SAT determines whether there exists a model for a given boolean formula. Although the problem is proved to be NP-Complete, 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 hardware and software verification, model checking, and circuit testing. Surprisingly, there is lack of work on computing MCSs of practical fault trees based on SAT solving. In this paper, we extend the Davis-Putnam-Loveland-Logemann (DPLL) framework, used by most successful complete SAT solvers, with the ability of computing MCSs. DPLL is a depth-first backtracking framework, which progresses iteratively by making a decision about a variable of a boolean formula and its value, propagates implications of this decision, and backtracks in the case of a conflict. We therefore exploit the power of the DPLL framework to search for possible combinations of basic events, while employ efficient algorithms for computing MCSs that are optimized for fault tree structures.

We present in this paper the following main features of our method. (a) *Tseitin Encoding of Fault trees.* DPLL requires a propositional formula presented in CNF for efficient search. The structure function of a fault tree, however, typically is a non-clausal formula. We then encode a fault tree in CNF using Tseitin encoding, which introduces new variables correspond naturally the gates in the fault tree. This encoding enables linear increase in the size of the formula. More importantly, from a cut set with the auxillary variables, we are able to derive the structure information of the fault tree that can be used for efficient extraction of an MCS. (b) *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. In our work, we provide a novel MCS extraction algorithm by exploiting the structures of a fault tree. We define an LPG induced by a cut set, characterizing the failure propagation between the events in the cut set. Testing necessity of basic events is then performed in an incremental way over the LPG that keeps

[†]Corresponding author

updated with the progress of testing. (c) *Clause Learning with Dynamic Deletion.* To find all the MCSs, information of previous computed MCSs are learned as blocking clauses that prevent the same MCSs being computed again. Conflict clauses found in DPLL are also used to prune the search space. Meanwhile, we employ a clause deletion strategy that periodically shrinks the size of the clauses, avoiding reduced DPLL performance and high memory usage caused by a large number of clauses.

We have implemented our method and compared it with FaultTree+, one of the most popular commercial FTA tools. We conduct experiments on practical fault trees from a benchmark repository [1]. Experiments show that our method exhibits better performance on both time and memory usage.

The paper is organized as follows. Section II introduces the necessary notations and background. Section III presents the main features of our method in detail. Section IV reports the experiments for performance evaluation of the method. Section V discusses related work and concludes the paper.

II. Preliminaries

This section reviews the notations of Boolean formulas and fault trees, and the DPLL framework.

Notations. A *literal* is either a boolean variable v, called a positive literal, or its negation $\neg v$, called negative. v and $\neg v$ are said to be opposite. A *product* is a set of literals that does not contain both a literal and its opposite. An *assignment* over a set of boolean variable V is denoted by a product π such that for any $v \in V$, v is assigned to true if and only if $v \in \pi$, and assigned to false if and only if $\neg v \in \pi$. If all the variables in V are assigned, i.e., $|\pi| = |V|$, π is called a *full* assignment over V, and *partial* otherwise. Given a boolean formula f defined over V, π is a *model* of f, denoted by $\pi \models f$, if and only if f evaluates to true under π. A *clause* is a disjunction of literals. A boolean formula is in conjunction normal form (CNF) if it is a conjunction of a set of clauses.

Fault Tree. A fault tree F is a directed acyclic graph consisting of two types of nodes: *events* and *gates*. An event is a boolean variable representing the failure status of a subsystem or an individual component, also called a basic event. Events are divided into basic events E_F and internal events G_F. The event $r_F \in G_F$ at the top of the tree is called the top level event. Events are connected through gates; each gate is associated with an internal event as its output. For the purpose of this paper, we consider coherent fault trees with AND ($\wedge$) and OR ($\vee$) gates. Following the logical structures in F, the semantics of F, i.e., the structure function, is a boolean formula f_F built over E_F, called the structure function. Let π be a product containing only positive literals corresponding to the basic events E_F. π is called a *cut set* of F if π is a model of f_F; π is said to be *minimal* if and only if there does not exist another cut set π' such that $\pi' \subset \pi$.

We use a set of boolean equations $\mathcal{Q}$ to describe the gates in F. 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 $\hat{g}$,

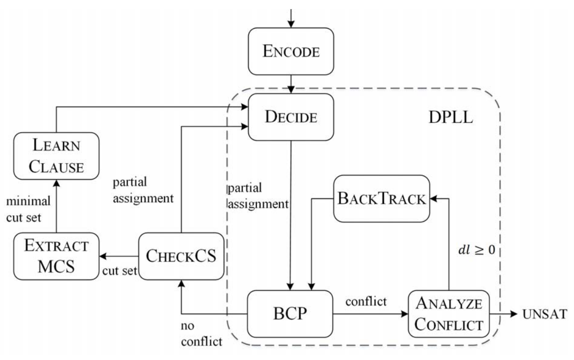

Fig. 1. The framework of SATMCS.

$In \subseteq E_F \cup G_F$ is the set of input events of $\hat{g}$, and $Out = g$ is the output event.

DPLL. Main components of the DPLL framework are shown in the dashed box of Fig. 1. DPLL is a depth-first backtracking framework. The input to DPLL is a CNF formula f. During each iteration, Decide uses some heuristics to choose a free variable v of f for assignment and branching search. The depth of the search is indicated by a *decision level (dl)*. *dl* starts at 1 and is increased by 1 for subsequent decisions until a backtrack occurs. BCP repeats applying the *unit clause rule* over the current partial assignment until either there are no more implications or a conflict is encountered. In the former case, DPLL calls Decide for next decision. Otherwise, DPLL will have to backtrack. AnalyzeConflict then finds a subset of the current variable assignments as a reason for the discovered conflict and returns the *dl* to backtrack to. The information of the subset of variable assignments is recorded as a conflict clause that prevents the same conflict from occurring again. BackTrack enters decision level -1 only when some variable is implied by the formula to be both true and false i.e. the instance is unsatisfiable.

III. SAT-based Computation of Minimal Cut Sets

In this section, we present our method for SAT-based computing MCSs, referred to as SATMCS. We give an overview of the method, and then introduce the main components.

A. Overview

The framework of SATMCS is shown in Fig. 1. SATMCS takes as an input a fault tree F represented as a set of clauses $\mathcal{F}$ using *Tseitin encoding*. Based on the DPLL procedure, SATMCS iteratively searches for cut sets of $\mathcal{F}$ and extract MCSs from them. During the iteration, SATMCS learns previous MCSs and conflict clauses to help to prune the search space and speed up the search process. To this end, SATMCS maintains a CNF formula $\mathcal{H} = \mathcal{F} \cup \mathcal{B} \cup \mathcal{C}$ that consists of three sets of clauses: $\mathcal{F}$ encodes F, $\mathcal{B}$ blocks already computed MCSs, and $\mathcal{C}$ contains the conflict clauses from DPLL.

Note that cut sets typically are partial assignments. Therefore, when there is no conflict detected for a partial assignment π in BCP, SATMCS calls CheckCS to determine whether π is model of $\mathcal{F}$; if true, according to the structures of $\mathcal{H}$, π is guaranteed to be a cut set of F not covered by any

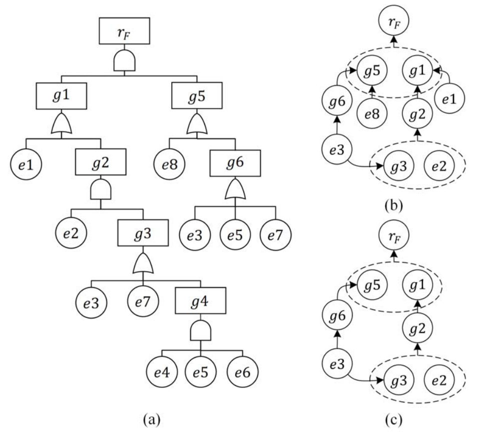

Fig. 2. (a) A fault tree; (b) An LPG; (c) A reduced LPG.

already computed MCSs. EXTRACTMCS, a procedure for testing necessity of the literals in a cut set, then is called to extract an MCS from π. The information of a new MCS is learned by LEARNCLAUSE by adding a blocking clause into $\mathcal{B}$, preventing the same MCS being computed again. At the same time, $\mathcal{C}$ is updated to include new conflict clauses from DPLL. The updated $\mathcal{H}$ is then sent to DECIDE for next iteration. SATMCS terminates when $\mathcal{H}$ is unsatisfiable, meaning that all the MCSs have been found. In the following, we introduce the components above in detail.

B. Tseitin Encoding of Fault Trees

A propositional formula in CNF representation allows for efficient iteration application of the unit clause rule in the DPLL framework, which is the workhorse of almost all modern SAT solvers. To take advantage of this feature, SATMCS takes as input a fault tree F encoded in CNF.

From the structure function of F, it is well known that the formula can be transformed into an equivalent CNF formula, while potentially increasing the size of the formula exponentially. On the other hand, at the price of introducing new variables for subformulas, Tseitin encoding [15] enables a transformation with only a linear increase in the size of the formula. In our work, given the set of boolean equations $\mathcal{Q}$ of F, we directly encode F as the CNF formula $\mathcal{F}$ using Tseitin encoding. According to the structures of fault trees, the auxillary variables correspond naturally to the internal events G_F. Specifically, for each boolean equation $Q_{\hat{g}} = \langle Op, \{x_1, \dots, x_n\}, g\rangle$, $Q_{\hat{g}}$ is transformed to a set of clauses $U_{\hat{g}} = L_{\hat{g}} \cup R_{\hat{g}}$, where $L_{\hat{g}}$ denotes the clauses transformed from the formula $g \Rightarrow Op(\{x_1, \dots, x_n\})$, and $R_{\hat{g}}$ – from the formula $g \Leftarrow Op(\{x_1, \dots, x_n\})$. If Op is OR, $L_{\hat{g}} = \{\neg g \vee \bigvee_{i=1}^{n} x_i\}$ and $R_{\hat{g}} = \bigcup_{i=1}^{n}\{\neg x_i \vee g\}$; if Op is AND, $L_{\hat{g}} = \bigcup_{i=1}^{n}\{\neg g \vee x_i\}$ and $R_{\hat{g}} = \{g \vee \bigvee_{i=1}^{n} \neg x_i\}$. $\mathcal{F}$ is then defined by $\bigcup_{g \in G_F} U_{\hat{g}} \cup \{r_F\}$. Note that, by assuming F be a coherent fault tree with only AND and OR gates, the above encoding can be optimized by considering only $L_{\hat{g}}$ for each $Q_{\hat{g}}$. We include $R_{\hat{g}}$ in our encoding because it helps to force assignments to auxillary variables in BCP, which simplifies the extraction of MCSs later.

C. Extracting MCSs using Local Fault Graph

As mentioned before, if no conflict is detected in BCP, SATMCS calls CHECKCS to determine whether the current partial assignment π is model of the formula $\mathcal{F}$; if true, π is guaranteed to be a cut set of F not covered by any already computed MCSs. In our implementation, we keep a list for the clauses in $\mathcal{F}$, and CHECKCS watches the first unresolved clause in the list for each decision level in DPLL. Then, given the decision level dl corresponding to π, instead of always checking the list from the beginning, CHECKCS checks whether the new assignments in dl can satisfy more clauses than that in the previous decision level, i.e., it starts from checking the unresolved clause watched for $dl - 1$. Thanks to the monotonic satisfiability of CNF, this watching mechanism improves the efficiency of CHECKCS by avoiding unnecessary clause checking.

After π is identified as a cut set of F, EXTRACTMCS is called to extract an MCS from π. Let $\pi^+_{E_F}$ be the set of positive literals in π corresponding to the basic events E_F of F. By assuming F be a coherent fault tree, the MCS must be a subset of $\pi^+_{E_F}$. EXTRACTMCS conducts a sequence of queries on F; each query tests the necessity for a literal in $\pi^+_{E_F}$. These tests can be implemented by direct calls to a SAT solver. To improve efficiency, we exploit the structures of F, providing an incremental extraction algorithm based on a *Local Propagation Graph* (LPG) induced by a cut set. Experiments (Section IV-B) show that our algorithm is more efficient than direct calls to a SAT solver, and the linear strategy is also better than the asymptotically optimal QuickXplain algorithm [9], [2] for practical fault trees.

An LPG can be considered as a projection of F over the positive literals in π, characterizing the failure propagation between the corresponding events. Formally, let π^+ be the set of positive literals in π. 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 hyper-edges. An edge $(S, v) \in R$ denotes that occurrence of all the events in S causes the event v to occur. Note that the top event r_F must be in V, since π is a cut set of F. P_{π^+} is constructed by application of the following two rules for each $v \in \pi^+$: (i) if there exists a boolean equation $\langle \mathsf{AND}, In, v\rangle \in \mathcal{Q}$, add (In, v) into R; (ii) if there exists a $v' \in \pi^+$ and a boolean equation $\langle \mathsf{OR}, In, v'\rangle \in \mathcal{Q}$ such that $v \in In$, then $(\{v\}, v')$ into R. Note that rule (i) is well-defined: according to our encoding of F in Section III-B, if $v \in \pi^+$, then $In \subseteq \pi^+$; therefore, $In \subseteq V$. Intuitively, the above two rules, respectively, captures the *definite* failure propagation to v and from v restricted by the events in π^+. For example, for the fault tree shown in Fig. 2(a), consider $\tilde{\pi}^+ = \{r_F, g1, g5, g2, g3, g6, e1, e2, e3, e8\}$ derived from a cut set $\tilde{\pi}$. The LPG over $\tilde{\pi}^+$ is shown in Fig. 2(b). For the basic event $e1$, according to $\langle \mathsf{OR}, \{g2, e1\}, g1\rangle$, if $e1$ occurs, $g1$ must occur; therefore, $P_{\tilde{\pi}^+}$ contains the edge $(\{e1\}, g1)$. For the internal event $g2$, $\langle \mathsf{AND}, \{g3, e2\}, g2\rangle$ implies that $g1$ occurs if and only if both $g3$ and $e2$ occur; hence, $P_{\tilde{\pi}^+}$ contains the hyper-edge $(\{g3, e2\}, g2)$.

Based on an LPG P_{π^+}, EXTRACTMCS tests the necessity of each basic event $v \in \pi^+_{E_F}$ by analyzing the failure propagation paths in P_{π^+} triggered by v. If removing those paths from P_{π^+} does not cause the top event r_F to be removed, that means v is unnecessary for causing the r_F to occur, and P_{π^+} is reduced for next test; otherwise, v is included into an MCS. Because P_{π^+} only preserves the failure propagation information relevant to π^+ and P_{π^+} continues to be reduced with the detection of unnecessary events, the efficiency of the sequence of queries in EXTRACTMCS is improved.

ALGORITHM 1: EXTRACTMCS

```
INPUT   : $\pi, \mathcal{Q}$
OUTPUT : an $MCS$ extracted from $\pi$
1  $mcs \leftarrow \emptyset$
2  $(V, R) \leftarrow$ BUILDLPG($\pi^+, \mathcal{Q}$)
3  for each $v \in \pi^+_{E_F}$ do
4      if ISNECESSARY($R, v$) then
5          $mcs \leftarrow mcs \cup \{v\}$
6      end
7  end
8  return $mcs$
```

ALGORITHM 2: ISNECESSARY

```
INPUT   : $R, v$
OUTPUT : the necessity status of $v$
1  $R' \leftarrow R$
2  $eventSet \leftarrow \{v\}$
3  while $eventSet \neq \emptyset$ do
4      $l \leftarrow eventSet.pop()$
5      for each $(S, l') \in R'$ do
6          if $l \in S$ then
7              if $(|S| \geq 2)$ or $\neg(\exists \tilde{l} \neq l \cdot (\{\tilde{l}\}, l') \in R')$ then
8                  if $l'$ is $r_F$ then
9                      return TRUE
10                 end
11                 $eventSet \leftarrow eventSet \cup \{l'\}$
12             end
13             $R' \leftarrow R' \setminus \{(S, l')\}$
14         end
15     end
16 end
17 $R \leftarrow R'$
18 return FALSE
```

ALGORITHM 1 shows the framework of EXTRACTMCS. The inputs include the set of boolean equations $\mathcal{Q}$ of a fault tree F and a cut set π identified by CHECKCS. BUILDLPG constructs the LPG $P_{\pi^+} = \langle V, R \rangle$ based on the rules above. ISNECESSARY (ALGORITHM 2) checks the necessity of a basic event v in $\pi^+_{E_F}$ over R. $eventSet$ is initialized with $\{v\}$, used to record the events influenced along the local propagation paths triggered by v; that is, if v does not occur, the events in $eventSet$ will not occur. ISNECESSARY creates a copy of R for backtracking and computes $eventSet$ through chaotic iteration. For each edge (S, l'), if an event $l \in S$, we conclude that l influences l' in two cases (line 7). Case (1) $|S| \geq 2$: by the construction of P_{π^+}, l' is associated with an AND gate; hence, l' is influenced by l. Case (2) $\neg(\exists \tilde{l} \neq l \cdot (\{\tilde{l}\}, l') \in R')$: note that $|S| = 1$ implies that l' corresponds to an OR gate; therefore, if l is the event in π^+ that causes l' to occur, l' is influenced by l. If we find l' is the top event r_F (line 8), that means r_F is influenced by v; thus, v is necessary, and ISNECESSARY returns $TRUE$. Otherwise, l' is added into $eventSet$ for iteration (line 11) and the edge (S, l') is removed. If r_F is detected during the iteration, which means v is not necessary for the occurrence of r_F, then R is reduced for next test, and ISNECESSARY returns $FALSE$.

For example, consider the previous LPG over $\tilde{\pi}^+$ again (Fig. 2(b)). 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. $e1$ does not influence $g1$, because of the existence of the edge $(\{g2\}, g1)$; therefore, the edge $(\{e1\}, g1)$ can be removed from the graph; in the reduced graph, $e2$ becomes an necessary event, because non-occurrence of $e2$ will imply non-occurrence of r_F through the propagation path $e2 \rightarrow g2 \rightarrow g1 \rightarrow r_F$. Subsequent tests show that $e3$ is necessary because of the path $e3 \rightarrow g3 \rightarrow g2 \rightarrow g1 \rightarrow r_F$, while $e8$ is unnecessary. The final reduced LPG is shown in Fig. 2(c), and $\{e2, e3\}$ is the discovered MCS.

D. Clause Learning with Dynamic Deletion

To find all the MCSs and improve efficiency, the CNF formula $\mathcal{H} = \mathcal{F} \cup \mathcal{B} \cup \mathcal{C}$ keeps updated in LEARNCLAUSE. After a new MCS is extracted, a blocking clause consisting of the opposite to the literals in the MCS is added to $\mathcal{B}$; this ensures that the same MCS will not be discovered again. Moreover, similar to DPLL, LEARNCLAUSE also updates $\mathcal{C}$ to include the conflict clauses generated from ANALYZECONFLICT, which helps BCP to prune the search space.

As the computation progresses, however, a large number of clauses in $\mathcal{H}$ will reduce the performance of the DPLL procedure. To address this issue, LEARNCLAUSE uses a clause delete strategy to dynamically shrink the size of $\mathcal{H}$. 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 DPLL and multiple conflict clauses can be discovered by ANALYZECONFLICT. 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 $\mathcal{C}$ and the DPLL procedure is restarted.

IV. EXPERIMENTS

We have implemented SATMCS over ZChaff [5], one of the state-of-art modern SAT solvers based on the DPLL framework. We assess the performance of SATMCS in this section through two classes of experiments. The first one compares SATCMS and FaultTree+ on the efficiency of computing MCSs. The second one compares our algorithm for extracting

TABLE I
EXPERIMENTAL RESULTS ON COMPUTING MCSS

Benchmark	#E	#G	#MCS	SATMCS		FaultTree+	
				Time (sec.)	Mem (MB)	Time (sec.)	Mem (MB)
baobab3	80	107	24386	12.7	66.4	–	–
chinese	25	36	392	<0.1	3.1	0.8	89.5
elf9601	145	242	151348	38.4	105.3	–	–
ftr10	175	94	396	<0.1	3.2	1.1	91.4
jbd9601	533	315	14007	1.6	8.5	2.8	109.2
das9205	51	20	17280	<0.1	3.4	0.8	86.1
das9203	51	30	16200	<0.1	3.1	0.7	86.4
das9204	53	30	16704	<0.1	3.3	0.6	89.7
das9202	49	36	27778	<0.1	3.3	1.6	88.9
das9209	109	73	8.2E10	<0.1	3.1	0.7	96.1
das9206	121	112	19518	0.4	5.2	3.6	92.9
das9201	122	82	14217	0.2	4.1	1.3	91.8
das9208	103	145	8060	1.9	9.8	1.5	96.2
das9207	276	324	25988	4.7	10.2	3.5	100.2
edf9205	165	142	21308	0.5	5.2	376.4	248.6
edf9201	183	132	579720	56.1	57.7	2423	723.8
edf9203	362	475	–	–	–	–	–
edf9204	323	375	–	–	–	–	–
edf9202	458	435	130112	5.5	9.7	–	–
edf9206	240	362	–	–	–	–	–
edfpa15r	88	110	26549	26.8	142.1	85.6	124.1
edfpa14r	106	132	–	–	–	–	–
edfpa15p	276	324	27870	22.5	142.3	409.8	179
edfpa14p	124	101	–	–	–	–	–
edfpa14q	311	194	–	–	–	–	–
edfpa15b	283	249	2910473	2662.6	1493.8	–	–
edfpa14o	311	173	–	–	–	–	–
edfpa14b	311	290	–	–	–	–	–
isp9606	89	41	1776	<0.1	3.1	0.9	90.2
isp9607	74	65	150436	<0.1	4.2	5.1	94.2
isp9603	91	95	3434	0.1	3.6	1.6	93.4
isp9602	116	122	5197647	1.6	12.6	16.6	94.1
isp9604	215	132	746574	<0.1	3.2	1.1	88

TABLE II
EXPERIMENTAL RESULTS ON EXTRACTION OF MCSS

Benchmark	SATMCS		SATMCS-Call		SATMCS-QX	
	Time (sec.)	Mem (MB)	Time (sec.)	Mem (MB)	Time (sec.)	Mem (MB)
baobab3	12.7	66.4	36.1	65.6	16.6	93.4
elf9601	38.4	105.3	118.1	104.1	51.8	136.7
edf9201	56.1	57.7	96.6	54.2	53.9	56.2
edf9202	5.5	9.7	43.9	9.5	13.3	21.1
edfpa15r	26.8	142.1	127.8	96.6	88.1	147.3
edfpa15p	22.5	142.3	114.9	150.4	43.1	238.2
edfpa15b	2662.6	1493.8	6659.5	644.8	5415.6	1005.5

MCSs with other alternatives. The experiments are performed on an Intel Xeon E5-1603 2.8GHz, with 16GB of memory.

A. Comparison between SATMCS and FaultTree+

FaultTree+ is one of the most popular FTA commercial tools that has been used worldwide on safety-critical industries for analyzing large-scale fault trees. According to the documentation,FaultTree+ conducts module identification on an input fault tree and then uses a bottom-up boolean manipulation method to calculate MCSs. We compare SATMCS with FaultTree+ on 33 coherent fault tree benchmarks [1] from real life applications such as aerospace and nuclear power industries. We use the technique in [10] to identify modules in SATMCS and set the clause deletion threshold be a constant 0.7. The time limit is set to one hour and the memory limit to 4GB.

Table I shows the experiment results. The first four columns present the information of the fault trees, including the name, the number of basic events (#E) and gates (#G), and the number of MCSs (#MCS). The benchmarks are organize into different groups based on the prefix of their names that indicate the sources, e.g., *das** come from aerospace industry, and *edf** come from nuclear power industry. The last four columns show the CPU and memory used by SATMCS and FaultTree+, where dashes indicate that computation did not complete due to either memory or time limits.

Overall, SATMCS can solve 25 cases of the 33 fault trees, which include all the 21 ones that FaultTree+ can successfully compute. Note that for the 21 cases solved by both of them, SATMCS and FaultTree+ produce exactly the same set of MCSs. It is observed that SATMCS generally computes much faster than FaultTree+ for the 21 cases; moreover, the memory used by SATMCS is significantly less than that used by FaultTree+ in most cases – SATMCS consumes about one order of magnitude less memory than FaultTree+ in 19 cases, except for *edfpa15r* and *edfpa15p* where the memory used by them are comparable.

All the 8 fault trees that SATMCS can not solve belong to the *edf** group. In our experiments, we have noticed that analyzing this group of fault trees generates a huge number of conflict clauses added to $\mathcal{H}$. This increase time and memory usage for BCP procedure in DPLL. We attempt to address this problem in future by considering more refined clause deletion strategy based on statists of usage and length information of conflict clauses [5].

B. Evaluation of Extracting MCSs

EXTRACTMCS tests the necessity of a basic event by analyzing failure propagation paths in an LPG induced by the cut set. Such a test can be done by a direct call to a SAT solver. To evaluate the different approaches, we implemented a variant of SATMCS, called SATMCS-Call, which replaces the LPG-based test in EXTRACTMCS with direct calling ZChaff. We compare SATMCS and SATMCS-Call on the same benchmarks as above. Table II shows a partial list of the results. SATMCS takes about less than half of the time used by SATMCS-Call, which shows the advantage of the LPG-based approach. On the other hand, SATMCS-Call consumes slightly less memory than SATMCS in most cases. Particularly, for *edfpa15b*, the memory used by SATMCS-Call is less than the half of that by SATMCS, which may be due to the efficient manipulation of a large number of clauses in ZChaff. This indicates that there is room for improving SATMCS by adopting optimized mechanisms for clause management.

Another feature of EXTRACTMCS is that it conducts a linear number of tests for deciding the necessity of basic events in a cut set. An alternative to this is to use the QuickXplain algorithm [9], [2], 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. In our case,SATMCS uses DPLL to search for cut sets. It turns out that the cut sets found by DPLL are very close to the MCSs extracted from them – according to our statistics from the benchmarks, the average ratio between the number of basic events in an MCS and that in the corresponding cut set is 0.72. In this case, we expect better performance of extracting MCSs using the linear approach implemented in EXTRACTMCS, than using QuickXplain. We implemented another variant of SATMCS using QuickXplain, called SATMCS-QX, and compared with SATMCS. Table II shows that, with comparable memory usage, SATMCS takes about half of the time used by SATMCS-QX in most cases, which confirms our analysis above.

V. Related Work and Conclusion

Classical methods for computing MCSs of fault trees are based on boolean manipulation [6] and BDDs [4], [13], which have been well studied and used for developing industrial tools. A number of improvements, e.g., based on zero-suppressed BDD [14] and symmetric structures [3], have been proposed. Given the complexity of the problem, however, these methods may still suffer from time or memory constraints in practice. It is necessary to develop new techniques from different perspectives to complement existing ones. We presented and implemented SATMCS, a novel method for computing MCSs based on SAT solving. We compared SATMCS with a popular FTA tool in market. Thanks to the efficiency of the DPLL procedure and the dedicated encoding and analysis algorithms that we have developed for fault trees, preliminary experiment results show the method based on SAT solving is promising. To the best of our knowledge, this is the first report that is targeted for computing MCSs in fault tree analysis based on SAT solving.

Our work is related to the All-SAT problem [7], [17] that computes all the satisfying assignments of a propositional logic 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 and BDDs. Although an All-SAT solver finds all the assignments that satisfy constraints, it is distinct from and computationally less expensive than the problem of computing MCSs, since the satisfying assignments are not necessary to be minimized.

Computing MCSs of fault trees can be considered as a special case of enumerating prime implicants of boolean formulas [8]. Along this line of research, the work by Previti et al. [12] is the closet to ours, where they proposed algorithms for finding prime implicants of non-clausal formulas based on iterative SAT solving. Their approach uses dual rail encoding 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 DPLL with the algorithms for computing MCSs; hence, we are able to work on partial assignments within the DPLL procedure and discover MCSs earlier. Moreover, the approach in [12] relies on the QuickXplain algorithm for extracting prime implicants from satisfying assignments. This algorithm, as discussed in Section IV-B, may require more number of satisfiability queries than our approach.

Our experiments suggest that the bottleneck of applying SATMCS to complex fault trees is the large number of clauses learned with the progress of computation. While the clauses can help to prune some of the search space, an excessive growth of them can slow down the BCP procedure and use up memory. Better trade-off may be achieved by considering the statistics of usage information of the clauses. we leave further investigations along this research direction to future work.

Acknowledgment

We thank Jian Zhang for discussion on the paper; Junhua Song for her help with the experiments; and anonymous referees for helpful comments. This work has been financially supported by the National Natural Science Foundation of China under Grant No.61170043, and the National Basic Research Program of China under Grant No.2014CB744903.

References

[1] A repository of fault tree benchmarks. http://web.archive.org/web/20061204235930/http://iml.univ-mrs.fr/~arauzy/aralia/benchmark.html.

[2] A. R. Bradley and Z. Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In *FMCAD'07*, pp. 173–180.

[3] D. Codetta-Raiteri. "BDD Based Analysis of Parametric Fault Trees". In *Reliability and Maintainability Symposium*, pp. 442–449. IEEE, 2006.

[4] O. Coudert and J. C. Madre. "Fault Tree Analysis: 10^{20} Prime Implicants and Beyond". In *Reliability and Maintainability Symposium*, pp. 240–245, 1993.

[5] Y. S. Mahajanand Z. Fu and S. Malik. "Zchaff2004: An Efficient SAT Solver". In *SAT'04*, pp. 360–375, 2004.

[6] J. B. Fussell and W. E. Vesely. "New Methodology for Obtaining Cut Sets for Fault Trees". Tech. Rep., Georgia Inst. of Tech., 1972.

[7] O. Grumberg, A. Schuster, and A. Yadgar. "Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis". In *FMCAD'04*, pp. 275–289, 2004.

[8] S. Jabbour, J. Marques Silva, L. Sais, and Y. Salhi. "Enumerating Prime Implicants of Propositional Formulae in Conjunctive Normal Form". In *JELIA'14*, pp. 152–164, 2014.

[9] U. Junker. "Preferred Explanations and Relaxations for Over-Constrained Problems". In *AAAI'04*, pp. 167–172, 2004.

[10] T. Kohda, E. J. Henley, and K. Inoue. "Finding Modules in Fault Trees". *IEEE Transactions on Reliability*, 38(2):165–176, 1989.

[11] S. Malik and L. Zhang. "Boolean Satisfiability from Theoretical Hardness to Practical Success". *Comm. ACM*, 52(8):76–82, 2009.

[12] A. Previti, A. Ignatiev, A. Morgado, and J. Marques Silva. "Prime Compilation of Non-Clausal Formulae". In *IJCAI'15*, pp. 1980–1987.

[13] A. Rauzy and Y. Dutuit. "Exact and Truncated Computations of Prime Implicants of Coherent and Non-Coherent Fault Trees within Aralia". *Reliability Engineering and System Safety*, 58(2):127–144, 1997.

[14] Z. Tang and J. B. Dugan. "Minimal Cut Set/Sequence Generation for Dynamic Fault Trees". In *Reliability and Maintainability Symposium*, pp. 207–213, 2004.

[15] G. Tseitin. "On the Complexity of Derivations in the Propositional Calculus". *Studies in Constrained Mathematics and Mathematical Logic*, pp. 234–259, 1968.

[16] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. "Fault Tree Handbook". Tech. Rep., U.S. Nuclear Regulatory Comm., Jan 1981.

[17] Y. Yu, P. Subramanyan, N. Tsiskaridze, and S. Malik. "All-SAT Using Minimal Blocking Clauses". In *International Conference on VLSI Design*, pp. 86–91, 2014.