An Efficient Two-phase Method for Prime Compilation of Non-clausal Boolean Formulae

1st Weilin Luo
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
luowlin3@mail2.sysu.edu.cn

2nd Hai Wan†
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
wanhai@mail.sysu.edu.cn

3rd Hongzhen Zhong
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
zhonghzh5@mail2.sysu.edu.cn

4th Ou Wei
Department of Computer Science
University of Toronto
Toronto, Canada
owei@cs.toronto.edu

5th Biqing Fang
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
fangbq3@mail3.sysu.edu.cn

6th Xiaotong Song
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
songxt5@mail2.sysu.edu.cn

2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 978-1-6654-4507-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCAD51958.2021.9643520

***Abstract*—Prime compilation aims to generate all prime implicates/implicants of a Boolean formula. Recently, prime compilation of non-clausal formulae has received great attention. Since it is hard for Σ_2^P, existing methods have performance issues. We argue that the main performance bottleneck stems from enlarging the search space using dual rail (DR) encoding, and computing a minimal clausal formula as a by-product. To deal with the issue, we propose a two-phase approach, namely `CoAPI`, for prime compilation of non-clausal formulae. Thanks to the two-phase framework, we construct a clausal formula without using DR encoding. In addition, to improve performance, the key in our work is a novel bounded prime extraction (BPE) method that, interleaving extracting prime implicates with extracting small implicates, enables constructing a succinct clausal formula rather than a minimal one. Following the assessment way of the state-of-the-art (SOTA) work, we show that `CoAPI` achieves SOTA performance. Particularly, for generating all prime implicates, `CoAPI` is up to about one order of magnitude faster. Moreover, we evaluate `CoAPI` on a benchmark sourcing from real-world industries. The results also confirm the outperformance of `CoAPI`[1].**

I. Introduction

Prime compilation aims to generate all the prime implicates/implicants of a Boolean formula. A prime implicate/implicant does not contain redundant literals so it can provide the property of minimality in some applications such as knowledge compilation [1], logic minimization [2, 3] and digital circuit analysis and optimization [4]. Besides, this problem has widely applications, *e.g.*, logic synthesis [5, 6], model checking [7], and fault tree analysis [8].

In practice, most problem representations are not in normal form [9]. A simple way to handle non-clausal[2] formulae is to exploit Shannon's expansion [10], but this approach results in an exponential size in the worst case. Hence, non-clausal formulae are often transformed into conjunctive normal form (CNF) by some encoding methods, *e.g.* Tseitin encoding [11], which reduces the complexity of the transformation by adding auxiliary variables. Most of the early works only generate all prime implicates/implicants of the transformed CNF. Note that the prime implicates/implicants of the transformed CNF are not that of the non-clausal formula because the transformed CNF and the non-clausal formula are not logical equivalent. Therefore, the early works cannot compute all prime implicates/implicants of a non-clausal formula. This limits their applications in practice. Moreover, prime compilation of non-clausal formula is hard for Σ_p^2 [12]. Developing new approaches over different paradigms remains to be an interesting research direction. Therefore, prime compilation of non-clausal formula has received great attention [13, 14].

Previti et al. (2015) proposed the state-of-the-art (SOTA) approaches for compiling non-clausal formulae [14]. They construct a minimal cover (Definition 1) represented as a set of prime implicates and find all prime implicants simultaneously. However, there are some issues to limit their performance. (1) They use dual rail (DR) encoding [15] throughout the algorithm. DR encoding needs a double number of variables as the original encoding resulting in a larger search space. (2) It it time-consuming to construct a minimal cover since

†Corresponding author.

This paper was supported by the Guangdong Province Science and Technology Plan projects (No. 2017B010110011), National Natural Science Foundation of China (No. 61976232), and Guangdong Province Natural Science Foundation (No. 2018A030313086).

[1]Our code and benchmarks are publicly available at https://github.com/LuoWeiLinWillam/CoAPI

[2]Throughout the paper, the term non-clausal is used to denote Boolean formulae not necessarily represented as CNF or DNF.

they requires considerable SAT queries to guarantee that each implicates is prime. (3) To ensure correction, they require minimal or maximal models in SAT solving, which limits the polarity heuristic of SAT solver.

We argue that DR encoding is the key to compute all prime implicates/implicants, but not necessary for constructing a cover. Besides, because the by-produce cover aims to generate all prime implicates/implicants, it is necessary to ensure the logical equivalent of the cover and the input Boolean formula, while not necessary to compute the minimal cover. Moreover, it is questionable whether finding a minimal cover has a practical value since the influence of the size of the cover on the other parts of the approach is only vaguely known.

In this paper, we propose a two-phase approach, namely CoAPI, to compile a non-clausal formula. In the first phase, CoAPI constructs a succinct cover in the form of CNF and it is logically equivalent to the input Boolean formula. In the second phase, based on the cover, CoAPI generates all prime implicates/implicants. Thanks to the two independent phases, in the first phase, we avoid enlarging the search space due to DR encoding and exploit the powerful polarity heuristic.

Computing a cover efficiently and constructing a succinct cover is trade-off. If we one-sidedly seek to efficiently compute implicates to construct a cover, it potentially results in a cover exponentially larger than a minimal one. The key in our work is a novel bounded prime extraction (BPE) method that dynamically interleaves extracting prime implicates with extracting small implicates to construct a succinct cover. The intuition behind BPE is to find a good trade-off between having an efficient algorithm and constructing a succinct cover.

Following the assessment way of the SOTA work [14], we evaluate our approach on four classic benchmarks. The experimental results illustrate that CoAPI is faster than the SOTA approaches on at least 88% instances. Particularly, for generating all prime implicates, CoAPI is up to about one order of magnitude faster in 32% instances. Moreover, we add a benchmark sourcing from real-world industries [8]. Compared with classic benchmarks, the industrial benchmark is more challenging since it maintains millions of prime implicates or implicants. The results on the industrial benchmark confirm that the CoAPI significantly improves the SOTA approaches. At the same time, the results confirm that BPE can provide a good trade-off.

II. Preliminaries

In this section, we review the basics of prime compilation. The symbols in this section are standard in all.

A. Notation

Let $\mathcal{V} = \{v_1, \cdots, v_n\}$ be a set of *Boolean variables*. A *literal* is either a Boolean variable (*positive* literal) or its negation (*negative* literal). A *term* π is a conjunction of literals, represented as a set of literals. $|\pi|$ is the size of π, *i.e.*, the number of literals in π. An *assignment* over $\mathcal{V}$ is denoted by a term π such that, for any $v \in \mathcal{V}$, v is assigned to true, iff $v \in \pi$, and is assigned to false, iff $\neg v \in \pi$. An assignment cannot include both positive and negative literals of a variable. 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. A *clause* is the disjunction of literals. The size of a clause is the number of literals in it. A CNF formula is formed as a conjunction of clauses, which is denoted as a set of clauses.

Let φ be a non-clausal Boolean formula. $\mathrm{VAR}(\varphi)$ (*resp.* $\mathrm{LIT}(\varphi)$) denotes the set of variables (*resp.* literals) of φ. A *model* of φ is a full assignment satisfying φ. Let ϕ be a Boolean formula. We say $\phi \models \varphi$, iff all models of ϕ are also the models of φ. Particularly, a model is said to be *minimal* (*resp. maximal*), when it contains the minimal (*resp.* maximal) number of variables assigned true. Two Boolean formulae are *logically equivalent* iff they are satisfied by the same models.

B. Encoding Non-clausal Boolean Formulae

Non-clausal Boolean formulae are often transformed into CNF formulae by encoding methods, such as *Tseitin encoding* [11]. Let Σ_φ (*resp.* $\Sigma_{\neg\varphi}$) be a CNF formula encoding φ (*resp.* $\neg\varphi$) via adding a set of auxiliary variables $\mathcal{V}_a$. We call $\mathcal{V}_o = \mathrm{VAR}(\varphi) = \mathrm{VAR}(\neg\varphi)$ a set of *original* variables. Obviously, $\mathcal{V}_o \cup \mathcal{V}_a = \mathrm{VAR}(\Sigma_\varphi)$ and $\mathcal{V}_o \cap \mathcal{V}_a = \emptyset$. Since the additional variables, φ and Σ_φ are not logically equivalent. $|\Sigma_\varphi|$ means the sum of the size of clauses in Σ_φ. We take Tseitin encoding as an example.

Example 1. *Let* $\varphi = (a \wedge b) \vee (\neg a \wedge c)$ *be a non-clausal Boolean formula. Tseitin encoding adds three auxiliary variables to encode* φ*:* $g_1 \leftrightarrow g_2 \vee g_3$*,* $g_2 \leftrightarrow a \wedge b$*, and* $g_3 \leftrightarrow \neg a \wedge c$*. Tseitin encoding is* $\Sigma_\varphi = \{\{g_1\}, \{g_1 \vee \neg g_2\}, \{g_1 \vee \neg g_3\}, \{\neg g_1 \vee g_2 \vee g_3\}, \{g_2 \vee \neg a \vee \neg b\}, \{\neg g_2 \vee a\}, \{\neg g_2 \vee b\}, \{g_3 \vee a \vee \neg c\}, \{\neg g_3 \vee \neg a\}, \{\neg g_3 \vee c\}\}$*.* $\Sigma_{\neg\varphi}$ *is obtained by changing the first clause in* Σ_φ *to* $\{\neg g_1\}$*. Its* $\mathcal{V}_o$ *is* $\{a, b, c\}$ *and* $\mathcal{V}_a$ *is* $\{g_1, g_2, g_3\}$*.*

C. Dual Rail Encoding

For every variable $v \in \mathcal{V}_o$, there exist two corresponding variables – x_v represents the literal v; $x_{\neg v}$ represents the literal $\neg v$. Therefore, there are three possible assignments for v: (i) $\neg x_v \wedge \neg x_{\neg v}$ means v is a *don't care*; (ii) $\neg x_v \wedge x_{\neg v}$ means $v =$ false; (iii) $x_v \wedge \neg x_{\neg v}$ means $v =$ true, where $x_v \wedge x_{\neg v}$ is forbidden by the constraint $\neg x_v \vee \neg x_{\neg v}$. Intuitively, if a variable v is a don't care, there exists a partial assignment π such that $v \wedge \pi \models \varphi$ and $\neg v \wedge \pi \models \varphi$.

D. Prime Compilation

A clause λ is called an *implicate* of φ if $\varphi \models \lambda$. An implicate λ of φ is called *prime* if any subset $\lambda' \subsetneq \lambda$ is not an implicate of φ. A term κ is called an *implicant* of φ if $\kappa \models \varphi$. An implicant κ of φ is called *prime* if any subset $\kappa' \subsetneq \kappa$ is not an implicant of φ.

Prime compilation aims to compute all the prime implicates or implicants. Given an implicant π of φ, $\neg\pi$ is an implicate of $\neg\varphi$, and vice versa (*duality*). Therefore, we can use the negation of implicants of $\neg\varphi$ as implicates of φ to construct a cover of φ. Due to the duality of the implicates and implicants, we can compute all prime implicates of the formula via solving

all prime implicants of the negation of the formula. For brevity, we illustrate our method to compute all prime implicants.

III. Two-phase Prime Compilation

In this section, we illustrate our approach, CoAPI. We first give an overview of the two-phase framework. Then, we introduce constructing a cover in the first phase and enumerating all prime implicants in the second phase.

A. Overview: Two-phase Framework

Before describing the two-phase framework, we introduce the concept of cover.

Definition 1. *A Boolean formula Φ is a* cover *of φ if Φ is a conjunction of implicates of φ and is logically equivalent to φ. The* size *of a cover Φ, denoted by $|\Phi|$, is the sum of the size of implicates in Φ. A cover is* minimal *if all the implicates of the cover are prime.*

Intuitively, the cover Φ of φ is accord with the form of CNF and only consists of the variables in $\mathcal{V}_o$. In order to generate *all* prime implicants of φ, it is necessary to compute prime implicants of the cover of φ under the DR encoding [14]. Previti et al. (2015) combined the construction of cover and the generation of all prime implicants into a unified process under DR encoding [14]. Naturally, constructing a cover independently need not use DR encoding, which avoids enlarging the search space. Moreover, we can exploit the powerful polarity heuristic method for SAT solving in the first phase.

Based on the considerations above, we design a two-phase approach, namely CoAPI, to compile a non-clausal Boolean formula. CoAPI takes Σ_φ, $\Sigma_{\neg\varphi}$, and $\mathcal{V}_o$ as input, which is the same as the approach [14]. CoAPI returns all prime implicants Π of φ. The two-phase framework is summarized as follows.

1) It first, using original encoding, computes a set of implicates with small size to construct a succinct cover Φ in the first phase (CompileCover);
2) then, using DR encoding, generates all prime implicants Π based on Φ in the second phase (CompileAll).

B. First Phase: Constructing a Succinct Cover

In this subsection, we first introduce CompileCover, and then discuss some ideas to extract implicates. Algorithm 1 shows the pseudo-code of CompileCover. It maintains a set of blocking clauses $\mathcal{B}$ to avoid searching for the same implicate. It keeps iterating until a cover has been constructed (line 9). At each iteration, it first searches for a model π of $\Sigma_{\neg\varphi}$ which is not blocked by $\mathcal{B}$ (line 3). Then, it extracts a partial assignment π_p, such that $\pi_p \models \neg\varphi$ (lines 5-6). Finally, it updates Φ and $\mathcal{B}$ by a blocking clause $\neg\pi_p$ (line 7).

Although CompileCover is inspired by the well-known blocking clause approach [16], it holds two additional constraints as follows: (i) it needs to extract implicates of φ and (ii) the size of the implicates is required to be small. The first constraint is straightforward. If π is a model (a full assignment) of $\Sigma_{\neg\varphi}$, after removing all the literals corresponding to $\mathcal{V}_a$ from π, $\pi_{\mathcal{V}_o} \models \neg\varphi$ holds. Therefore, $\pi_{\mathcal{V}_o}$ (line 5) is a model of $\neg\varphi$. However, the second constraint is challenging. If it directly uses $\neg\pi_{\mathcal{V}_o}$ as an implicate of φ to construct a cover, the size of the cover is exponentially larger than the size of φ potentially. We will introduce a novel extraction method BPE (line 6) in Section IV.

Algorithm 1: CompileCover

```
   Input  : Σ_φ, Σ_¬φ, and V_o
   Output : A cover Φ of φ
1  Φ ← ∅, B ← ∅
2  while true do
3      (st, π) ← SAT(Σ_¬φ ∪ B)
4      if st is SAT then
5          π_{V_o} ← {l ∈ π| VAR(l) ∈ V_o} /* π_{V_o} ⊨ ¬φ */
6          π_p ← BPE(Σ_φ,π_{V_o}) /* π_{V_o} ⊨ π_p, π_p ⊨ ¬φ*/
7          Φ ← Φ ∪ {¬π_p}, B ← B ∪ {¬π_p}
8      else
9          return Φ
```

We establish the correctness of Algorithm 1 as follows.

Theorem 1. *Let φ be a non-clausal Boolean formula, Σ_φ a CNF formula encoding φ, $\Sigma_{\neg\varphi}$ a CNF formula encoding $\neg\varphi$, and $\mathcal{V}_o$ a set of variables of φ.* CompileCover *terminates and Φ =* CompileCover*(Σ_φ,$\Sigma_{\neg\varphi}$,$\mathcal{V}_o$) is a cover of φ.*

Sketch of proof. We first prove that if CompileAll terminates, Φ is a cover of φ. If CompileCover terminates, $\Sigma_{\neg\varphi} \wedge \mathcal{B}$ is unsatisfiable. It means that $\neg\varphi \wedge \mathcal{B}$ is unsatisfiable because $\neg\varphi$ is satisfiability equivalent to $\Sigma_{\neg\varphi}$. Due to $\mathcal{B} = \Phi$ and $\text{VAR}(\mathcal{B}) \subseteq V_E$, $\Phi \models \varphi$ holds. For each clause $c \in \Phi$, because $\neg c$ is an implicant of $\neg\varphi$ ($\neg c \models \neg\varphi$), we have $\varphi \models c$. Therefore, $\varphi \models \Phi$. Obviously, Φ is of the form CNF. So Φ is a cover of φ.

We are now going to prove that if Φ is a cover of φ, CompileAll terminates. Obviously, when $\mathcal{B}$ is logically equivalent to φ, *i.e.*, Φ is a cover of φ, $\neg\varphi \wedge \mathcal{B}$ is unsatisfiable where CompileCover terminates. □

We use an example to illustrate CompileCover.

Example 2 (Example 1 continued)**.** *In the first iteration, a model $\pi = \neg g_1 \wedge \neg g_2 \wedge \neg g_3 \wedge \neg a \wedge b \wedge \neg c$ of $\Sigma_{\neg\varphi}$ is found; then we extract a small implicant $\pi_p = \neg a \wedge \neg c$ of $\neg\varphi$; afterward, an implicate $a \vee c$ of φ is obtained. In the second iteration, we can obtain an additional implicate $\neg a \vee b$. Afterward, $\Sigma_{\neg\varphi} \cup \{a \vee c, \neg a \vee b\}$ is unsatisfiable. In total,* CompileCover *returns a cover $\Phi = (a \vee c) \wedge (\neg a \vee b)$.*

C. Second Phase: Generating All Prime Implicates/Implicants

In the second phase, based on the cover Φ, we design CompileAll to generate all prime implicants. We reformulate the prime implicants generation problem as the minimal model enumeration problem. Algorithm 2 shows the pseudo-code of CompileAll. In this way, we build up $\mathcal{H} = \mathcal{P} \cup \mathcal{L}_-$ by DR encoding (line 2), where each clause $c_i = l_1 \vee ... \vee l_k$ in Φ is transformed into a clause $c_i^H = x_{l_1} \vee ... \vee x_{l_k}$, denoted by $\mathcal{P}$, and $\mathcal{L}_-$ includes the clauses $\neg x_v \vee \neg x_{\neg v}$ for every variable

$v \in \mathcal{V}_o$. At each iteration, firstly, we search a minimal model[3] π^H of $\mathcal{H} \cup \mathcal{B}$ where $\mathcal{B}$ is a set of blocking clauses. Secondly, we extract a partial model π^H_+ containing all positive literals in π^H (lines 6-7). Finally, Π and $\mathcal{B}$ are updated (line 8). Note that $\bigwedge_{x_{l_i} \in \pi^H_+} l_i$ is a prime implicant of φ since it satisfies all the clauses in Φ using a minimal number of literals. When CompileAll terminates ($\mathcal{H} \cup \mathcal{B}$ is unsatisfiable), Π includes all prime implicants of φ (line 10).

Algorithm 2: CompileAll

```
   Input  : Φ, V_o
   Output : All prime implicants Π
1  Π ← ∅, B ← ∅
2  H ← EncodeDualRail(Φ,V_o)
3  while true do
4      (st, π^H) ← MinimalSAT(H ∪ B)
5      if st is SAT then
6          π^H_+ ← {l ∈ π^H | l is a positive literal}
7          π^F ← ⋀_{x_{l_i} ∈ π^H_+} l_i
8          Π ← Π ∪ {π^F}, B ← B ∪ {¬π^H_+}
9      else
10         return Π
```

This idea is similar to the work [17]. However, we observe that a large number of the constraints expressing the minimal model will hang up for memory problems and severely reduce performance for Boolean constraint propagation (BCP) in SAT solving, particularly for large cases. Therefore, we do not use the constraints establishing a bijection between prime implicants and the models.

We establish the correctness of CompileAll as follows.

Theorem 2. *Let φ be a non-clausal Boolean formula, Φ a cover of φ, and $\mathcal{V}_o$ a set of variables of φ.* CompileAll *terminates and* Π = CompileAll*(Φ,$\mathcal{V}_o$) includes all prime implicants of φ.*

Sketch of proof. We first prove that if there exists a prime implicant of φ which has not been found, $\mathcal{H} \cup \mathcal{B}$ is satisfiable. Without loss of generality, suppose that there is a single prime implicant π^F not in Π. π^F corresponds to a partial assignment $\pi^H = x_1 \wedge ... \wedge x_m$ for $\mathcal{H}$. We extend π^H as a full assignment $\pi^H_+ = x_1 \wedge ... \wedge x_m \wedge \neg x_{m+1} \wedge, ..., \wedge \neg x_n$. Due to the π^F is a prime implicant of Π, π^H_+ satisfies $\mathcal{H}$. Moreover, every pair of prime implicants differs in at least one variable on $\mathcal{H}$. This means that at least one literal of each clause in $\mathcal{B}$ is in the set of literals $\{\neg x_{m+1}, ..., \neg x_n\}$. Therefore, π^H_+ also satisfies $\mathcal{B}$.

We are now going to prove that if all prime implicants of φ are computed, CompileAll terminates. Because of the duality between the prime implicate and prime implicant, $\neg\mathcal{B}$ is logically equivalent to $\mathcal{P}$ when all prime implicants of φ are computed, in which $\mathcal{H} \cup \mathcal{B}$ is unsatisfiable. □

Recall Example 2, we compute all prime implicants.

[3]In an SAT solver, *e.g.* MiniSAT, we can set the polarity of all variables to be false.

Example 3 (Example 2 continued)**.** CompileAll *encodes Φ as $\mathcal{H} = (x_a \vee x_c) \wedge (x_{\neg a} \vee x_b) \wedge (\neg x_a \vee \neg x_{\neg a}) \wedge (\neg x_b \vee \neg x_{\neg b}) \wedge (\neg x_c \vee \neg x_{\neg c})$. In the first iteration, a minimal model $\pi = \neg x_a \wedge x_{\neg a} \wedge \neg x_b \wedge \neg x_{\neg b} \wedge x_c \wedge \neg x_{\neg c}$ is found. Therefore, $\pi_p = x_{\neg a} \wedge x_c$ and we get a prime implicant $\neg a \wedge c$. Finally, after similar steps, we obtain all prime implicants $\neg a \wedge c$, $b \wedge c$, and $a \wedge b$.*

IV. Bounded Prime Extraction

In this section, we first introduce a novel method to extract a small implicant; and then analyze its performance.

In order to extract small implicants of $\neg\varphi$ to construct the cover of φ, we introduce the concept of a *necessary* literal.

Definition 2. *Given an implicant π of $\neg\varphi$, $l \in \pi$ is* necessary *for π if it fulfills the requirement that $\bigwedge_{l_i \in \pi, l_i \neq l} l_i \wedge \Sigma_\varphi$ is satisfiable.*

We can check whether $\bigwedge_{l_i \in \pi, l_i \neq l} l_i \wedge \Sigma_\varphi$ is satisfiable with one SAT query. A straightforward idea to extract a small implicant is to remove all the unnecessary literals from a model. It keeps a minimal size of the implicant, *i.e.*, prime implicant, to construct a minimal cover. A naive method, namely linear method, linearly query the necessity of all literals in a model. Alternatively, QuickXplain [7] (QX) expands the checking of the necessity of a single literal to part of literals. It recursively splits an implicant and checks the necessity of the partial literals. So QX requires exponentially fewer queries in the optimal case than the linear method.

However, it is time-consuming to construct a minimal cover because there are considerable SAT queries to guarantee that each implicant is minimal. In addition, although a minimal cover can result in a greatly succinct representation, it is not clear whether the greatly succinct representation is cost-effective in two phases. Therefore, we design BPE to find a good trade-off between having an efficient algorithm and constructing a succinct cover.

Algorithm 3: BPE

```
   Input  : Σ_φ and a model π of ¬φ
   Output : An implicant π_p of ¬φ
1  (st, π') ← AssumpSAT(Σ_φ, π) /* PreE */
2  /* sup and inf are global variables and are initialized as the size
      of the first prime implicant obtained. */
3  if |π'| > sup then
4      π_p ← LinearExtract(Σ_φ, π') /* PE */
5  else if |π'| > inf then
6      π_p ← IntervalExtract(Σ_φ, π') /* SE */
7  else
8      π_p ← π'
9  /* update boundary */
10 if |π_p| > sup then
11     sup ← |π_p|
12 else if |π_p| < inf then
13     inf ← |π_p|
14 return π_p
```

Algorithm 3 summarizes BPE. Depending on the size of the obtained implicants (line 1), BPE dynamically returns

either a prime (line 4) or a small implicant (line 6 and 8) that subsumes it. The decision between these two cases is done by keeping track of the maximum and minimum sizes of implicants already in the cover, stored in the global variables sup and inf respectively. Intuitively, the small size of different instances is distinct, so we use sup and inf to dynamically measure the small size for different instances.

We define three operations – *preliminary extraction (*PRE*)*, *prime extraction (*PE*)*, and *small extraction (*SE*)* to extract implicants in different levels.

In PRE, we extract an implicant via querying a SAT solver whether Σ_φ is satisfiable under a set of assumptions π. A set of *failed assumptions* [18] π' is a new implicant with the same or a smaller size of π. Based on the size of π', we select executing PE or SE to further shrink π'.

If $|\pi'| > sup$ (PE), in order to control the size of the cover, we must reduce the size of the implicant or prove that a larger prime implicant exists. Although QX is asymptotically optimal in the number of SAT queries to extract a prime implicant, it potentially requires more queries than the linear method when most of the literals in an implicant are necessary [7]. Moreover, using the failed assumptions, a SAT solver potentially obtains an implicant with most of the necessary literals. Therefore, we expect the better performance of extracting prime implicants using the linear method, than using QX.

Algorithm 4: INTERVALEXTRACT

```
   Input  : Σ_φ and an implicant π of ¬φ
   Output : An implicant π_p of ¬φ
1  π_p ← π
2  if |π_p| is 1 then
3      return π_p
4  else
5      (π_l, π_r) ← PARTITION(π_p)
6      (st, π') ← ASSUMPSAT(Σ_φ, π_l)
7      if st is UNSAT then                    /* φ ⊨ ¬π' */
8          return INTERVALEXTRACT(Σ_φ, π')
9      else
10         (st, π') ← ASSUMPSAT(Σ_φ, π_r)
11         if st is UNSAT then                /* φ ⊨ ¬π' */
12             return INTERVALEXTRACT(Σ_φ, π')
13         else
14             return π_p
```

If $inf < |\pi'| \leqslant sup$ (SE), we relax the requirement for the small size and pursue high performance. To this end, we design INTERVALEXTRACT (Algorithm 4) to extract a small implicant. It first partitions π into two partial assignments (π_l and π_r) with the same size (line 5). Then, based on a divide and conquer, checks whether π_l (line 6) or π_r (line 10) is an implicant of $\neg\varphi$. If at least one is an implicant, we obtain a new implicant π' (half size of π) and continue to partition π' (lines 8 and 12). Compared with QX, INTERVALEXTRACT avoids discussing the case (line 14), in which neither π_l nor π_r is an implicant, to cut down the time consumption.

The following theorems show that BPE can achieve a good trade-off between having an efficient algorithm and constructing a succinct cover.

Theorem 3. *In Algorithm 3, π_p is smaller than or equal in size to the largest prime implicant of $\neg\varphi$.*

Sketch of proof. Because sup is only updated when extracting a prime implicant (Alg.2 of line 3), sup indicates the size of obtained largest prime implicants. Therefore, π_p is smaller than or equal in size to the largest prime implicant of $\neg\varphi$. □

Theorem 3 avoids producing the exponential number of implicants to construct a cover in the worst case where the size of the implicants is close to the models.

Theorem 4. *In Algorithm 3, if $|\pi'| > sup$,* BPE *requires $\mathcal{O}(|\pi'|)$ SAT queries; if $inf < |\pi'| \leqslant sup$, requires $\mathcal{O}(\lg \frac{|\pi'|}{|\pi_p|})$; otherwise requires $\mathcal{O}(1)$.*

Sketch of proof. If $|\pi'| > sup$, we utilize the linear method. It needs to call SAT solver $|\pi'|$ times to query whether each literals in π' is necessary. If $inf < |\pi'| \leqslant sup$, we utilize INTERVALEXTRACT. In the worst case, all π_l is not an implicant of $\neg\varphi$, while π_r is that, until the size of π_r is 1. It needs to call SAT solver $\lg \frac{|\pi'|}{|\pi_p|}$ times. Otherwise, we accept π' as result, where it only needs one SAT query (line 1). □

Theorem 4 maintains the high performance of BPE. In the worst case ($|\pi'| > sup$), although we require $\mathcal{O}(|\pi'|)$ SAT queries, we obtain a prime implicant helping to construct a succinct cover. Otherwise ($|\pi'| \leqslant sup$), we only require fewer SAT queries, which usually exhibit better performance, and obtain an acceptable size.

V. EXPERIMENTAL RESULTS

Following the assessment way of the SOTA work [14], we considered the research questions as follows.

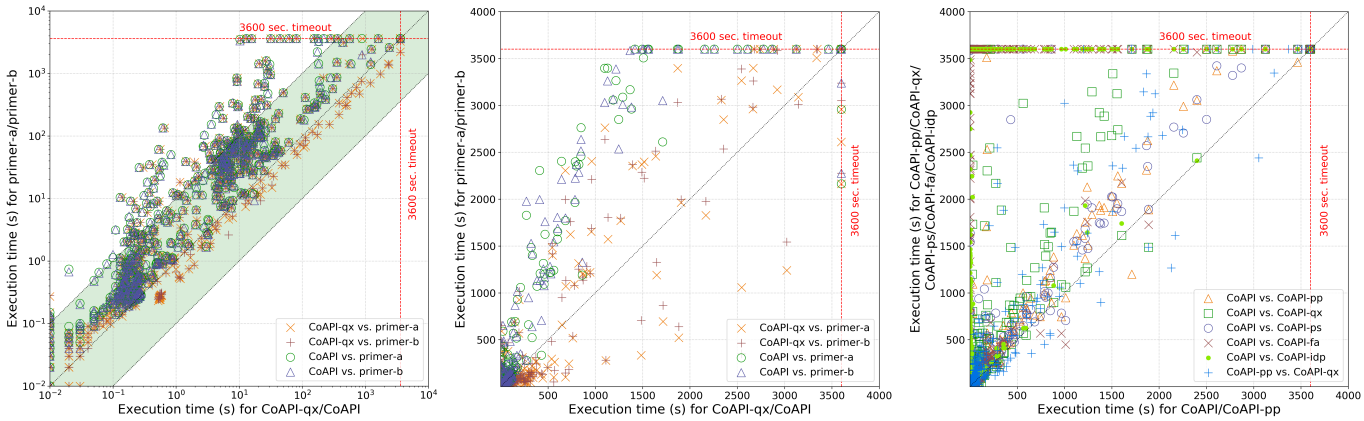
RQ 1. *What is the performance of* `CoAPI` *compared with the SOTA approaches for prime implication of non-clausal Boolean formulae?*

RQ 2. *Can* `CoAPI` *outperform the SOTA approaches in the real-world industrial benchmark?*

RQ 3. *Can* BPE *keep a good trade-off between having an efficient algorithm and constructing a succinct cover?*

Benchmarks. We evaluated `CoAPI` on two kinds of benchmarks. The classic benchmarks are introduced by Previti et al. (2015), namely *QG6*, *Geffe gen.*, *F+PHP*, and *F+GT*. The instances in the four benchmarks, from quasigroup classification problems (*QG6*), cryptanalysis of the Geffe stream generator (*Geffe gen.*), and crafted formulae (*F+PHP* and *F+GT*), have fairly large numbers of variables but without a large number of prime implicates/implicants.

The industrial benchmark *FT* [8] comes from real-world safety-critical systems in the areas such as the aerospace and nuclear power industries. Compared with classic benchmarks, *FT* is more challenging since it maintains millions of prime implicants (details in Table II). In this area, a *fault tree* can be represented as a non-clausal Boolean formula, in which



(a) Generating prime implicates. (b) Generating prime implicants. (c) Generating prime implicates & implicants.

Fig. 1. CPU time comparison of `primer-a`, `primer-b`, `CoAPI` and the variants of `CoAPI`. The X-axis and Y-axis represent the execution time in seconds for prime compilation. Note that (a) is under log coordinates.

basic events characterize the failure of system components and *internal events* characterize the failure of the subsystem. Computing prime implicants of a fault tree, *i.e.*, finding all minimal combinations of the status of system components resulting in a system failure, is a fundamental step in fault tree analysis [19]. A prime implicant characterizes the root cause of a system failure, so all prime implicants provide important information about all vulnerabilities of a system [20]. Also, Contini et al. (2008) claimed that the significance of computing all prime implicants lies in the complete analysis of the conditions leading to the system failure, particularly for the safety-critical systems [21].

State-of-the-art competitors. For answering RQ1 and RQ2, we compared `CoAPI` with the SOTA approach [14] to compiling a non-clausal Boolean formula, namely `primer`. It has two variants based on different search strategies, namely `primer-a` and `primer-b`. Note that the approaches based on binary decision diagrams (BDDs) [22] were not tried since they require building a BDD for a formula and it is unable to construct a BDD for the most of formulae within 1 hour or it is out of memory.

Because of the inherent intractability of compiling a non-clausal Boolean formula and the large scale of a fault tree in reality, the SOTA approaches to compiling a fault tree only consider some approximate solutions, such as XFTA [23] and SATMCS [8]. Therefore, we did not compare with them for computing exact prime implicants. For the exact approaches [24], they are based on BDDs. We also did not report their results since they were unable to construct a BDD for most instances because of the exponential memory cost during building a BDD.

For answering RQ3, we compared `CoAPI` with its variants. We categorized the variants of `CoAPI` into three classes based on different extraction methods. They are only different in extracting implicants to construct a cover in CompileCover.

The first one (our method) uses BPE – `CoAPI`. It interleaves extracting prime implicates with extracting small implicates to construct a cover.

The second class of methods is only to extract prime implicants, including `CoAPI-pp` and `CoAPI-qx`.

- `CoAPI-pp` first utilizes PreE, then only exploits PE.
- `CoAPI-qx` uses QX.

The third class of methods is only to extract small implicants including `CoAPI-ps`, `CoAPI-fa`, and `CoAPI-idp`.

- `CoAPI-ps` first uses PreE, then only exploits SE.
- `CoAPI-fa` iteratively invokes a SAT solver holding failed assumptions [25, 26]. In this way, we keep invoking a SAT solver until the size of the implicant is unchanged. It can be seen as an extension of the idea utilizing the consecutive dual propagation [26].
- `CoAPI-idp` utilizes the interleaved dual propagation [27].

Settings. We implemented `CoAPI` and its variants utilizing MiniSAT [28] that was also used to implement, `primer-a`, and `primer-b`. For each instance, we considered two tasks: (1) generating all prime implicates; (2) generating all prime implicants. Due to the duality of the implicates and implicants, we generated all prime implicates of formula via solving all prime implicants of the negation of the formula. Note that there are not all prime implicates in the prime cover for `primer-a` and `primer-b`. In order to obtain all prime implicates, we must also invoke `primer-a` and `primer-b` to compute all prime implicants of the negation of the formula. The time limit was 1 hour and the memory limit was 10 GByte for each instance. All the experiments were performed on an Intel Core i5-7400 3 GHz, with 20 GByte of memory and running Ubuntu 18.04. The results of each approach reported is the mean data of 10 repeated experiments.

A. Comparison on The Classic Benchmarks

In order to assess the two-phase framework, we additionally compared `CoAPI-qx` with `primer`. `CoAPI-qx` uses QX to construct a prime cover, which is the same as `primer`, and only differs from `primer` in the two-phase framework.

TABLE I
THE NUMBER OF INSTANCES COMPUTED (TASK (I) / TASK (II)).

	QG6 (83)	*Geffe gen.* (600)	*F+PHP* (30)	*F+GT* (30)	*Total* (743)
primer-a	30 / 68	577 / **596**	30 / 30	28 / 30	665 / 724
primer-b	30 / 67	578 / **596**	30 / 30	28 / 30	666 / 723
CoAPI-qx	30 / 73	**589** / 593	30 / 30	26 / 30	675 / 726
CoAPI	30 / **82**	**589** / 595	30 / 30	**30** / 30	**679** / **737**

Table I shows the number of instances that can be computed successfully, where **boldface** numbers refer to the better results. Overall, CoAPI-qx, primer-a, and primer-b are comparable. CoAPI successfully computes more instances than primer-a and primer-b. Particularly, CoAPI increases the number of instances in *QG6* for the task (ii) and in *Geffe gen.* for the task (i).

More details are shown in Figure 1(a) and (b) where the points above the diagonal indicate advantages for CoAPI-qx or CoAPI. For task (i), CoAPI computes much faster than primer-a (*resp.* primer-b) in 664 (*resp.* 666) instances – it is at least one order of magnitude faster than primer-a (*resp.* primer-b) in 214 (*resp.* 215) instances. For task (ii), CoAPI dominates primer-a and primer-b on *QG6*, *F+PHP*, and *F+GT*. Particularly, for *QG6*, CoAPI reduces execution time for at least 53% (*resp.* 50%) compared with primer-a (*resp.* primer-b). Obviously, CoAPI greatly improves the efficiency of the task (i). The reason is that most of the literals in an implicant are necessary, primer-a or primer-b requires significantly more SAT queries for extracting a prime implicant. Therefore, it seriously reduces the performance of primer-a and primer-b.

CoAPI-qx is also better than primer-a and primer-b. It is in 70% (*resp.* 67%) instances that CoAPI-qx beats primer-a (*resp.* primer-b). The advantage of CoAPI-qx is due to the two-phase framework, which avoids enlarging the search space and benefits from the polarity heuristic. In addition, CoAPI is greatly better than CoAPI-qx because of adopting BPE, which we will discuss in Section V-C.

Since primer-a, primer-b, CoAPI, and the variants of CoAPI are comparable in memory cost for prime compilation, we omit the report about memory. The comparable performance in space of these approaches results from the blocking clause framework for enumerating all results.

To summarize, the better performance of CoAPI-qx than the SOTA approaches confirms that the two-phase framework is efficient. Moreover, CoAPI outperforms SOTA approaches and CoAPI-qx because of the two-phase framework and the novel bounded prime extraction.

B. Comparison on The Industrial Benchmark

Following the settings of Section V-A, we evaluate CoAPI-qx and CoAPI on the industrial benchmark. Table II illustrates the results, where "N/A" indicates the results out of the CPU time limit and if the time usage is less than 0.01s, it is marked as 0.00. Overall, CoAPI-qx and CoAPI can successfully solve 16 instances which include all the 15 (*resp.* 15) instances successfully computed by primer-a (*resp.* primer-b). For most cases, CoAPI uses significantly less CPU time than primer-a and primer-b.

TABLE II
CPU TIME (S) ON THE INDUSTRIAL BENCHMARKS

Instances	#BE[1]	#IE[2]	#PI[3]	primer-a	primer-b	CoAPI-qx	CoAPI
baobab3	80	107	24,386	N/A	N/A	N/A	N/A
chinese	25	36	392	0.02	0.01	**0.00**	**0.00**
das9201	122	82	14,217	164.95	163.68	145.60	**83.00**
das9202	49	36	27,778	62.69	60.21	28.68	**28.62**
das9203	51	30	16,200	0.40	0.33	**0.24**	**0.24**
das9204	53	30	16,704	2.18	**0.68**	0.70	0.83
das9205	51	20	17,280	0.25	0.29	**0.02**	0.03
das9206	121	112	19,518	256.08	331.06	**44.78**	72.56
das9207	276	324	25,988	N/A	N/A	N/A	N/A
das9208	103	145	8,060	5.96	6.35	6.06	**1.17**
das9209	109	73	8.2E10	N/A	N/A	N/A	N/A
edf9201	183	132	579,720	1,164.51	1,100.92	830.85	**742.67**
edf9202	458	435	130,112	N/A	N/A	N/A	N/A
edf9203	362	475	20,807,446	N/A	N/A	N/A	N/A
edf9204	323	375	32,580,630	N/A	N/A	N/A	N/A
edf9205	165	142	21,308	86.82	**43.57**	46.68	53.16
edf9206	240	362	N/A	N/A	N/A	N/A	N/A
edfpa14b	311	290	105,955,422	N/A	N/A	N/A	N/A
edfpa14o	311	173	105,927,244	N/A	N/A	N/A	N/A
edfpa14p	124	101	415,500	N/A	N/A	N/A	N/A
edfpa14q	311	194	105,950,670	N/A	N/A	N/A	N/A
edfpa14r	106	132	380,412	N/A	N/A	N/A	N/A
edfpa15b	283	249	2,910,473	N/A	N/A	N/A	N/A
edfpa15p	276	324	27,870	N/A	N/A	1,061.29	**1,012.33**
edfpa15r	88	110	26,549	2,755.73	2,872.39	1,087.08	**967.62**
elf9601	145	242	151,348	N/A	N/A	N/A	N/A
ftr10	175	94	305	14.11	13.82	15.44	**1.24**
isp9602	116	122	5,197,647	N/A	N/A	N/A	N/A
isp9603	91	95	3,434	570.37	565.98	568.08	**80.90**
isp9604	215	132	746,574	N/A	N/A	N/A	N/A
isp9606	89	41	1,776	72.19	77.73	33.61	**13.69**
isp9607	74	65	150,436	452.26	540.61	273.99	**262.09**
jbd9601	533	315	14,007	N/A	N/A	N/A	N/A
#win				0	2	4	**12**

[1] the number of basic events.
[2] the number of internal events.
[3] the number of prime implicants.

In summary, CoAPI has an excellent ability in generating all prime implicants on the benchmarks of fault tree compared with the SOTA approaches. At the same time, we also notions that *FT* is challenging for all approaches: only about half of the instances can be computed. The reason is that a large number of blocking clauses resulting from lots of prime implicants reduces the performance of BCP. Our future work will optimize CoAPI to handle more industrial instances.

C. Comparison with Other Extraction Methods

To evaluate our method, we compared CoAPI with its variants on the five benchmarks: *QG6*, *Geffe gen.*, *F+PHP*, *F+GT*, and *FT*. Overall, the most points are above the diagonal line, which represents CoAPI is faster in most instances. Both CoAPI-pp and CoAPI are also faster than CoAPI-qx, which confirms our analysis about the weakness of QX in Section IV. Besides, either only extracting prime implicants or only extracting small implicants cannot outperform the combination, which shows that BPE does play a vital role.

The more detail statistics are shown in Table III, where the statistics present the ratio of the average results of a CoAPI's variant to that of CoAPI and we use underline to negative results for CoAPI. Compared with the methods to only extract small implicants, CoAPI constructs a more succinct cover than any other method that only extracts small implicants, particularly for task (ii). And, CoAPI does not suffer from the

performance issue resulting from the slightly larger cover in CompileAll. In addition, compared with the methods to only extract prime implicants, CoAPI requires fewer SAT queries, which speeds up in CompileCover. Above comparisons show that BPE provides a better trade-off between having an efficient algorithm and constructing a succinct cover.

TABLE III
Results of extracting methods (task (i) / task (ii)).

	$\|\Sigma\|$[1]	#SAT[2]	T. in CC[3]	T. in CA[4]
CoAPI-pp	1.00 / 0.45	6.82 / 1.44	4.31 / 1.26	1.36 / 0.88
CoAPI-qx	1.00 / 0.33	11.38 / 1.57	4.50 / 2.27	0.96 / 1.18
CoAPI-ps	1.00 / 3.31	1.60 / 2.25	1.54 / 1.45	1.30 / 4.58
CoAPI-fa	1.00 / 93.62	0.96 / 114.19	1.73 / 700.09	1.36 / 5.73
CoAPI-idp	4.60 / 19525.32	0.75 / 609.71	2.86 / 121.83	1.24 / 7049.41

[1] the size of the cover. The statistic is lager than 1 means that CoAPI is more succinct.
[2] the number of SAT queries in CompileCover. The statistic is lager than 1 means that CoAPI requires fewer SAT queries.
[3] the execution time in CompileCover. The statistic statistic is lager than 1 means that CoAPI uses less time for constructing a cover.
[4] the execution time in CompileAll. The statistic is lager than 1 means that CoAPI uses less time for generating all prime implicants.

Note that the methods to extract prime implicants produce the smallest average size of the cover than any other method. However, their performance is obviously inferior to CoAPI. It confirms that finding a minimal cover is not always good for the whole task. The reason is that the cost of constructing minimal cover is too high. Although CoAPI-pp and CoAPI-qx are comparable in the average size of the cover, CoAPI-pp generally uses fewer SAT queries than CoAPI-qx, which results in a lower time cost. It also illustrates that QX often reaches the worst case in extracting prime implicants.

In summary, we verify that our extraction algorithm (BPE) can find a good trade-off between having an efficient algorithm and constructing a succinct cover.

VI. Related Work

A. Prime Compilation

A large family [29, 30, 31] is based on 0-1 integer linear programming to compile a Boolean formula. Motivated by the work [31], Jabbour et al. (2014) proposed a SAT-based reformulation approach [17]. Some other works [32, 33, 34] exploited the adaptations of DPLL-like procedure.

The above works only focuses on compiling a clausal formula. With the introduction of new technologies, it is possible to efficiently generate all prime implicates/implicants of a non-clausal formula. Such as, Ngair (1993) studied an incremental algorithm working on a more general input [35]. Ramesh et al. (1997) used the negation normal form with path dissolution [36]. Matusiewicz et al. (2009) proposed prime implicate trie to encode prime implicates [13]. In addition, a number of approaches were based on BDDs [22] or zero-suppressed BDDs [24]. These approaches, however, still suffer from time or memory limitations in practice.

Recently, Previti et al. (2015) described the SOTA approach [14]. Their work shares high-level motivation with our work, but differs with respect to how it is realized, constructing a prime cover and finds all prime implicants simultaneously. Our experiments show that our approach is much better than their approach. In the rare case where all covers of a formula require an exponential number of implicates, our approach will time out in the first phase. Although the approach [14] can output partial results, they cannot return all the results because they also rely on the cover.

B. Extracting Small Implicants

Dual propagation. Goultiaeva et al. (2013) proposed interleaved dual propagation for QBF solving [27]. We can use the interleaved dual propagation to detect partial models early in a SAT solving. Niemetz et al. (2014) proposed consecutive dual propagation [26]. To extract implicants, this method is similar to the work [25]. Zhang and Malik (2003) took multiple SAT queries to extract a small implicant with different failed assumption orders to extract a small implicant [25]. Our experiments show that these methods based on dual propagation cannot produce small implicants in some cases.

Unsatisfiable sets/subformula. Extracting a small implicant is a special case of computing a small/minimal unsatisfiable set/subformula (US). To this end, each literal of an implicant is treated as a unit clause, and a small implicant can be obtained by extracting the unit clauses in a small/minimal US.

The results of methods proposed by Gershman et al. (2006) (small US) and Mencía et al. (2019) (minimal US) were not shown in experiments, since they are unable to construct a cover for most of the formulae within 1 hour. It is reasonable that computing a special case using a general algorithm often leads to poor performance.

C. All-SAT Problem

Although the blocking clause approach is to solve All-SAT problem, in our work, the assignment must fulfill the additional constraints. Therefore, All-SAT problem is computationally less expensive than the problem we tackled. We have tried to modify the All-SAT solver [39] to construct the cover in the first phase. Due to the limitation of extraction algorithms, they hung up for memory problems for most instances.

VII. Conclusion and Future Work

For prime compilation of non-clausal Boolean formulae, we have proposed a two-phase approach – CoAPI and a dynamical extraction method – BPE. Note that our approach can handle not only non-clausal but also clausal formulae. Our experimental results show that CoAPI has dramatically pushed the limits of the performance of the state-of-the-art approaches on the classic and industrial benchmarks. Moreover, the results confirm that BPE provides a good trade-off between having an efficient algorithm and constructing a succinct cover. Future work will develop optimization techniques to improve performance for the industrial benchmark.

Acknowledgment

We thank Yongmei Liu and Liangda Fang for discussion on the paper and anonymous referees for helpful comments.

REFERENCES

[1] P. Marquis, "Knowledge compilation using theory prime implicates," in *IJCAI*, 1995, pp. 837–845.

[2] A. Ignatiev, A. Previti, and J. MarquesSilva, "Sat-based formula simplification," in *SAT*, 2015, pp. 287–298.

[3] J. Echavarria, S. Wildermann, and J. Teich, "Design space exploration of multi-output logic function approximations," in *ICCAD*, 2018, pp. 52–59.

[4] P. Tison, "Generalized consensus theory and applications to the minimization of boolean circuits," *IEEE Trans. on Computers*, vol. 16, no. 4, pp. 446–456, 1967.

[5] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013, pp. 779–786.

[6] A. Raghuvanshi and M. A. Perkowski, "Logic synthesis and a generalized notation for memristor-realized material implication gates," in *ICCAD*, 2014, pp. 470–477.

[7] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *FMCAD*, 2007, pp. 173–180.

[8] W. Luo and O. Wei, "Wap: Sat-based computation of minimal cut sets," in *ISSRE*, 2017, pp. 146–151.

[9] P. J. Stuckey, "There are no cnf problems," in *SAT*, 2013, pp. 19–21.

[10] C. E. Shannon, "The synthesis of two-terminal switching circuits," *The Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, 1949.

[11] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," *SCMML*, pp. 234–259, 1983.

[12] P. Liberatore, "Redundancy in logic i: Cnf propositional formulae," *AIJ*, vol. 163, no. 2, pp. 203–232, 2005.

[13] A. Matusiewicz, N. V. Murray, and E. Rosenthal, "Prime implicate tries," in *TABLEAUX*, 2009, pp. 250–264.

[14] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva, "Prime compilation of non-clausal formulae," in *IJCAI*, 2015, pp. 1980–1987.

[15] R. E. Bryant, D. B., K. B., K. Cho, and T. Sheffler, "Cosmos: A compiled simulator for mos circuits," in *DAC*, 1987, pp. 9–16.

[16] K. L. McMillan, "Applying sat methods in unbounded symbolic model checking," in *CAV*, 2002, pp. 250–264.

[17] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi, "Enumerating prime implicants of propositional formulae in conjunctive normal form," in *JELIA*, 2014, pp. 152–164.

[18] N. E. and N. Sörensson, "Temporal induction by incremental sat solving," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[19] W. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," NRC Washington DC, Tech. Rep., 1981.

[20] 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.

[21] S. Contini, G. Cojazzi, and G. Renda, "On the use of non-coherent fault trees in safety and security studies," *Reliab. Eng. Syst. Saf.*, vol. 93, no. 12, pp. 1886–1895, 2008.

[22] O. Coudert and J. C. Madre, "Implicit and incremental computation of primes and essential primes of boolean functions," in *DAC*, 1992, pp. 36–39.

[23] A. Rauzy, "Anatomy of an efficient fault tree assessment engine," in *PSAM*, 2012, pp. 25–29.

[24] S. Contini and V. Matuzas, "Reduced zbdd construction algorithms for large fault tree analysis," *Reliability, Risk and Safety - Back to the Future. UK: Taylor & Francis Group*, pp. 898–906, 2010.

[25] L. Zhang and S. Malik, "Searching for truth: Techniques for satisfiability of boolean formulas," Ph.D. dissertation, Princeton University Princeton, 2003.

[26] A. Niemetz, M. Preiner, and A. Biere, "Turbo-charging lemmas on demand with don't care reasoning," in *FMCAD*, 2014, pp. 179–186.

[27] A. Goultiaeva, M. Seidl, and A. Biere, "Bridging the gap between dual propagation and cnf-based qbf solving," in *DATE*, 2013, pp. 811–814.

[28] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003, pp. 502–518.

[29] J. Marques-Silva, "On computing minimum size prime implicants," in *IWLS*, 1997.

[30] V. M. Manquinho, P. F. Flores, J. Marques-Silva, and A. L. Oliveira, "Prime implicant computation using satisfiability algorithms," in *ICTAI*, 1997, pp. 232–239.

[31] L. Palopoli, F. Pirri, and C. Pizzuti, "Algorithms for selective enumeration of prime implicants," *AIJ*, vol. 111, no. 1-2, pp. 41–72, 1999.

[32] R. Schrag, "Compilation for critically constrained knowledge bases," in *AAAI*, 1996, pp. 510–515.

[33] T. Castell, "Computation of prime implicates and prime implicants by a variant of the davis and putnam procedure," in *ICTAI*, 1996, pp. 428–429.

[34] D. Déharbe, P. Fontaine, D. L. Berre, and B. Mazure, "Computing prime implicants," in *FMCAD*, 2013, pp. 46–52.

[35] T.-H. Ngair, "A new algorithm for incremental prime implicate generation," in *IJCAI*, 1993, pp. 46–51.

[36] A. Ramesh, G. Becker, and N. V. Murray, "Cnf and dnf considered harmful for computing prime implicants/implicates," *J. Autom. Reasoning*, vol. 18, no. 3, pp. 337–356, 1997.

[37] R. Gershman, M. Koifman, and O. Strichman, "Deriving small unsatisfiable cores with dominators," in *CAV*, 2006, pp. 109–122.

[38] C. Mencía, O. Kullmann, A. Ignatiev, and J. Marques-Silva, "On computing the union of muses," in *SAT*, 2019, pp. 211–221.

[39] T. Toda and T. Soh, "Implementing efficient all solutions sat solvers," *JEA*, vol. 21, pp. 1–12, 2016.