CoAPI: An Efficient Two-Phase Algorithm Using Core-Guided Over-Approximate Cover for Prime Compilation of Non-Clausal Formulae

ID: 233

Abstract

Prime compilation, *i.e.*, the generation of all prime implicates or implicants (*primes* for short) of formulae, is a prominent fundamental issue for AI. Recently, the prime compilation for *non-clausal* formulae has received great attention. The state-of-the-art approaches generate all primes along with a *prime* cover constructed by prime implicates using dual rail encoding. However, the dual rail encoding potentially expands search space. In addition, constructing a prime cover, which is necessary for their methods, is time-consuming. To address these issues, we propose a novel *two-phase* method – CoAPI. The two phases are the key to construct a cover without using dual rail encoding. Specifically, given a non-clausal formula, we first propose a core-guided method to rewrite the non-clausal formula into a cover constructed by *over-approximate* implicates in the first phase. Then, we generate all the primes based on the cover in the second phase. In order to reduce the size of the cover, we provide a *multi-order* based shrinking method, with a good tradeoff between the small size and efficiency, to compress the size of cover considerably. The experimental results show that CoAPI outperforms state-of-the-art approaches. Particularly, for generating all prime implicates, CoAPI consumes about one order of magnitude less time.

1 Introduction

Prime compilation is a prominent fundamental issue for AI. Given a non-clausal Boolean formula, prime compilation aims to generate all the primes of the formula. A prime does not contain redundant literals so that it can represent refined information. Because of that, this problem has widely applications, including logic minimization [Ignatiev *et al.*, 2015], multi-agent systems [Slavkovik and Agotnes, 2014], fault tree analysis [Luo and Wei, 2017], model checking [Bradley and Manna, 2007], bioinformatics [Acuna *et al.*, 2012], *etc.*

This problem is computationally hard. For a non-clausal Boolean formula, the number of primes may be exponential in the size of the formula, while finding one prime is hard for the second level of the PH. In practice, most problems can be hardly expressed in clausal formulae [Stuckey, 2013]. Hence, non-clausal formulae are often transformed into CNF by some encoding methods, such as *Tseitin encoding* [Tseitin, 1968], which reduce the complexity by adding auxiliary variables. Most of the earlier works only generate all primes of a CNF, but they cannot directly compute all primes of a non-clausal formula. Therefore, this issue for non-clausal formula has received great attention [Previti *et al.*, 2015].

The state-of-the-art approaches [Previti *et al.*, 2015] are capable of generating all primes of a non-clausal formula through several iterations. They use *dual rail encoding* [Bryant *et al.*, 1987; Roorda and Claessen, 2005] to encode search space in all produce. Either a prime implicate or a prime implicant is computed at each iteration until a *prime* cover and all primes are obtained. The cover is logically equivalent to the non-clausal formula, which guarantees that all the primes can be obtained. Particularly, they extract a prime from an assignment based on the asymptotically optimal QuickXplain algorithm [Junker, 2004; Bradley and Manna, 2007].

There are three issues in their methods. (i) The dual rail encoding with twice the number of variables than original encoding results in larger search space. (ii) It is a time-consuming task to construct a prime cover because it should completely remove all redundant literals. (iii) It requires a minimal or maximal assignment in order to ensure the correctness, which often exerts a negative influence on SAT solving. These issues probably explain that their performance on the inherent intractability of computing cases is still not satisfactory. Notably, it is questionable whether finding a prime cover has a practical value since the influence of the size of the cover on the other parts of the algorithm is only vaguely known although the prime cover can be smaller.

We propose a novel two-phase method – CoAPI that focuses on an over-approximate cover with a good tradeoff between the small size of the cover and efficiency to improve the performance. We stay within the idea of the work [Previti *et al.*, 2015] that generates all primes based on a cover. However, we use two separate phases to avoid using dual rail encoding in all phases. We construct a cover without dual rail encoding in the first phase. In the second phase, we generate all primes with that. Furthermore, we introduce the notion of the *over-approximate implicate (AIP for short)* that is an implicate containing as few literals as possible. We consider

constructing a cover with a set of AIPs – *over-approximate cover (AC for short)*, rather than with a prime cover. Note that an AC is also logically equivalent to the non-clausal formula.

There are two challenges in our work. The first one is the efficient computation of AIP. Motivated by the applications of the *unsatisfiable core* in optimizing large-scale search problems [Narodytska and Bacchus, 2014; Yamada *et al.*, 2016], we propose a core-guided method to produce AIPs. A smaller unsatisfiable core containing fewer literals should be efficiently obtained, which helps to reduce the number of AIPs in the cover. It is the second challenge, *i.e.*, producing smaller unsatisfiable cores. We notice that the SAT solvers based on the two literal watching scheme [Moskewicz *et al.*, 2001] cannot produce the smallest unsatisfiable core because of the limitation of the partial watching. As for this, we provide a *multi-order* based shrinking method, in which we defined different decision orders to guide the shrinking of unsatisfiable cores in an iterative framework.

We evaluate `CoAPI` on four benchmarks introduced by Previti *et al.*. The experimental results show that `CoAPI` exhibits better performance than state-of-the-art methods. Especially for generating all prime implicates, `CoAPI` is faster about one order of magnitude.

The paper is organized as follows. Section 2 first introduces the basic concepts. Then, Section 3 and Section 4 present the main features of `CoAPI` in detail. After that, Section 5 reports the experiments. Finally, Section 6 discusses related works and Section 7 concludes this paper.

Due to space limit, omitted proofs and supporting materials are provided in the additional file and online appendix (http://tinyurl.com/IJCAI19-233).

2 Preliminaries

This section introduces the notations and backgrounds. The symbols in this section are standard in all.

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. Given a Boolean formula φ, a model is an assignment satisfying φ. Particularly, a model is said to be *minimal* (*resp. maximal*), when it contains the minimal (*resp.* maximal) number of variables assigned true. A *clause* is the disjunction of literals, which is also represented by the set of its literals. The size of a clause is the number of literals in itself. A Boolean formula is in conjunctive normal form (CNF) if it is formed as a conjunction of clauses, which denotes a set of clauses. For a Boolean formula Σ_φ in CNF, $|\Sigma_\varphi|$ means the sum of the size of clauses in Σ_φ. Two Boolean formulae are *logically equivalent* iff they are satisfied by the same models.

Definition 1. *A clause I_e is called as an* implicate *of φ if $\varphi \models I_e$. Especially, I_e is called as* prime *if any clause I'_e s.t. $I'_e \models I_e$ is not an implicate of φ.*

Definition 2. *A term I_n is called as an* implicant *of φ if $I_n \models \varphi$. Especially, I_n is called as* prime *if any term I'_n s.t. $I_n \models I'_n$ is not an implicant of φ.*

The prime compilation aims to compute all the prime implicates or implicants, respectively, denoted by PI^a_e and PI^a_n.

Given a Boolean formula φ, if φ is unsatisfiable, an SAT solver based on CDCL, such as MiniSAT [Eén and Sörensson, 2003], can produce a *proof of unsatisfiability* [McMillan and Amla, 2003; Zhang and Malik, 2003b] using the resolution rule.

Definition 3. *A* proof of unsatisfiability Π *for a set of clauses Σ_φ is a directed acyclic graph (V_Π, E_Π), where V_Π is a set of clauses. For every vertex $c \in V_\Pi$, if $c \in \Sigma_\varphi$, then c is a* root*; otherwise c has exactly two predecessors, c_l and c_r, such that c is the resolvent of c_l and c_r. The empty clause, denoted by $\square$, is the unique* leaf.

Definition 4. *Given a proof of unsatisfiability $\Pi = (V_\Pi, E_\Pi)$, for every clause $c \in V_\Pi$, the* fan-in cone *of c includes of all the $c' \in V_\Pi$ from which there is at least one path to c.*

A proof of unsatisfiability can answer what clauses are in the transitive fan-in cone of the empty clause. Therefore, an unsatisfiable core can be generated through backward traversing from $\square$.

Definition 5. *Given a Boolean formula Σ_φ in CNF, an* unsatisfiable core *$\mathcal{C}$ is a subset of Σ_φ and $\mathcal{C}$ is inconsistent.*

The SAT solver, such as MiniSAT, is capable of handling assumptions. When the solver derives unsatisfiability based on the assumptions for a Boolean formula, it can return *failed assumptions*, which is a subset of assumptions inconsistent with the formula. From here on, we use the terms *failed assumptions* and *unsatisfiable core* interchangeably since every unsatisfiable core corresponds to definite failed assumptions.

3 New Approach – `CoAPI`

In this section, we first introduce the overview of `CoAPI`, then show the details.

3.1 Overview

Given a Boolean formula φ, we first, based on original encoding, construct a cover in CNF to rewrite φ in the first phase, *i.e.*, the cover is logically equivalent to φ; then generate all primes in the second phase based on dual rail encoding. Note that the two-phase produce not only avoids using dual rail encoding in all phases but also exploits the powerful heuristic branching method for SAT solving. For simplicity, this paper only introduces the generation of all prime implicants of φ (similarly for prime implicates because of the duality).

We extend the concepts of the prime implicate and prime cover into the AIP and AC, respectively, which are essential concepts in our algorithm and are defined as follows.

Definition 6. *An* over-approximate implicate *is a clause τ s.t. $\varphi \models \tau$. Given two over-approximate implicates α and β of φ, if $\alpha \models \beta$, then we call α is* smaller *than β.*

The concept of AIP is different from the concept of implicate because the former is as small as possible. Notably, the prime implicate is a minimal AIP.

Definition 7. *An* over-approximate cover *Ω of φ is a conjunction of over-approximate implicates of φ and Ω is logically equivalent to φ. The* cost *of an over-approximate cover Ω, denoted by* $\text{COST}(\Omega) = |\Omega|/|\text{PRIME}(\varphi)|$.

Intuitively, $\text{COST}(\Omega)$ measures the degree of approximation of Ω to $\text{PRIME}(\varphi)$[1].

[1] $\text{PRIME}(\varphi)$ returns a prime cover of φ.

Figure 1: The framework of CoAPI.

The framework of CoAPI includes two phases, namely COMPILECOVER and COMPILEALL, which is shown in Figure 1. It takes a non-clausal Boolean formula φ and its negation $\neg\varphi$ as inputs. The inputs are encoded as a set of clauses by Tseitin encoding or other methods. Its output is all primes of φ. COMPILECOVER first produces an AC of φ and then COMPILEALL computes all primes. We introduce the two phases in detail as follows.

3.2 Core-Guided Over-Approximate Cover

In order to construct a cover, the work [Previti *et al.*, 2015] produces several prime implicates based on the QuickXplain algorithm. A naive approach to extract a prime from an implicate, namely linear approach, is to linearly query whether it is still an implicate after flipping each literal of the implicate. Therefore, the QuickXplain algorithm, based on recursively splitting the implicate, requires exponentially fewer queries in the optimal case than the linear approach. However, it is still time-consuming for producing a prime implicate because there are considerable SAT queries to guarantee the prime.

In addition, the influence of the size of the cover on the other phases is only vaguely known. Hence, although more computation time can lead to a smaller cover, it is not clear whether it is cost-effective in the overall algorithm. The results of the Experiment 5.1 demonstrate this view. Based on the above considerations, we propose a core-guided method to produce AIPs to rewrite φ. It is possible to trade off the quality of the cover with the run time for extraction.

ALGORITHM 1: COMPILECOVER

Input : A formula Σ_φ and its negation $\Sigma_{\neg\varphi}$ in CNF
Output : An AC PI_e^c

1 $\mathcal{R} \leftarrow \Sigma_{\neg\varphi}$; $PI_e^c \leftarrow \emptyset$
2 **while** true **do**
3 $\quad (st, \pi) \leftarrow$ SAT($\mathcal{R}$)
4 $\quad$ **if** *st is* UNSAT **then**
5 $\quad\quad$ **return** PI_e^c
6 $\quad$ **else**
7 $\quad\quad \pi_p \leftarrow$ OVERAPPROXIMATE(Σ_φ, π)
8 $\quad\quad PI_e^c \leftarrow PI_e^c \cup \{\neg\pi_p\}$
9 $\quad\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{\neg\pi_p\}$

We construct a cover to rewrite φ by iteratively computing AIPs in COMPILECOVER shown in Algorithm 1. To this end, COMPILECOVER maintains a set of clauses $\mathcal{R} = \Sigma_{\neg\varphi} \cup PI_e^c$, where $\Sigma_{\neg\varphi}$ encodes $\neg\varphi$ in CNF (Σ_φ for φ) and PI_e^c blocks already computed models.

We illustrate each iteration as follows. COMPILECOVER first searches for a model π of $\neg\varphi$ which is not blocked by PI_e^c (Line 3). Then, OVERAPPROXIMATE is invoked to shrink the unsatisfiable core of π and φ (Line 7). The more detail will be introduced in Section 4. After shrinking, COMPILECOVER updates PI_e^c by adding $\neg\pi_p$ (Line 8). Clearly, $\neg\pi_p$ is a smaller AIP of φ than $\neg\pi$, since $\varphi \models \neg\pi_p$ and $\neg\pi_p \models \neg\pi$. In the end, the updated $\mathcal{R}$ prunes the search space for the next iteration (Line 9).

During the iterations, on the one hand, COMPILECOVER applies an incremental SAT solver to continually shrink the search space by conflict clauses. On the other hand, it also uses PI_e^c to block the space that has been found. Eventually, PI_e^c prunes all the search space of $\neg\varphi$, *i.e.*, an AC of φ has been constructed by PI_e^c. At this point, $\mathcal{R}$ is unsatisfiable and the algorithm terminates. We summarize an example for Algorithm 1 as follows.

Example 1. *Given a formula $\varphi = (a \wedge b) \vee (\neg a \wedge c)$, in the first iteration, a model $\neg a \wedge b \wedge \neg c$ of $\neg\varphi$ is found; then, by consecutive SAT queries, we get a core $\neg a \wedge \neg c$; finally, an AIP $a \vee c$ is produced, clearly, $\varphi \models a \vee c$. During the same step, we can obtain a now AIP $\neg a \vee b$. In total, $\mathcal{R}$ is unsatisfiable, where* COMPILECOVER *produces an AC $(a \vee c) \wedge (\neg a \vee b)$.*

For this example, Previti *et al.* constructs the same result as us. However, CoAPI needs fewer SAT queries while their methods need more queries according to the size of implicate. In general, CoAPI reduces the number of SAT queries to speed up each iteration although it may take more iterations.

3.3 Generation of All Primes

In COMPILEALL, we encodes PI_e^c by dual rail encoding to initialize $\mathcal{H}$. Then, based on SAT solving, we iteratively compute all the minimal models of $\mathcal{H}$, *i.e.*, all the prime implicants of φ. This process is similar to [Jabbour *et al.*, 2014]. The more details about COMPILEALL show in the additional file.

4 Multi-Order based Shrinking

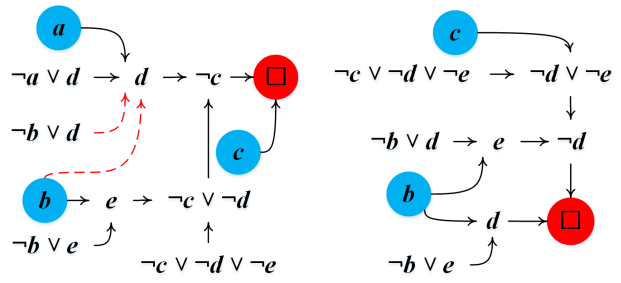
Constructing an AC of φ can be carried out iteratively to produce unsatisfiable cores. Unfortunately, the SAT solver based on deterministic branching strategy often produces similar unsatisfiable cores for similar assumptions. In the worst case, the unsatisfiable core is the same size as the assumptions. Therefore, it is worthwhile finding smaller unsatisfiable cores to compress the size of AC.

Given a proof of unsatisfiability Π, an unsatisfiable core can be produced by traversing Π backward. Therefore, the generation of Π determines the size of the unsatisfiable core. We notice that the SAT solver based on the two literal watching scheme, which is powerful for SAT solving, selectively ignores some information during generating Π. We call this case *blocker ignoring* defined as follows.

Definition 8. *Given a Boolean formula Σ_φ in CNF and its a proof of unsatisfiability $\Pi = (V_\Pi, E_\Pi)$, the clause $\beta \in V_\Pi$ is a* bolcker, *if $|\beta| = 1$ and β is not a root. An SAT solver* ignore *the satisfiability of the clauses containing β.*

Theorem 1. *If a clause β is a blocker of $\Pi = (V_\Pi, E_\Pi)$, then there does not exist a clauses $c \in V_\Pi$ s.t. $\beta \models c$ unless c is in the fan-in cone of β.*

Intuitively, if a blocker β is generated, *i.e.*, the literal β is satisfied, then all clauses containing β are naturally satisfied that can be ignored until backtracking to result in the freedom of β in SAT solving. Therefore, these clauses do not appear in the Π except the fan-in cone of β. This is powerful



(a) (b)

Figure 2: An example for the proof of unsatisfiability.

to search for a model because only the satisfiability of the necessary clauses needs to be considered, which is the core of the two literal watching scheme. However, the blocker ignoring can miss important information for producing a small unsatisfiable core. We use an example to explain this point.

Example 2. *Given $\Sigma_\varphi = (\neg a \vee d) \wedge (\neg b \vee d) \wedge (\neg b \vee e) \wedge (\neg c \vee \neg d \vee \neg e)$, the proof of unsatisfiability is shown in Figure 2(a) based on the assumptions $a \wedge b \wedge c$. We can notice that the $\neg b \vee d$ is missing due to blocker d, in which d is not the resolvent of b and $\neg b \vee d$, but a and $\neg a \vee d$. This proof can produce the unsatisfiable care $a \wedge b \wedge c$.*

Disturbing decision order iteratively in the SAT solving is a useful and straightforward method to guide the smaller unsatisfiable core. A similar approach was proposed by [Zhang and Malik, 2003a]. They iteratively invoke an SAT solver based on a random decision strategy to shrink an unsatisfiable core. However, it lacks power for the prime compilation, which can be shown by the results of the Experiment 5.2. We propose a multi-order decision strategy instead of the random to iteratively shrink a small unsatisfiable core. The multi-order decision strategy includes three kinds of decision orders defined as follows.

Definition 9. *A* decision order *is a list of variables $\Delta = \langle ..., x^i, ..., x^j, ... \rangle$, in which $i < j$ and x^i is picked earlier then x^j by an SAT solver.*

Definition 10. *Given an original decision order Δ, the* forward *decision order Δ_f is the same as Δ. The* interval *decision order has two parts Δ_l and Δ_r with the following properties: (i) if x^i is in Δ_l, then x^{i+1} is in Δ_r; (ii) $\forall x^i, x^j$ in Δ_l (resp. Δ_r), if x^i, x^j in Δ s.t. $i < j$, then x^i is also picked earlier than x^j in Δ_l (resp. Δ_r). The* backward *decision order Δ_b is a reverse of Δ.*

Our method allows an SAT solver to have the opportunity to produce a smaller unsatisfiable core from different definite orders. Given a Boolean formula Σ_φ in CNF and its three variables α, β, and γ, assume that an SAT solver can produce an unsatisfiable core of Σ_φ based on $\langle \alpha, \beta \rangle$ while $\langle \alpha, \gamma \rangle$ or $\langle \gamma, \alpha \rangle$ can result in blockers to enlarge the size of the core. Intuitively, for an original decision order $\langle \alpha, \beta, \gamma \rangle$ (*resp.* $\langle \gamma, \alpha, \beta \rangle$), based on the forward (*resp.*backward) decision order, `CoAPI` can reduce the impact of blockers. For an original order $\langle \alpha, \gamma, \beta \rangle$, the impact can be lessened based on the interval decision decision order ($\Delta_l = \langle \alpha, \beta \rangle$, $\Delta_r = \langle \gamma \rangle$).

Example 3. *Following the above Example 2, the proof of unsatisfiability in Figure 2(b) is based on Δ_b. In this case, a smaller unsatisfiable core, $a \wedge b$, can be produced.*

We provide a multi-order based shrinking method shown in Algorithm 2, in which OrderSAT invokes an SAT solver with certain decision order. The whole algorithm consists of two phases: *basic* and *iterative*. In the basic phase, we first apply Δ_f (Line 2), and then use Δ_l and Δ_r (Line 4). In the iterative phase, we use Δ_f (Line 7) and Δ_b (Line 9) alternately until the bound of iterations or the fixpoint has been reached. The fixpoint is that the size of the core does not change.

ALGORITHM 2: OverApproximate

Input : A CNF Σ_φ and a model π of $\neg\varphi$
Output : An unsatisfiable core π_p

1 Initialize orders Δ_f, Δ_l, Δ_r, and Δ_b based on the order of the index of variables in π
2 $(st, \pi_p) \leftarrow$ OrderSAT$(\Sigma_\varphi, \pi, \Delta_f)$
3 Update Δ_l and Δ_r based on π_p
4 $\pi_p \leftarrow$ Interval$(\Sigma_\varphi, \pi_p, \Delta_l, \Delta_r)$
5 **while** *The bound of iterations or the fixpoint has not been reached* **do**
6 **if** *the last order is* Δ_b **then**
7 $(st, \pi_p) \leftarrow$ OrderSAT$(\Sigma_\varphi, \pi_p, \Delta_f)$
8 **else**
9 $(st, \pi_p) \leftarrow$ OrderSAT$(\Sigma_\varphi, \pi_p, \Delta_b)$
10 **return** π_p

ALGORITHM 3: Interval

Input : A CNF Σ_φ, a model π of $\neg\varphi$, Δ_l, and Δ_r.
Output : An unsatisfiable core π_p

1 $\pi_p \leftarrow \pi$
2 **if** $|\pi_p|$ *is* 1 **then**
3 **return** π_p
4 **else**
5 $(\pi_l, \pi_r) \leftarrow$ Partition$(\pi_p, \Delta_l, \Delta_r)$
6 $(st, \mathcal{C}) \leftarrow$ OrderSAT$(\Sigma_\varphi, \pi_l, \Delta_l)$
7 **if** *st is* UNSAT **then**
8 Update Δ_l and Δ_r based on $\mathcal{C}$
9 **return** Interval$(\Sigma_\varphi, \mathcal{C}, \Delta_l, \Delta_r)$
10 **else**
11 $(st, \mathcal{C}) \leftarrow$ OrderSAT$(\Sigma_\varphi, \pi_r, \Delta_r)$
12 **if** *st is* UNSAT **then**
13 Update Δ_l and Δ_r based on $\mathcal{C}$
14 **return** Interval$(\Sigma_\varphi, \mathcal{C}, \Delta_l, \Delta_r)$
15 **else**
16 **return** π_p

Based on the interval decision order, we partition the unsatisfiable core to explore better results. Algorithm 3 summarizes Interval that is similar to the QuickXplain algorithm, in which Partition partitions an unsatisfiable core based on Δ_l and Δ_r. Compared with the QuickXplain algorithm, Interval avoids discussing the case where none of π_l and π_r is a model of $\neg\varphi$ to cut down the time consumption (Line 16).

Note that OrderSAT with Δ_l or Δ_r is potentially harder than that with Δ_f or Δ_b. The reasons are as follows. First, OrderSAT with the assumptions π in Interval, in which $\pi \not\models \neg\varphi$ is unknown, is in NPC. Second, based on Δ_f or Δ_b, OrderSAT with the assumptions π, in which $\pi \not\models \neg\varphi$

holds, is in polynomial time. Hence, we only use the interval decision order in the basic phase while apply Δ_f and Δ_b in the two phases.

5 Experimental Results

To evaluate our method, we compared CoAPI and its variants with the state-of-the-art methods – primer-a and primer-b [Previti *et al.*, 2015][2] over four benchmarks, and discussed the effects of different shrinking strategies. In each experiment, we considered two tasks: (i) generating all prime implicates; (ii) generating all prime implicants. We implemented CoAPI utilizing MiniSAT[3] that was also used to implement primer-a and primer-b. The benchmarks are introduced by Previti *et al.*, denoted by *QG6*, *Geffe gen.*, *F+PHP*, and *F+GT*, respectively. The experiments were performed on an Intel Core i5-7400 3 GHz, with 8 GByte of memory and running Ubuntu. For each case, the time limit was set to 3600 seconds and the memory limit to 7 GByte.

5.1 Comparisons between CoAPI and primer

We assess the performance of CoAPI in this section. In this experiment, we implemented the variant of CoAPI, denoted by CoAPI-qx, which uses the QuickXplain algorithm to construct a prime cover in the first phase. We also implemented CoAPI with only one iteration, denoted by CoAPI-1it. We evaluate the performance of CoAPI-qx, CoAPI-1it, primer-a, and primer-b by the 743 cases.

Table 1: The number of solved cases.

	QG6 (83)	*Geffe gen.* (600)	*F+PHP* (30)	*F+GT* (30)	*Total* (743)
primer-a	30 / 66	576 / **596**	30 / 30	28 / 30	664 / 722
primer-b	30 / 65	577 / **596**	30 / 30	28 / 30	665 / 721
CoAPI-qx	30 / 70	**589** / 592	30 / 30	26 / 30	675 / 722
CoAPI-1it	30 / **81**	**589** / 591	30 / 30	**30** / 30	**679** / **732**

Table 1 shows the number of cases that can be computed. The results are separated by the symbol '/', on the right of which is for the task (i) and the left of which is for the task (ii). It is used for all tables. Overall, CoAPI-qx and CoAPI-1it can successfully solve more cases than primer-a and primer-b. Note that the 679 cases solved by CoAPI-1it include all the 664 (*resp.* 665) ones solved by primer-a (*resp.* primer-b) in the task (i). It is obvious that, for *QG6*, CoAPI-qx and CoAPI-1it dramatically increase the number of cases successfully solved in the task (ii).

The more detail comparisons of these methods for the task (i) are shown in Figure 3(a). The X-axis indicates the time in seconds taken by CoAPI-qx or CoAPI-1it, and the Y-axis indicates that taken by primer-a or primer-b. Points above the diagonal indicate advantages for CoAPI-qx or CoAPI-1it. CoAPI-1it generally computes much faster than primer-a (*resp.* primer-b) in 92% (*resp.* 96%) cases – it consumes about at least one order of magnitude less time than primer-a (*resp.* primer-b) in 26% (*resp.* 27%) cases. For CoAPI-qx, the advantage is still obvious. It is in 73% (*resp.* 80%) cases that CoAPI-qx beats primer-a (*resp.* primer-b), in which CoAPI-qx is about one orders of magnitude faster in 18% (*resp.* 19%) cases than primer-a (*resp.* primer-b). In this task, most of the literals in implicate are necessary. Therefore, the QuickXplain algorithm may require significantly more SAT queries than our method.

Figure 3(b) shows the performance of these methods for the task (ii) in detail. Cases that are negative for CoAPI-1it focus on *F+PHP* and *F+GT*, because the prime covers of these formulae are in the form $(x_1 \vee y_1) \wedge ... \wedge (x_m \vee y_m)$ that is extremely beneficial to generate all primes. We focus on the challenging cases that are computed over 1000s by primer-a or primer-b, *i.e.*, the cases are shown in the green area in Figure 3(b). Most of the points above the diagonal (at least 62% cases for CoAPI-qx and 84% for CoAPI-1it) indicate the advantage of our methods. In particular, CoAPI-1it dominates primer-a and primer-b on *QG6* reducing used time for at least 40.26%.

Table 2: The improvement of our methods for challenging cases.

	Win	Win x10+
CoAPI-qx vs. primer-a	70% / 64%	47% / 0%
CoAPI-qx vs. primer-b	70% / 62%	47% / 0%
CoAPI-1it vs. primer-a	**93% / 86%**	**50%** / 0%
CoAPI-1it vs. primer-b	**93% / 84%**	**50%** / 0%

For challenging cases, the improvements of our methods are shown in Table 2, in which the columns present the percentage of faster cases (Win) and the percentage of at least one order of magnitude faster cases (Win x10+). Note that, for CoAPI-1it, the number of faster cases in the task (ii) increases to 86% (*resp.* 84%) and the number of cases with at least one order of magnitude faster improves to 50% (*resp.* 50%) in the task (i).

In general, our methods outperform the state-of-the-art methods, particularly in the task (i). The outstanding performance of CoAPI-qx shows that the two-phases framework is efficient because it avoids using dual rail encoding and the minimal or maximal assignment strategy throughout the whole algorithm. Moreover, we can notice that CoAPI-1it is better than CoAPI-qx because of the AC, which is described in the next section.

5.2 Evaluations of Over-Approximation

To evaluate the different shrinking strategies, we implemented CoAPI-0it without iterations and CoAPI-2it with two iterations. Moreover, CoAPI-zm uses the strategy proposed by [Zhang and Malik, 2003a]. Based on our experiences, CoAPI-zm with 11 iterations gives the best performance for the two tasks in practice.

Table 3: Results of shrinking unsatisfiable cores.

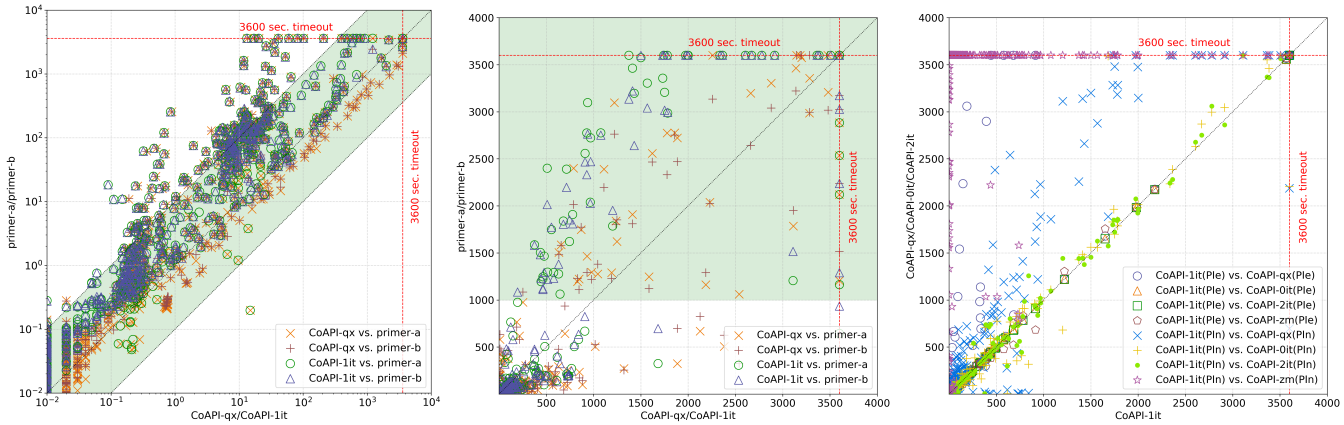
	Cost	Fixpoint	First Shrink	Other Shrink
CoAPI-0it	1.00 / 2.59	– / –	7% / 92%	7% / 93%
CoAPI-1it	1.00 / 1.75	0% / 0%	7% / 92%	7% / 94%
CoAPI-2it	1.00 / 1.72	99% / 74%	7% / 92%	7% / 94%
CoAPI-zm	1.00 / 6854.80	100% / 99%	7% / 65%	7% / 67%

We compare CoAPI and CoAPI-zm in different shrinking strategies on the same benchmarks as above. The results are shown in Figure 3(c). The most points are above the diagonal line, which represents a less used time for CoAPI-1it in most cases. CoAPI-2it and CoAPI-zm are comparable in the task (i). However, in the task (ii), CoAPI-zm only solves 302 of 743 cases that are all simple for CoAPI-1it.

Table 3 shows the statistics on average for shrinking unsatisfiable cores. Due to CoAPI-qx with a prime cover,

[2]https://reason.di.fc.ul.pt/wiki/doku.php?id=-primer.

[3]https://github.com/niklasso/minisat.



(a) Computation of prime implicate. (b) Computation of prime implicant. (c) Shrinking results.

Figure 3: Performance comparison.

we compute the cost of ACs based on the CoAPI-qx. The columns present the cost (Cost), the ratio of reaching the fixpoint (Fixpoint), the ratio of the shrinking size in the first time (First Shrink), and the ratio of the shrinking size in the other times (Other Shrink).

The generally lower costs of CoAPI-0it, CoAPI-1it, and CoAPI-2it show the shrunk unsatisfiable core can be much smaller in all cases. From the statistics, the cost is often reduced by running the shrinking procedure iteratively, but usually, the gains for the shrinking core are not as substantial as the first shrinking. This point is also reflected in the ratio of shrinking size, in which the size of the AIPs reduces dramatically in the first time, but not by much during the following shrinkings. We also note that CoAPI-2it can reach a fixpoint in most cases. These statistics mean that the only one iteration is the best tradeoff for the quality of the unsatisfiable core with the run time for these benchmarks. Comparing CoAPI-2it with CoAPI-zm, the cost and the ratio of the shrinking size in the task (ii) illustrate that CoAPI-zm cannot effectively control shrinking, while CoAPI-2it does well for it.

6 Related Works and Disscusions

Many of techniques to the prime compilation are based on branch and bound/backtrack search procedures [Castell, 1996; Ravi and Somenzi, 2004; Déharbe *et al.*, 2013; Jabbour *et al.*, 2014]. They take full advantage of powerful SAT solvers, while these methods cannot generate the primes for non-clausal formulae. In addition, a number of approaches based on *binary decision diagrams (BDD)* [Coudert and Madre, 1992] or *zero-suppressed BDD (ZBDD)* [Simon, 2001] have been proposed. These methods can encode primes in a compact space thanks to BDD. Given the complexity of the problem, however, these methods may still suffer from time or memory limitations in practice. Almost simultaneously, a *0-1 integer linear programming (ILP)* formulation [Pizzuti, 1996; Manquinho *et al.*, 1997; Marques-Silva, 1997; Palopoli *et al.*, 1999] was proposed to compute primes of CNF formulae. Although these approaches can naturally encode the minimal constraints utilizing ILP, their efficiency is questionable.

Most present works [Castell, 1996; Pizzuti, 1996; Manquinho *et al.*, 1997; Marques-Silva, 1997; Palopoli *et al.*, 1999; Ravi and Somenzi, 2004; Jabbour *et al.*, 2014] only focus on computing primes of CNF or DNF, while there are also some approaches for working on non-clausal formulae. Such as, Ngair [1993] studied a more general algorithm for prime implicate generation, which allows any conjunction of DNF formulae. The approaches based on the BDD/ZBDD can compute prime implicants of non-clausal formulae. Additionally, Ramesh *et al.* [1997] computed prime implicants and prime implicates of NNF formula. Recently, Previti *et al.* [2015] described the most efficient approach at present.

In order to produce a small AIP, we need to generate small unsatisfiable cores. Zhang and Malik [2003a] produced small unsatisfiable cores by the random order and multiple iterations. This approach is similar to our idea, but they fail to find the relationship between the order and the size of unsatisfiable cores, resulting in their approach without the ability to further shrink unsatisfiable cores. Gershman *et al.* [2006] suggested a more effective shrinking procedure based on dominators resulting from the proof of unsatisfiability. Naturally, analysis based on proof of unsatisfiability increases the cost of a single iteration. Hence, considering large-scale iterations for shrinking different unsatisfiable cores in our work, their method does not work well to this.

7 Conclusions and Future Works

We have proposed a novel approach – CoAPI for the prime compilation based on unsatisfiable cores. Compared with the work [Previti *et al.*, 2015], CoAPI separates the generating processes into two phases, which can permit us to construct a cover without using dual rail encoding resulting in shrinking the search space. Moreover, we have proposed a core-guided approach to construct an AC to rewrite the formula. It should emphasize that the AC can be efficiently computed. Besides, we have provided a multi-order based method to shrink a small unsatisfiable core. The experimental results have shown that CoAPI has a significant advantage for the generation of prime implicates and better performance for prime implicants than state-of-the-art methods.

For future work, we expect that our method can be applied to the task of producing a small proof of unsatisfiability.

References

[Acuna *et al.*, 2012] Vicente Acuna, Paulo Vieira Milreu, Ludovic Cottret, Alberto Marchettispaccamela, Leen Stougie, and Mariefrance Sagot. Algorithms and complexity of enumerating minimal precursor sets in genome-wide metabolic networks. *Bioinformatics*, 28(19):2474–2483, 2012.

[Bradley and Manna, 2007] A. R. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMAC*, pages 173–180, 2007.

[Bryant *et al.*, 1987] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. Cosmos: A compiled simulator for mos circuits. In *DAC*, pages 9–16, 1987.

[Castell, 1996] Thierry Castell. Computation of prime implicates and prime implicants by a variant of the davis and putnam procedure. In *ICTAI*, pages 428–429, 1996.

[Coudert and Madre, 1992] Olivier Coudert and Jean Christophe Madre. Implicit and incremental computation of primes and essential primes of boolean functions. In *DAC*, pages 36–39, 1992.

[Déharbe *et al.*, 2013] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *FMCAD*, pages 46–52, 2013.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[Gershman *et al.*, 2006] Roman Gershman, Maya Koifman, and Ofer Strichman. Deriving small unsatisfiable cores with dominators. In *CAV*, pages 109–122, 2006.

[Ignatiev *et al.*, 2015] A. Ignatiev, A. Previti, and J. Marques-Silva. Sat-based formula simplification. In *SAT*, pages 287–298, 2015.

[Jabbour *et al.*, 2014] Said Jabbour, Joao Marques-Silva, Lakhdar Sais, and Yakoub Salhi. Enumerating prime implicants of propositional formulae in conjunctive normal form. In *JELIA*, pages 152–165, 2014.

[Junker, 2004] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–175, 2004.

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

[Manquinho *et al.*, 1997] Vasco M. Manquinho, Paulo F. Flores, Joao Marques-Silva, and Arlindo L. Oliveira. Prime implicant computation using satisfiability algorithms. In *ICTAI*, pages 232–239, 1997.

[Marques-Silva, 1997] Joao Marques-Silva. On computing minimum size prime implicants. In *IWLS*, 1997.

[McMillan and Amla, 2003] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.

[Moskewicz *et al.*, 2001] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, 2001.

[Narodytska and Bacchus, 2014] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *AAAI*, pages 2717–2723, 2014.

[Ngair, 1993] Teow-Hin Ngair. A new algorithm for incremental prime implicate generation. In *IJCAI*, pages 46–51, 1993.

[Palopoli *et al.*, 1999] Luigi Palopoli, Fiora Pirri, and Clara Pizzuti. Algorithms for selective enumeration of prime implicants. *Journal of Artificial Intelligence*, 111(1-2):41–72, 1999.

[Pizzuti, 1996] Clara Pizzuti. Computing prime implicants by integer programming. In *ICTAI*, pages 332–336, 1996.

[Previti *et al.*, 2015] Alessandro Previti, Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. Prime compilation of non-clausal formulae. In *IJCAI*, volume 15, pages 1980–1987, 2015.

[Ramesh *et al.*, 1997] Anavai Ramesh, George Becker, and Neil V. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18(3):337–356, 1997.

[Ravi and Somenzi, 2004] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *TACAS*, pages 31–45, 2004.

[Roorda and Claessen, 2005] Jan-Willem Roorda and Koen Claessen. A new sat-based algorithm for symbolic trajectory evaluation. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 238–253, 2005.

[Simon, 2001] L. Simon. Efficient consequence finding. In *IJCAI*, pages 359–365, 2001.

[Slavkovik and Agotnes, 2014] M. Slavkovik and T. Agotnes. A judgment set similarity measure based on prime implicants. *Adaptive Agents and Multi Agents Systems*, pages 1573–1574, 2014.

[Stuckey, 2013] P. J. Stuckey. There are no cnf problems. In *SAT*, pages 19–21, 2013.

[Tseitin, 1968] G. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, pages 234–259, 1968.

[Yamada *et al.*, 2016] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *ASE*, pages 614–624, 2016.

[Zhang and Malik, 2003a] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *SAT*, 2003.

[Zhang and Malik, 2003b] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 880–885, 2003.