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 trade-off 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 Previ 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 \( \tau \) is a conjunction of literals, represented as a set of literals. \( |\tau| \) is the size of \( \tau \), i.e., the number of literals in \( \tau \). Given a Boolean formula \( \varphi \), a model is an assignment satisfying \( \varphi \). 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 \( \Sigma_\varphi \) in CNF, \( |\Sigma_\varphi| \) means the sum of the size of clauses in \( \Sigma_\varphi \). Two Boolean formulae are *logically equivalent* if they are satisfied by the same models.

**Definition 1.** A clause \( I_n \) is called an implicite of \( \varphi \) if \( \varphi \models I_n \). Especially, \( I_n \) is called as prime if any clause \( I'_n \) s.t. \( I'_n \models I_n \) is not an implicite of \( \varphi \).

**Definition 2.** A term \( I_n \) is called as an implicant of \( \varphi \) 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 \( \varphi \).

The prime compilation aims to compute all the prime implicates or implicants, respectively, denoted by \( \text{PI}_n^\alpha \) and \( \text{PI}_n^\beta \).

Given a Boolean formula \( \varphi \), if \( \varphi \) 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 \( \Pi \) for a set of clauses \( \Sigma_\varphi \) is a directed acyclic graph \( (V_{\Pi}, E_{\Pi}) \), where \( V_{\Pi} \) 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, \( c_1 \) and \( c_r \), such that \( c \) is the resolvent of \( c_1 \) 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 \( \Sigma_\varphi \) in CNF, an unsatisfiable core \( \mathcal{C} \) is a subset of \( \Sigma_\varphi \) 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 \( \varphi \), we first, based on original encoding, construct a cover in CNF to rewrite \( \varphi \) in the first phase, i.e., the cover is logically equivalent to \( \varphi \); 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 implicates of \( \varphi \) (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 \( \tau \) s.t. \( \varphi \models \tau \). Given two over-approximate implicates \( \alpha \) and \( \beta \) of \( \varphi \), if \( \alpha \models \beta \), then we call \( \alpha \) is smaller than \( \beta \).

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

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

Intuitively, \( \text{COST} (\Omega) \) measures the degree of approxima-

\footnote{\text{PRIME} (\varphi) returns a prime cover of \varphi.}
The framework of CoAPI includes two phases, namely COMPILECOVER and COMPILEALL, which is shown in Figure 1. It takes a non-clausal Boolean formula $\varphi$ 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 $\varphi$. COMPILECOVER first produces an AC of $\varphi$ 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 implicants 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 run time for extraction. 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 $\varphi$. It is possible to trade off the quality of the cover with the run time for extraction.

**Algorithm 1:** COMPILECOVER

**Input**: A formula $\Sigma_{\varphi}$ and its negation $\Sigma_{\neg \varphi}$ in CNF

**Output**: An AC $PI^c_{\Sigma}$

1. $R \leftarrow \Sigma_{\neg \varphi} \cup PI^c_{\Sigma} \leftarrow \emptyset$
2. **while** true **do**
3. \hspace{1em} $(s, \pi) \leftarrow \text{SAT}(R)$
4. \hspace{1em} **if** $s$ is UNSAT **then**
5. \hspace{2em} **return** $PI^c_{\Sigma}$
6. \hspace{1em} **else**
7. \hspace{2em} $\pi_p \leftarrow \text{OVERAPPROXIMATE}(\Sigma_{\varphi}, \pi)$
8. \hspace{2em} $PI^c_{\Sigma} \leftarrow PI^c_{\Sigma} \cup \{\neg \pi_p\}$
9. \hspace{2em} $R \leftarrow R \cup \{\neg \pi_p\}$

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

We illustrate each iteration as follows. COMPILECOVER first searches for a model $\pi$ of $\neg \varphi$ which is not blocked by $PI^c_{\Sigma}$ (Line 3). Then, OVERAPPROXIMATE is invoked to shrink the unsatisfiable core of $\pi$ and $\varphi$ (Line 7). The more detail will be introduced in Section 4. After shrinking, COMPILECOVER updates $PI^c_{\Sigma}$ by adding $\neg \pi_p$ (Line 8). Clearly, $\neg \pi_p$ is a smaller AIP of $\varphi$ than $\neg \pi$, since $\varphi \models \neg \pi_p$ and $\neg \pi_p \models \neg \pi$. In the end, the updated $R$ 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 $PI^c_{\Sigma}$ to block the space that has been found. Eventually, $PI^c_{\Sigma}$ prunes all the search space of $\neg \varphi$, i.e., an AC of $\varphi$ has been constructed by $PI^c_{\Sigma}$. At this point, $R$ is unsatisfiable and the algorithm terminates. We summarize an example for Algorithm 1 as follows.

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

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 $PI^c_{\Sigma}$ by dual rail encoding to initialize $H$. Then, based on SAT solving, we iteratively compute all the minimal models of $H$, i.e., all the prime implicants of $\varphi$. 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 $\varphi$ 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 $\Pi$, an unsatisfiable core can be produced by traversing $\Pi$ backward. Therefore, the generation of $\Pi$ 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 $\Pi$. We call this case blocker ignoring defined as follows.

**Definition 8.** Given a Boolean formula $\Sigma_{\varphi}$ in CNF and its a proof of unsatisfiability $\Pi = \{V_{\Pi}, E_{\Pi}\}$, the clause $\beta \in V_{\Pi}$ is a blocker, if $|\beta| = 1$ and $\beta$ is not a root. An SAT solver ignore the satisfiability of the clauses containing $\beta$.

**Theorem 1.** If a clause $\beta$ 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 $\beta$.

Intuitively, if a blocker is generated, i.e., the literal $\beta$ is satisfied, then all clauses containing $\beta$ are naturally satisfied that can be ignored until backtracking to result in the freedom of $\beta$ in SAT solving. Therefore, these clauses do not appear in the $\Pi$ except the fan-in cone of $\beta$. This is powerful
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 \( \Sigma_\varphi = (\neg a \vee d) \land (\neg b \vee d) \land (\neg b \vee e) \land (\neg c \vee \neg d \vee \neg e) \), 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 core \( a \wedge b \wedge c \).

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. A similar approach was proposed by [Zhang a]

for an original decision order \( \Delta \), the proof order can produce the unsatisfiable care \( \Delta_0 \).

Our method allows an SAT solver to have the opportunity to produce a smaller unsatisfiable core from different definite orders. Given a Boolean formula \( \Sigma_\varphi \) in CNF and its three variables \( \alpha, \beta, \) and \( \gamma \), assume that an SAT solver can produce an unsatisfiable core of \( \Sigma_\varphi \) based on \( \langle \alpha, \beta \rangle \) while \( \langle \alpha, \gamma \rangle \) or \( \langle \gamma, \alpha, \beta \rangle \) 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, CoAP1 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 order (\( \Delta_1 = \langle \alpha, \beta \rangle, \Delta_2 = \langle \gamma \rangle \)).

**Example 3.** Following the above Example 2, the proof of unsatisfiability in Figure 2(b) is based on \( \Delta_b \). 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 \( \Delta_f \) (Line 2), and then use \( \Delta_l \) and \( \Delta_b \) (Line 4). In the iterative phase, we use \( \Delta_f \) (Line 7) and \( \Delta_l \) (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 \( \Sigma_\varphi \) and a model \( \pi \) of \( \neg \varphi \)
Output : An unsatisfiable core \( \pi_p \)
1 Initialize orders \( \Delta_f, \Delta_l, \Delta_r \), and \( \Delta_b \) based on the order of the index of variables in \( \pi \)
2 \((st, \pi_p) \leftarrow ORDERSAT(\Sigma_\varphi, \pi, \Delta_f )\)
3 Update \( \Delta_l \) and \( \Delta_r \) based on \( \pi_p \)
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 \( \Delta_l \), 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 \( \pi_p \)
```

**ALGORITHM 3:** INTERVAL

```
Input : A CNF \( \Sigma_\varphi \), a model \( \pi \) of \( \neg \varphi \), \( \Delta_l \), and \( \Delta_r \).
Output : An unsatisfiable core \( \pi_p \)
1 \( \pi_p \leftarrow \pi \)
2 if \( |\pi_p| \) is 1 then
3 return \( \pi_p \)
4 else
5 \((\pi_l, \pi_r) \leftarrow PARTITION(\pi_p, \Delta_l, \Delta_r )\)
6 \((st, C) \leftarrow ORDERSAT(\Sigma_\varphi, \pi_l, \Delta_l )\)
7 if st is UNSAT then
8 Update \( \Delta_l \) and \( \Delta_r \) based on \( C \)
9 return INTERVAL(\(\Sigma_\varphi, C, \Delta_l, \Delta_r \))
10 else
11 \((st, C) \leftarrow ORDERSAT(\Sigma_\varphi, \pi_r, \Delta_r )\)
12 if st is UNSAT then
13 Update \( \Delta_l \) and \( \Delta_r \) based on \( C \)
14 return INTERVAL(\(\Sigma_\varphi, C, \Delta_l, \Delta_r \))
15 else
16 return \( \pi_p \)
```

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 \( \Delta_l \) and \( \Delta_r \). Compared with the QuickXplain algorithm, INTERVAL avoids discussing the case where none of \( \pi_l \) and \( \pi_r \) is a model of \( \neg \varphi \) to cut down the time consumption (Line 16).

Note that ORDERSAT with \( \Delta_l \) or \( \Delta_r \) is potentially harder than that with \( \Delta_f \) or \( \Delta_b \). The reasons are as follows. First, ORDERSAT with the assumptions \( \pi \) in INTERVAL, in which \( \pi \models \neg \varphi \) is unknown, is in NPC. Second, based on \( \Delta_f \) or \( \Delta_b \), ORDERSAT with the assumptions \( \pi \), in which \( \pi \models \neg \varphi \).
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] 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 implicates. 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-lit. We evaluate the performance of CoAPI-qx, CoAPI-lit, primer-a, and primer-b by the 743 cases.

Table 1: The number of solved cases.

<table>
<thead>
<tr>
<th></th>
<th>QG6</th>
<th>Geffe gen.</th>
<th>F+PHP</th>
<th>F+GT</th>
<th>Total</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(83)</td>
<td>(600)</td>
<td>(30)</td>
<td>(30)</td>
<td>(743)</td>
</tr>
<tr>
<td>primer-a</td>
<td>30/66</td>
<td>57/7596</td>
<td>30/30</td>
<td>28/30</td>
<td>664/722</td>
<td></td>
</tr>
<tr>
<td>primer-b</td>
<td>30/65</td>
<td>57/7596</td>
<td>30/30</td>
<td>28/30</td>
<td>665/722</td>
<td></td>
</tr>
<tr>
<td>CoAPI-qx</td>
<td>30/30</td>
<td>589/591</td>
<td>30/30</td>
<td>26/30</td>
<td>675/722</td>
<td></td>
</tr>
<tr>
<td>CoAPI-lit</td>
<td>30/81</td>
<td>589/591</td>
<td>30/30</td>
<td>30/30</td>
<td>679/722</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 shows the number of cases that can be computed. The results are separated by the symbol ‘\(|\rangle\)’, 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-lit can successfully solve more cases than primer-a and primer-b. Note that the 679 cases solved by CoAPI-lit 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-lit dramatically increase the number of cases successfully solved in the task (ii).

The more detailed 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-lit, and the Y-axis indicates that taken by primer-a or primer-b. Points above the diagonal indicate advantages for CoAPI-qx or CoAPI-lit. CoAPI-lit 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 implicates 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-lit focus on F+PHP and F+GT, because the prime covers of these formulae are in the form \((x_1 \lor y_1) \land \ldots \land (x_m \lor y_m)\) 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-lit) indicate the advantage of our methods. In particular, CoAPI-lit 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Prime Cover</th>
<th>Win</th>
<th>Win x10+</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoAPI-qx vs. primer-a</td>
<td>70% / 84%</td>
<td>47% / 70%</td>
<td></td>
</tr>
<tr>
<td>CoAPI-qx vs. primer-b</td>
<td>70% / 82%</td>
<td>47% / 70%</td>
<td></td>
</tr>
<tr>
<td>CoAPI-lit vs. primer-a</td>
<td>93% / 86%</td>
<td>50% / 70%</td>
<td></td>
</tr>
<tr>
<td>CoAPI-lit vs. primer-b</td>
<td>93% / 84%</td>
<td>50% / 70%</td>
<td></td>
</tr>
</tbody>
</table>

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-lit, 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-lit 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cost</th>
<th>Fixpoint</th>
<th>First Shrink</th>
<th>Other Shrink</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoAPI-0it</td>
<td>1.00 / 2.59</td>
<td>-/-</td>
<td>7% / 92%</td>
<td>7% / 93%</td>
</tr>
<tr>
<td>CoAPI-1it</td>
<td>1.00 / 1.75</td>
<td>0% / 70%</td>
<td>7% / 92%</td>
<td>7% / 94%</td>
</tr>
<tr>
<td>CoAPI-2it</td>
<td>1.00 / 1.75</td>
<td>99% / 70%</td>
<td>7% / 92%</td>
<td>7% / 94%</td>
</tr>
<tr>
<td>CoAPI-zm</td>
<td>1.00 / 6834.80</td>
<td>100% / 70%</td>
<td>7% / 95%</td>
<td>7% / 97%</td>
</tr>
</tbody>
</table>

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-lit 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-lit.

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

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; 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


