Accelerate Safety Model Checking Based on Complementary Approximate Reachability

Xiaoyu Zhang, Shengping Xiao, Yechuan Xia, Jianwen Li, Mingsong Chen, and Geguang Pu

***Abstract*—Model checking is an automatic formal verification method that is widely applied to hardware verification. Safety properties are the mainly verified properties in practice that can be falsified within finite steps if they do not hold for systems. However, state-of-the-art safety model checking algorithms cannot meet the performance requirement driven by the industry as the sizes of (hardware) systems to be verified increase rapidly. Therefore, more efficient techniques are still eagerly in demand. Recently, a new safety model checking technique Complementary Approximate Reachability (CAR) was presented and received considerable concerns from the community. CAR has shown its advantages in unsafe checking (bug finding), but cannot be as competitive as other state-of-the-art techniques, e.g., IC3/PDR, on safe checking (proving correctness). In this paper, we propose four kinds of heuristics, two inspired by IC3/PDR and another two dedicated to CAR, to improve the performance of CAR. We integrate the heuristics into the open-source model checker SimpleCAR and compare the performance to the original CAR and IC3/PDR on 748 instances from the hardware model-checking competitions. Our results show that by fixing the time and memory resources, CAR can solve 124 more instances with the four proposed heuristics, i.e., 53.4% more instances can be solved comparing to the original CAR. Furthermore, CAR in both forward and backward directions can solve 10 more instances than IC3/PDR in corresponding directions, and uniquely solve 44 more instances that IC3/PDR in corresponding directions cannot solve, which increases the capability of the current model-checking portfolio.**

I. Introduction

Formal verification techniques, such as model checking, are becoming more and more attractive to the hardware design community [1], [2]. As a complement to the traditional simulation technique, model checking can not only detect additional bugs but also prove the correctness of the hardware system. Given a model M and a property P, model checking searches all possible behaviors of M to check whether P holds for M. Once a system behavior is detected to violate the property P, the model checker returns a *counterexample* as an evidence, which demonstrates the execution of the system leading to the property violation. Such a process is called *bug finding*. We focus on the topic of safety model checking, in which P is a safety property such that the violation of the property can be detected within finite steps. It is well known that the safety model checking problem can be reduced to that of reachability analysis [3].

The authors Xiaoyu Zhang, Shengping Xiao, Yechuan Xia, Jianwen Li, Mingsong Chen, and Geguang Pu are with the Software Engineering Institute, East China Normal University, Shanghai, 200062, China. Geguang Pu is also with the Shanghai Trusted Industrial Control Platform Co., Ltd, Shanghai, 200333, China. Jianwen Li and Geguang Pu are the corresponding authors (email address: lijwen2748@gmail.com, ggpu@sei.ecnu.edu.cn).

State-of-the-art safety model checking techniques include Bounded Model Checking (BMC) [4], [5], Interpolation Model Checking (IMC) [6], Property Directed Reachability (IC3/PDR) [7], [8], and Complementary Approximate Reachability (CAR) [9], all of which integrate the Boolean Satisfiability (SAT) technique to boost the performance. Notably, there is so far no such algorithm that can dominate others, though from the experience BMC is good at unsafe checking (bug finding), IMC and IC3/PDR are better for safe checking (correctness proof), and CAR complements both BMC on bug finding and IMC/IC3/PDR on proving correctness by solving a considerable number of industrial instances that cannot be solved by others within a given time and hardware resources [9], [10]. Therefore, a portfolio consisting of different techniques is maintained for different verification tasks.

However, based on the discussion with our industrial partners, the aforementioned model checking techniques still cannot meet the performance requirement when verifying large designs. That is, the sizes of models are too large to be checked by extant techniques within the considerable time and memory resources. An alternative to ease the urgent verification overhead is to use some divide-and-conquer strategies like *assume-guarantee* [11], which decompose the large model into small ones that can be verified by state-of-the-art model checkers, and proving the correctness of such small parts can induce the correctness of the original model. Yet, even the models from decomposition are becoming difficult to be verified, considering the current techniques' performance. As a result, there is still a dearth of more efficient model checking techniques for verifying large hardware designs.

Complementary Approximate Reachability (CAR) is a new technique of SAT-based model checking, inspired by the traditional reachability analysis [9]. Analogous to the reachability analysis, CAR can be performed in both forward and backward directions, which are named Forward and Backward CAR respectively. CAR maintains an over-approximate *abstract* state-sets sequence (i.e., the O-sequence) for safe checking, and an under-approximate *explicit* state-sets sequence (i.e., the U-sequence) for unsafe checking. Generally speaking, CAR frequently invokes SAT calls, from which the satisfiable result, namely the *assignment*, is used to update the U-sequence while the unsatisfiable result, namely the *Unsatisfiable Core* (UC), is used to refine the O-sequence. CAR has shown superiority on unsafe checking to IC3/PDR [10], [12], but still cannot perform as well as IC3/PDR [9] on safe checking.

From our understanding, the performance difference between the two techniques is caused by the different purposes they were designated to. On one hand, IC3/PDR focuses on proving correctness, so it costs a lot of effort to compute the

so-called *minimal inductive cores* (MIC), from which the proof certificate is shown to be conducted more efficiently. However, the drawback of bug-finding performance may have to be paid for. On the other hand, CAR operates more like a search algorithm, which tries to detect a counterexample as soon as possible. During the process, the collected constraints are used mainly for the guidance of state search. As a result, CAR has a flexible framework to use different search strategies to find bugs, while it may lose performance when proving correctness.

This paper proposes four kinds of different heuristics to enhance CAR's performance in proving correctness. The first two are inspired by IC3/PDR, namely *UC-propagation* and *Partial-state generation*. The idea of UC-propagation is to propagate the elements in O_i to O_{i+1} such that the proof certificate can be located more quickly. The Partial-state generation aims to extract a set of states from a single one such that more states can be added into the U-sequence to accelerate the checking. Compared to that implemented in IC3/PDR, we propose more aggressive strategies for these two heuristics that additionally make use of the state information in the O- and U-sequences. The other two heuristics are achieved by *MUC-extraction* and *Dead-state detection*, which are specific in CAR and not used in IC3/PDR. MUC-extraction is presented to compute the Minimal Unsatisfiable Core (MUC) instead of the Unsatisfiable Core (UC) to refine the O-sequence, which can be more efficient. A dead state is a state which does not have any predecessors. The Dead-state detection heuristic is to detect such dead states and block them forever to prune the state space significantly.

To evaluate the efficiency of these proposed heuristics, an extensive evaluation on a 748-instances benchmark from the hardware model checking competition (HWMCC) 2015 [13] and 2017 [14] was performed in our experiments. By fixing the time and memory resources, CAR can solve 124 more instances with the four proposed heuristics, i.e., 53.4% more instances can be solved comparing to the original CAR. Furthermore, CAR in both directions (forward and backward) can solve 10 more instances than IC3/PDR in both directions, and uniquely solves 44 more instances that IC3/PDR in corresponding directions cannot solve, which increases the ability of the current model-checking portfolio.

Notably, by integrating the four heuristics, neither single direction of CAR is able to perform better than the default (Forward) IC3/PDR: This indicates that computing MIC is still a better generalization technique than simply computing MUC ones. However, our experiments show that the two directions of CAR can complement each other in a better way than that of IC3/PDR. Therefore, the shortcoming of CAR in proving correctness can be made up by conducting a portfolio including both directions of CAR, without losing its advantage on bug-finding.

In summary, the contributions of this paper are as follows:

- We propose four different heuristics to enhance the performance of CAR, making CAR a state-of-the-art model checking technique as competitive as IC3/PDR.
- We conduct an extensive experimental evaluation to show the efficiency of our proposed heuristics. We show that even though Forward/Backward CAR cannot outperform the mainstream Forward IC3/PDR, combining both directions together enables CAR to outperform IC3/PDR, which provides a (potential) new direction to improve state-of-the-art model-checking techniques.

We continue in the next section with related work. Section III introduces the preliminaries. Section IV demonstrates a motivating example. Section V presents our approaches including the heuristics. Section VI shows the experimental results. Finally, Section VII concludes the paper.

II. Related Work

In early stages, model checking can be performed explicitly by calculating fixpoints on the Kripke structure with CTL properties [15] or by searching states on the product automaton constructed from both the input model and LTL property [16]. Although such explicit techniques are straightforward and easy to understand, they have very poor capability to check large systems and are now replaced by symbolic ones that rely on BDD [17], and more efficiently, SAT [18] solvers as the computation core. In fact, the SAT-based model checking is considered as the most promising automatic verification technique for practical purposes [19].

State-of-the-art SAT-based model checking techniques include BMC [4], [5], IMC [6], IC3/PDR [7], [8], [20] and CAR [9], [12], [10]. BMC is the first approach to reduce model checking to SAT in which every SAT query with a k-unrolling system-input answers whether there is a counterexample with length k. BMC is good at finding bugs (counterexamples), yet it does not handle correctness proofs well [21], [22]. IMC is based on BMC and accomplishes an efficient correctness proof by maintaining an over-approximate states sequence whose elements are refined by the *interpolants* from unsatisfiable SAT queries, see [6]. However, both BMC and IMC may involve the scalability problem on SAT solving, as the size of SAT formula blows up rapidly.

Compared to BMC and IMC, IC3/PDR is able to perform model checking by unrolling the system at most once, thus can significantly ease the heavy computation of a single SAT query (though a much larger number of queries can be invoked than BMC/IMC). Moreover, IC3/PDR computes the so-called *minimal inductive cores* (MIC) to refine the maintained over-approximate sequence such that the proof can converge more efficiently. Notably, IC3/PDR requires the over-approximate sequence to be incremental. From previous evaluation, IC3/PDR has a clear advantage in proving correctness. Subsequently, several extensions on IC3/PDR, e.g., AVY/KAVY [23], [24], QUIP [25] and IC3-INN [26], are proposed but can only enhance the performance on certain benchmarks. Also, the efforts to combine abstractions together with IC3/PDR are investigated, e.g., [27], [28].

Recently, a new model checking algorithm CAR was proposed and received concerns from the community. Compared to IC3/PDR, CAR maintains both the over- and under-approximate sequences, which are used to prove correctness and to find bugs, respectively. Moreover, the over-approximate sequence in CAR are not required to be incremental, as it follows the traditional way used in reachability analysis to

converge the proof. Therefore, CAR provides a more flexible framework and different heuristics can be integrated more easily. For example, the *intersection* and *rotation* strategies have been integrated into CAR and successfully improve the performance [12]. From previous results, CAR is good at bug finding but cannot perform as well as IC3/PDR on proving correctness.

In this paper, we propose four different heuristics based on CAR, which are either IC3/PDR-related or CAR-specific. We explore an efficient way to combine IC3/PDR and CAR together such that the new approach can inherit the advantages from both sides. That is, we leverage the light-weight heuristics from IC3/PDR (excluding the heavy-weight induction) and flexible features from CAR so as to conduct a better model-checking algorithm than both single ones.

III. Preliminaries

A. Boolean Transition System

A Boolean transition system Sys is a tuple (V, I, T), where V and V' denote the set of variables in the present state and the next state, respectively. The state space of Sys is the set of possible variable assignments. I is a Boolean formula corresponding to the set of initial states, and T is a Boolean formula over $V \cup V'$, representing the transition relation. State s_2 is a successor of state s_1 iff $s_1 \wedge s_2' \models T$, which is also denoted by $(s_1, s_2) \in T$. A *path* of length k is a finite state sequence $s_1, s_2, \ldots, s_k$, where $(s_i, s_{i+1}) \in T$ holds for $(1 \leq i \leq k-1)$. A state t is reachable from s in k steps if there is a path of length k from s to t. Let $X \subseteq 2^V$ be a set of states in *Sys*. We denote the set of successors of states in X as $R(X) = \{t \mid (s,t) \in T, s \in X\}$. Conversely, we define the set of predecessors of states in X as $R^{-1}(X) = \{s \mid (s,t) \in T, t \in X\}$. Recursively, we define $R^0(X) = X$ and $R^i(X) = R(R^{i-1}(X))$ where $i \geq 0$, and the notation $R^{-i}(X)$ is defined analogously. In short, $R^i(X)$ denotes the states that are reachable from X in i steps, and $R^{-i}(X)$ denotes the states that can reach X in i steps.

B. Safety Checking and Reachability Analysis

Given a transition system $Sys = (V, I, T)$ and a safety property P, which is a Boolean formula over V, a model checker either proves that P holds for any state reachable from an initial state in I, or disproves P by producing a *counterexample*. In the former case, we say that the system is safe, while in the latter case it is unsafe. A counterexample is a finite path from an initial state s to a state t violating P, i.e., $t \in \neg P$, and such a state is called a *bad* state. In symbolic model checking, safety checking is reduced to symbolic reachability analysis. Reachability analysis can be performed in forward or backward search. Forward search starts from initial states I and searches for reachable states of I by computing $R^i(X)$ with increasing values of i, while backward search begins with states in $\neg P$ and computes $R^{-i}(X)$ with increasing values of i to search for states reaching $\neg P$. Table I gives the corresponding formal definitions.

For forward search, F_i denotes the set of states that are reachable from I within i steps, which is computed by iteratively applying R. At each iteration, we first compute a new F_i, and then perform safe checking and unsafe checking. If the safe/unsafe checking hits, the search process terminates. Intuitively, unsafe checking $F_i \cap \neg P \neq \emptyset$ indicates some bad states are within F_i and safe checking $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ indicates that all reachable states from I has been checked and none of them violate P. For backward search, the set B_i is the set of states that can reach $\neg P$ in i steps, and the search procedure is analogous to the forward one.

TABLE I
Standard reachability analysis.

	Forward	Backward
Base	$F_0 = I$	$B_0 = \neg P$
Induction	$F_{i+1} = R(F_i)$	$B_{i+1} = R^{-1}(B_i)$
Safe Check	$F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$	$B_{i+1} \subseteq \bigcup_{0 \leq j \leq i} B_j$
Unsafe Check	$F_i \cap \neg P \neq \emptyset$	$B_i \cap I \neq \emptyset$

C. SAT Solving and Unsatisfiable Core

Our setting is standard propositional logic. A *literal* is an atomic variable or its negation. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals. Apparently, the negation of a clause is a cube and vice versa. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. For simplicity, we also treat a CNF formula ϕ as a set of clauses and make no difference between the formula and set forms of ϕ. Similarly, a cube or a clause c can be treated as a set of literals or a Boolean formula, depending on the context.

We say a CNF formula ϕ is satisfiable if there exists an assignment of each Boolean variable in ϕ such that ϕ is true; otherwise, ϕ is unsatisfiable if there is no assignment that makes ϕ true. Commonly, we invoke the SAT solver, e.g., MiniSat [29], [30], to decide whether a CNF formula ϕ is satisfiable or not. If ϕ is satisfiable, an assignment of variables, called a model of ϕ, can be returned by the SAT solver. Otherwise, an unsatisfiable subset of clauses in ϕ, i.e., $C \subseteq \phi$, called the *unsatisfiable core* (UC) of ϕ, is returned as the reason why ϕ is unsatisfiable.

A *minimal unsatisfiable core* (MUC) of formula ϕ is a non-reducible UC of ϕ, i.e., removing any elements of the MUC makes the remaining part satisfiable [31]. Formally, for all C, $C \subsetneq MUC$ implies C is satisfiable. Given a UC, the trivial way to extract a MUC from it is to remove clauses of the UC one by one, and then invoke the SAT solver to check whether the left clauses are still unsatisfiable. If this is the case, the dropping clause is not in the MUC and can be removed permanently. Otherwise, the dropping clause is in the MUC. Finally, the remaining part of the UC, which consists of only non-reducible clauses, is the MUC that we want [31].

For an unsatisfiable formula ϕ, we only focus on a part of ϕ and consider the rest unimportant. We call the unimportant part *remainder* of ϕ. Under this premise, we define the term high-level UC and high-level MUC. Given an unsatisfiable formula $\phi = \mathcal{A} \wedge \mathcal{R}$, where $\mathcal{A}$ is the important part of the formula and $\mathcal{R}$ is the *remainder*, we say that C is a high-level UC of ϕ, if $C \subseteq \mathcal{A}$ and $C \wedge \mathcal{R}$ is also unsatisfiable. Analogously, we say

that M is a high-level MUC of ϕ if M is a high-level UC and removing any elements from it makes its conjunction with the remainder satisfiable [32], i.e., $M \subseteq \mathcal{A}$, $M = C_1 \wedge C_2 \wedge ... \wedge C_k$, $M \wedge \mathcal{R}$ is unsatisfiable, but $(M \backslash C_i) \wedge \mathcal{R}$ becomes satisfiable for each $1 \leq i \leq k$. In this paper, we consider the SAT queries with the form SAT($\mathcal{R}, \mathcal{A}$), where $\mathcal{R}$ is a CNF formula and $\mathcal{A}$ is a cube. For the SAT solving, we focus on assumption $\mathcal{A}$ by considering $\mathcal{R}$ as the remainder. We intend to extract the high-level UC or MUC from assumptions, i.e., $C \subseteq \mathcal{A}$.

D. Complementary Approximate Reachability

1) Theory of CAR: In standard forward search, each F_i, as described in Section III-A, is a set of states reachable from I within i steps. To obtain states in F_{i+1}, previous symbolic model checking approaches invoke SAT solver to solve the formula $\phi = F_i(x) \wedge T(x, x')$, and then obtain all states in F_{i+1} from ϕ by projecting to the prime part of the model. In this way, the set of reachable states is computed and a sequence of accurate reachable state sets is maintained, which enables both safe and unsafe checking. Whereas Forward CAR maintains two sequences of reachable states sets: an over-approximate states sets sequence $(O_0, O_1, \dots)$, which contains supersets of reachable states from initial states I, and an under-approximate states sets sequence $(U_0, U_1, \dots)$, which contains subsets of reachable states to bad states $\neg P$. Due to the over-approximation, the O-sequence is only able to perform safe checking, and the U-sequence is used to perform unsafe checking. The formal definitions of these two sequences are shown in Table II.

TABLE II
OVER/UNDER-APPROXIMATE STATE SEQUENCE.

	O-sequence	U-sequence
Base	$O_0 = I$	$U_0 = \neg P$
Induction	$O_{j+1} \supseteq R(O_j)(j \geq 0)$	$U_{j+1} \subseteq R^{-1}(U_j)(j \geq 0)$
Constraint	$O_j \subseteq P(0 \leq j)$	–

We call each O_i $(i \geq 0)$ a frame. We also define the notation $S(O) = \bigcup_{0 \leq j \leq n} O_j$, which is the set of all states in the O-sequence (suppose the O-sequence is of size n), and $S(U) = \bigcup_{0 \leq j \leq m} U_j$ denotes the set of states in the U-sequence (suppose m is the length of the U-sequence). As we have mentioned above, each O_{i+1} is an over-approximate states set reachable from O_i in one step, and we make no difference between the following representation of O_i: a set of states, a set of clauses, or a Boolean formula in CNF. Analogously, each U_{i+1} is an under-approximate states set reachable to U_i in one step, and we make no difference between the following representation of U_i: a set of states, a set of cubes, or a Boolean formula in DNF. Moreover, the length of these two sequences is not required to be the same. The following theorems w.r.t. the O- and U-sequences are used to guarantee the correctness of safe and unsafe checking in CAR.

Theorem 3.1 (Safe Checking): Given a system Sys and a safety property P, Sys is safe for P iff there exists an O-sequence $(O_0, O_1, \dots, O_i, O_{i+1})$ with $i \geq 0$ such that $O_{i+1} \subseteq \bigcup_{0 \leq j \leq i} O_j$.

Theorem 3.2 (Unsafe Checking): Given a system Sys and a safety property P, Sys is unsafe for P iff there exists a U-sequence $(U_0, U_1, \dots, U_i)$ with $i \geq 0$ such that $I \cap U_i \neq \emptyset$.

In addition, Theorem 3.3 below reveals a direction to refine the O-sequence and update the U-sequence, which tries to make the states in these two sequences unreachable. Refinement details are shown in Theorem 3.4.

Theorem 3.3 (Sequences Refinement): Given a system Sys and a safety property P, Sys is safe for P iff there exists an O-sequence, such that for every U-sequence $S(O) \cap R^{-1}(S(U)) = \emptyset$ holds.

Theorem 3.4: Given an O-sequence $(O_0, O_1, \dots)$, a U-sequence $(U_0, U_1, \dots)$, a cube $c_1 \in U_j (j \geq 0)$ and the formula $\phi = O_i(x) \wedge T(x, x') \wedge {c_1}'(x')(i \geq 0)$:

(1) If ϕ is satisfiable, there is a cube c_2 such that every state $t \in c_2$ is a predecessor of some state s in c_1 and $t \in O_i$. By updating $U_{j+1} = U_{j+1} \cup \{c_2\}$, the sequence is still a U-sequence;
(2) If ϕ is unsatisfiable, $\{c_1\} \cap R(O_i) = \emptyset$. Moreover, there is a cube c_2 such that $c_1 \Rightarrow c_2$ and $\{c_2\} \cap R(O_i) = \emptyset$. By updating $O_{i+1} = O_{i+1} \cup \{\neg c_2\}$, the sequence is still an O-sequence.

In the theorem above, the first item suggests to add a set of states rather than a single one to the U-sequence. Analogously, the second item suggests to refine the O-sequence by blocking a set of states rather than a single one. These goals can be achieved by utilizing the UC returned from the SAT solver. In both situations, it will speed up the computation. Similarly to the standard reachability analysis, Backward CAR performs the same framework on $Sys^{-1} = (V, \neg P, T^{-1})$ with respect to $\neg I$.

2) Algorithm of CAR: Then we describe the framework of CAR, in which we integrate the proposed four heuristics. The main part of CAR Algorithm is described between Line 1 and 8 of Algorithm 1. The lines in red are the heuristics we propose and do not belong to the original CAR algorithm. The main part takes $Sys = (V, I, T)$ and a safety property P as inputs, and returns safe if Sys satisfies P or unsafe with an *counterexample* if not.

The main procedure first checks whether the initial states intersect with the bad states (Line 1). Then the O- and U-sequences are initialized (Line 2). In the main loop (Line 3-8), CAR performs unsafe and safe checking iteratively. For unsafe checking (Line 4-5), CAR picks a state s from the U-sequence and invokes the UNSAFECHECK procedure to check if s is reachable from the initial states. The PICKSTATE function enumerates states in the U-sequence (Line 4). For safe checking, the SAFECHECK procedure checks if the O-sequence has included all states that are reachable from the initial states (Line 7).

The UNSAFECHECK procedure (Line 10-22) takes as inputs a state s in the U-sequence and the current frame level i of the O-sequence. The UNSAFECHECK procedure checks whether s is reachable from the states in O_i by making the SAT query $SAT(O_i \wedge T, s')$ (Line 11). If the result is true, the returned assignment t (Line 13) is a state which can reach s, i.e.,

Algorithm 1 Implementation of Forward CAR

```
1:  if SAT(I, ¬P) then return unsafe
2:  O_0 := I, U_0 := ¬P, k := 0, Deads := ∅
3:  while true do
4:      while (Cube s := PickState(U)) ≠ ∅ do
5:          if UnsafeCheck(s, k) then return unsafe;
6:      if Propagation(k) then return safe
7:      if SafeCheck(k) then return safe
8:      k := k + 1
9:
10: procedure UnsafeCheck(s, i)
11:     while SAT(O_i ∧ T, s') do
12:         if i = 0 then return true
13:         Cube t := GetAssignment()
14:         Cube Input_t := GetInput()
15:         t := GetPartial(t, Input_t, s)                ▷ optional
16:         U_{j+1} := U_{j+1} ∪ t supposing s is in U_j (j ≥ 0)
17:         if UnsafeCheck(t̂, i − 1) then return true
18:     Cube c := GetUnsatCore()
19:     Cube ĉ := ExtractMuc(c', O_i ∧ T)
20:     O_{i+1} := O_{i+1} ∩ ¬ĉ
21:     BlockDead(s)
22:     return false
23:
24: procedure SafeCheck(k)
25:     i := 0
26:     while i < k do
27:         if not SAT(O_{i+1} ∧ ¬(⋁_{0≤j≤i} O_j)) then
28:             return true
29:     return false
```

$SAT(t \wedge T, s')$ is satisfiable. If $i = 0$, then s is reachable from the initial states in O_0 (Line 12), which indicates a counterexample is detected. Otherwise, we add t to U_{j+1} according to Theorem 3.4, supposing that s is in U_j, then we recursively check whether t is reachable from states in O_{i-1} (Line 17). If the SAT query returns false, we get an unsatisfiable core $c \subseteq s$ from the SAT solver (Line 18) and add $\neg c$ to O_{i+1} according to Theorem 3.4. Note that c is an over-approximation of states that contains s and is not reachable from O_i, so we add $\neg c$ to O_{i+1} to block states in c. Intuitively, the shorter c is, the more states c represents, and the more states $\neg c$ can block at O_{i+1}.

The SafeCheck procedure at Line 24-29 takes as input the maximal frame level k of the O-sequence. At Line 27, CAR checks whether O_{i+1} is contained in the union of frames in the O-sequence whose indexes range from 0 to i (including 0 and i). If this is the case, all states reachable from the initial states are already included in O-sequence, leading to the conclusion that the system is safe w.r.t the safety property P.

Above is the framework of forward CAR, as for the backward direction, the main framework is similar. While the main difference is that the two sequences have different meanings and that we check if a picked state s can reach certain O_i in Line 11 of Algorithm 1, i.e., SAT($s \wedge T, O_i'$), hence the backward CAR searches for the successors of a state instead of the predecessors.

IV. Motivating Example

In this section, we demonstrate a motivating example in Figure 1 to explain the difference between CAR and IC3/PDR. There are 8 states in total, 000 is the initial state and 111 is the only state in $\neg P$ (bad state). The transition relation is already drawn out explicitly, and we can tell directly that this model satisfies the safety property since the state 111 is unreachable from the initial state.

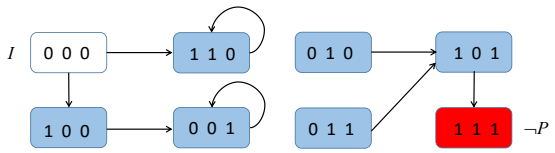


Fig. 1. Example to demonstrate the difference between CAR and IC3/PDR.

A. Model Checking by IC3/PDR

First we give a brief intuition of how IC3/PDR works. IC3/PDR maintains an over-approximate sequence O, and makes SAT queries to check whether certain O_i can reach the bad state. If unsat, IC3/PDR updates the O-sequence with the UC returned. Else, IC3/PDR obtains a new state and recursively checks whether O_{i-1} can reach the new state. The over-approximate sequence O in IC3/PDR is initialized by setting $O_0 = I$ and $O_1 = P$, i.e., O_0 is the initial state and O_1 contains all states except for the bad state 111. IC3/PDR starts to check whether states in O_1 can reach $\neg P$ by making the SAT query $SAT(O_1 \wedge T \wedge \neg P')$, and the SAT solver returns that the state 101 can reach $\neg P$. Then IC3/PDR recursively checks whether O_0 can reach 101. Since it is unreachable, we assume that the unsatisfiable core returned is 101 itself. Therefore we block 101 from O_1, and since all states in O_1 cannot reach $\neg P$, a new frame $O_2 = P$ is created.

IC3/PDR repeats the previous procedure by checking whether states in O_2 can reach $\neg P$, and it turns out that 101 can reach $\neg P$. Then it checks whether 101 is reachable from O_1. From the figure, the states 010 and 011 can reach 101, but they are not reachable from O_0. Therefore IC3/PDR blocks 010 and 011 from O_1. After that there are only four states inside O_1, i.e., $O_1 = \{000,110,100,001\}$. Since 101 is no longer reachable from O_1, IC3/PDR blocks 101 from O_2.

Note that there is a propagation heuristic in IC3/PDR, which tries to block the same UC in different frames. Since previously 010 and 011 are blocked in O_1, IC3/PDR now tries to block them in O_2 by checking whether those two states are reachable from O_1. Obviously, they are not. Therefore IC3/PDR blocks 010 and 011 in O_2. Afterwards we have $O_2 = \{000,110,100,001\}$. Now since $O_1 = O_2$ is true, which means that states inside O_1 cannot transit out of O_1, O_1 is an *invariant*, or say, *proof certificate*. The safety property holds and the IC3/PDR algorithm terminates.

B. Model Checking by CAR

We consider here Forward CAR and demonstrate its execution on the above example. Similar to IC3/PDR, the first step is to initialize the sequence, except that we need to initialize both the over-approximate O-sequence and the under-approximate U-sequence. According to Line 2 of Algorithm 1, we initialize the two sequences by setting O_0 to the initial state and setting U_0 to the bad state, i.e., $O_0 = \{000\}$, $U_0 = \{111\}$.

In step 0, CAR takes the bad state 111 from U_0 and checks whether O_0 can reach 111 by making the SAT query $SAT(O_0 \wedge T \wedge 111')$ (Line 11 of Algorithm 1). Obviously, the initial state cannot reach $\neg P$ and suppose the UC returned by the SAT solver is that the third bit of the state is 1. Then CAR adds a new frame $O_1 = P$ and blocks the UC from O_1 according to Line 20 of Algorithm 1, which means that all states containing the third bit of 1 are deleted from O_1, i.e., $O_1 = \{000,100,110,010\}$.

In step 1, CAR repeatedly takes 111 from U_0, and checks whether O_1 can reach 111. Obviously states in O_1 cannot reach 111 and suppose the UC returned by the SAT solver is that the last two bits of the state are both 1. Then CAR adds a new frame $O_2 = P$ and blocks the UC from O_2, i.e., $O_2 = \{000,100,110,001,010,101\}$.

In step 2, CAR repeatedly takes 111 from U_0 and checks whether O_2 can reach 111. It turns out that $101 \in O_2$ can reach 111 . Then CAR creates a new frame U_1 and adds 101 to U_1 according to Line 16 of Algorithm 1, i.e., $U_1 = \{101\}$. Afterwards CAR recursively checks whether O_1 can reach 101, clearly $010 \in O_1$ can reach 101. Likewise, CAR creates a new frame U_2 and adds 010 to U_2, i.e., $U_2 = \{010\}$. CAR recursively checks whether 010 is reachable from O_0. Obviously, it is not. Suppose that the UC is 010 itself, then the UC is blocked from O_1, i.e., $O_1 = \{000,100,110\}$. Then CAR in turn backtracks to check whether O_1 can reach 101 and whether O_2 can reach 111. Obviously, they are both unreachable. Suppose that the UC to block is both the states themselves, i.e., $O_2 = \{000,100,110,001,010\}$, $O_3 = P$.

In step 3, CAR takes $010 \in U_2$ and checks whether O_3 can reach 010, clearly it is not and CAR blocks 010 in O_4. Afterwards CAR takes $101 \in U_1$ and checks whether O_3 can reach 101. Obviously $010, 011 \in O_3$ can reach 101 and CAR recursively checks whether they are reachable from O_2. Clearly, both 010 and 011 are not, therefore they are all blocked from O_3. Now that no states in O_3 can reach 101, CAR then takes $111 \in U_0$ and checks whether O_3 can reach 111. Likewise, 111 is reachable from $101 \in O_3$ but 101 is not reachable from O_2, therefore 101 is removed from O_3, i.e., $O_3 = \{000,100,110,001\}$.

Note that $O_3 = \{000,100,110,001\}$ and $O_2 \cup O_1 \cup O_0 = \{000,100,110,001,010\}$. Therefore $O_3 \subseteq O_2 \cup O_1 \cup O_0$ holds, which means that states in O_3 can only be reachable from states in O_2, O_1 or O_0. Therefore, CAR has successfully found an invariant and proved that the safety property holds.

V. Our Approaches

We propose four heuristics and integrate them in Algorithm 1 (as shown in the text in red.) At Line 6, we propose a heuristic called *UC-propagation*, which checks whether there exist element of $O_i (i \geq k)$ in the O-sequence that can also be contained in O_{i+1}. Moreover, such heuristic can produce a safe result if there exists $O_i (i \geq k)$ such that all of its elements can be propagated to O_{i+1}. This heuristic complements the SafeCheck procedure. At Line 15, there could be more then one full state t that satisfies the SAT query $SAT(t \wedge T, s')$, i.e, there could be a partial assignment $\hat{t}$ such that $\hat{t}$ also satisfies $SAT(\hat{t} \wedge T, s')$ and $\hat{t} \supseteq t$ holds. By replacing full assignment t with such partial assignment $\hat{t}$, the computation speeds up and the whole UnsafeCheck procedure can be accelerated. Therefore we propose a heuristic GetPartial(t, s) to produce a partial assignment $\hat{t}$ from t. We propose the heuristic of MUC-extraction at Line 19, which can extract a MUC from c, to block more states at O_{i+1} and accelerate the convergence of the SafeCheck procedure. At Line 21 ,we propose a heuristic BlockDead(s) to block s if it does not have any predecessors. In this case, we call s a dead state and block it as the constraint in the SAT solver, with the purpose of accelerating the computation.

All the proposed heuristics can be applied in the Forward CAR. While the Dead-state detection and partial-state heuristic cannot be applied in Backward CAR, as both of them are based on obtaining the predecessors of states. As for the MUC-extraction heuristic and UC-propagation heuristic, since Backward CAR also depends on obtaining UC and blocking UC in the O-sequence, both heuristics can be adopted analogously.

In the following subsections, we present in order the proposed heuristics, namely, MUC-extraction, UC-propagation, Partial-state generation, and Dead-state detection, which successfully boost the safe-checking performance of CAR.

A. Heuristic I : MUC-Extraction

For the purpose of boosting the safe-checking performance of CAR, we focus on accomplishing a faster convergence of the over-approximate O-sequence. Notably, CAR adds UC returned from the SAT solver to the O-sequence to block states that are not reachable from the states in O-sequence, i.e., $O_{i+1} := O_{i+1} \cap \neg c$. As a result, a smaller UC represents more states and thus blocks more states in the O-sequence. Moreover, UC is also obtained and used to represent a states set when we calculate dead state and partial state, extracting a smaller UC out of the UC clearly benefits both procedures. Therefore, we propose our first heuristic to extract the minimal unsatisfiable core (MUC) from a UC.

Instead of using off-the-shelf tools to extract MUC, which is both inconvenient and time-wasting, we choose to implement the MUC-extraction algorithm [31], [32] inside CAR. Given a UC, i.e., $c = l_1 \wedge l_2 \wedge ... \wedge l_n$, we drop literals in c one by one and then invoke the SAT solver to check whether the unsatisfiable result is preserved. If not, the literal is in the MUC and should be restored. Otherwise, the literal is not in an MUC and can be removed permanently. Moreover, since the SAT query returns unsatisfiable, we obtain a smaller UC from the SAT solver, which accelerates the literal dropping process.

The ExtractMuc procedure (Algorithm 2) takes a pre-acquired unsatisfiable core Uc together with the corresponding

Algorithm 2 MUC-Extraction

```
1:  procedure ExtractMuc(Uc, remainder)
2:      Muc := ∅
3:      while Uc ≠ ∅ do
4:          l := GetLiteral(Uc)
5:          Uc := Uc \ {l}
6:          if SAT(remainder, Uc ∧ Muc) then
7:              Muc := Muc ∪ {l}
8:          else
9:              Cube uc := GetUnsatCore()
10:             Uc := uc \ Muc
11:     if IsPrime(Uc) then
12:         Muc := UnPrime(Muc)
13:     return Muc
```

Algorithm 3 UC-Propagation

```
1:  procedure Propagation(k)
2:      for 0 ≤ i ≤ k − 1 do
3:          flag := true
4:          for (¬uc) ∈ O_i do
5:              if not SAT(O_i ∧ T, uc') then
6:                  Cube new_uc := GetUnsatCore()
7:                  O_{i+1} := O_{i+1} ∩ ¬new_uc
8:              else
9:                  flag = false;
10:         i := i + 1
11:         if flag then return true
12:     return false
```

$remainder$ as inputs. The literal set Muc in Algorithm 2 stores the literals which belong to the MUC and is initialized at Line 2. In the main loop (Line 3-10), when Uc is not empty, a literal $l \in Uc$ is chosen and dropped from Uc (Line 5). Then it checks if l is in MUC by making the SAT query $SAT(remainder, Uc \wedge Muc)$ (Line 6). If the SAT solver returns true, l is in MUC and then added to Muc (Line 7). Otherwise, l is redundant and can be removed permanently. Moreover, the SAT solver helps to remove some extra literals in c by returning a smaller UC uc (Line 9), i.e., $uc \subseteq Uc \wedge Muc$, so Uc can be replaced by $uc \wedge Muc$ (Line 10). When all literals in Uc have been dropped, i.e., $Uc = \emptyset$, we have successfully extracted an MUC out of Uc, and return Muc as the MUC we want (Line 13). Since Muc may contain primed literals, Line 11-12 are added to un-prime such literals.

Note that the correctness of using MUC instead of UC to refine the O-sequence is straightforward, since MUC is a special kind of UC.

B. Heuristic II : UC-Propagation

In the SafeCheck procedure, CAR checks whether there exists some i ($i \leq k$) such that O_{i+1} is contained in the union of O_j, where j ranges from 0 to i, by making the SAT query $SAT(\emptyset, \neg(O_{i+1} \Rightarrow (\bigvee_{0 \leq j \leq i} O_j)))$. However, as the length of O-sequence grows, the formula $O_{i+1} \Rightarrow (\bigvee_{0 \leq j \leq i} O_j)$ also grows in size, and this makes it difficult for the SAT solver to solve the formula and ultimately reduces the convergence speed of safety checking. To mitigate this problem, we propose our second heuristic, namely UC-propagation, which provides another safe checking procedure in CAR. The motivation of the UC-propagation heuristic directly comes from the fact that the safe checking may converge faster if more elements in O_i can also appear in O_{i+1}. In particular, we prove that if there is O_i such that all of its elements are also the elements of O_{i+1}, the safe result can be concluded. To that end, UC-propagation makes the SAT query $SAT(O_i \wedge T, uc')$ for every $\neg uc \in O_i$ to decide if $\neg uc$ can be added to O_{i+1}. If all $\neg uc \in O_i$ falsify the query, then $O_{i+1} \subseteq \neg uc_1 \wedge \neg uc_2 \wedge ... \wedge \neg uc_n = O_i$ holds, thus the system is safe.

The Propagation procedure (Algorithm 3) takes the maximal frame level k of the O-sequence as the input. The main loop (Line 2-11) enumerates $0 \leq i \leq k-1$ to check whether $O_{i+1} \subseteq O_i$ holds for some i. $flag$ is a variable initialized to be true (Line 3). The second loop (Line 4-9) enumerates $\neg uc \in O_i$ and makes the SAT query $SAT(O_i \wedge T, uc')$ (Line 5). This query asks whether uc is reachable from O_i. If the result is false, states in uc are not reachable from O_i, which indicates that $\neg uc$ represents a set of states that O_i cannot reach. Therefore, $\neg uc$ can be added to O_{i+1}. Moreover, we get a smaller UC $new\_uc \subseteq uc$ from the SAT solver (Line 6), the negation of which can be added into O_{i+1} instead of $\neg uc$ (Line 7). Notably, new_uc represents more states than uc, thus $\neg uc$ contains $\neg new\_uc$ and we have $O_{i+1} \subseteq \neg uc$ after $\neg new\_uc$ is added to O_{i+1}. Notably, IC3/PDR does not compute new_uc when doing the propagation. If some propagation query returns true, $flag$ is set to false, meaning that $\neg uc$ cannot be added to O_{i+1}. Otherwise, the whole propagation procedure returns true, which implies that convergence is committed.

Theorem 5.1: If the UC-propagation algorithm returns true, the system is proved to be safe.

Proof: Algorithm 3 returns true if for some i, all $\neg uc \in O_i$ falsify the query, and we have $O_{i+1} \subseteq \neg uc_1 \wedge \neg uc_2 \wedge \neg uc_3 \wedge ... \wedge \neg uc_n = O_i$. As a result, $O_{i+1} \subseteq (\bigvee_{0 \leq j \leq i} O_j)$ holds, which meets the safe checking condition. ■

C. Heuristic III : Dead State Detection

In the UnsafeCheck procedure, CAR checks whether the state s is reachable from the states in O_i by making the SAT query $SAT(O_i \wedge T, s')$. If it is satisfiable, a state t that can reach s is obtained and CAR recursively checks if $SAT(O_{i-1} \wedge T, t')$ holds. Whereas an unsatisfiable result suggests that s is not reachable from O_i and therefore can be blocked in O_{i+1}. Note that s may be revisited, i.e., an analogous SAT query $SAT(O_j \wedge T, s')(j \neq i)$ can be made. We notice that there exists such a state s that $SAT(O_i \wedge T, s')$ does not hold for an arbitrary i, even further we find out that $SAT(T, s')$ is actually unsatisfiable. That is to say, s is unreachable from any other states, and we refer to such a state s as a *dead state*, which can be illustrated as in Figure 2. By

Algorithm 4 The Dead-State Detection Implementation.

1: **procedure** BLOCKDEAD(s)
2: **if not** $SAT(\neg Deads \wedge T, s')$ **then**
3: $Cube\ uc \coloneqq$ GETUNSATCORE()
4: $uc \coloneqq$ EXTRACTMUC($uc', \neg Deads \wedge T$)
5: $Deads \coloneqq Deads \cup uc$
6: add $\neg uc$ to SAT solver

blocking such dead states in the SAT solver, we can avoid revisiting them and therefore save time to visit other states. Moreover, since the SAT query $SAT(T, s')$ returns false, we can obtain a UC from s, which we refer to as dead UC, and therefore more states can be blocked by adding the negation of the dead UC to the SAT solver. All UC obtained is added to $Deads$, which denotes all the detected dead states. Notice that if s is *only* reachable from a dead state (within one or more transitions), we can also block s in SAT solver, for that visiting s will eventually lead us to a dead state. Therefore, we expand the definition of a dead state to the following definition 5.1.

Definition 5.1: A state s is a dead state if one of the following conditions holds:

(1) $SAT(T, s')$ is unsatisfiable;
(2) s can only be reached from dead states.

The BLOCKDEAD(s) procedure (Algorithm 4) first checks whether $SAT(\neg Deads \wedge T, s')$ is unsatisfiable (Line 2), if so, either s is unreachable from any other states, i.e., $SAT(T, s')$ is unsatisfiable, or s is only reachable from states in $Deads$, i.e., $SAT(Deads \wedge T, s')$ holds, we consider s a dead state either way. Then we obtain a UC, i.e., $uc \subseteq s$ (Line 3) such that $SAT(\neg Deads \wedge T, uc')$ is also unsatisfiable. Then uc is added to $Deads$ (Line 5) and is blocked in SAT solver (Line 6). Moreover, we adopt a EXTRACTMUC($uc', \neg Deads \wedge T$) procedure (Line 4), so that we can generate a MUC out of UC and block more dead states.

Regarding the complexity of BLOCKDEAD(s), it can be considered as follows: given s, the procedure is invoked at most once in each frame level, according to Algorithm 1. Also, it is not hard to see that in the worst case, there can be N frame levels computed by CAR, where N is the number of all states in the system. As a result, BLOCKDEAD(s) can be invoked at most N^2 times.

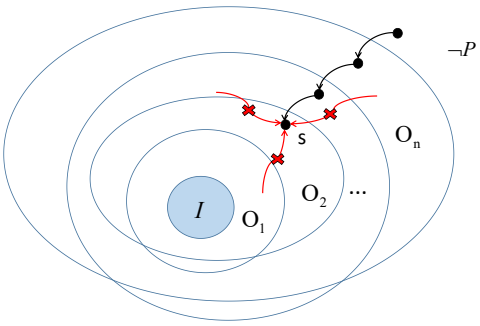


Fig. 2. A high-level illustration of the Dead-state detection heuristic. The state s is a dead state because no other state can reach it in an arbitrary step.

Algorithm 5 The Partial-State Generation Implementation

1: **procedure** GETPARTIAL($t, Input_t, s$)
2: **Assert** (**not** $SAT(Input_t \wedge T \wedge \neg s', t)$)
3: $Cube\ uc \coloneqq$ GETUNSATCORE()
4: $uc \coloneqq$ EXTRACTMUC($uc, Input_t \wedge T \wedge \neg s'$)
5: **return** uc

Theorem 5.2: The correctness of CAR is preserved after integrating the Dead-state detection heuristic.

Proof: According to Algorithm 4, dead states are blocked forever in the SAT computation (Line 6). Also by Definition5.1, dead states are those that cannot be reachable from initial ones, which can be safely pruned during the state search process. As a result, the integration of the heuristic does not affect the correctness of the CAR framework. ■

D. Heuristic IV : Partial State Generation

In the UNSAFECHECK procedure, CAR makes the SAT query $SAT(O_i \wedge T, s')$ to find out whether s is reachable from O_i. If satisfiable, a full state $t \in O_i$ is obtained such that $SAT(t \wedge T, s')$ is satisfiable. However, there could be more than one full state which satisfies $SAT(t \wedge T, s')$, i.e., there could be a partial assignment $\hat{t}$ such that $\hat{t}$ satisfies $SAT(\hat{t} \wedge T, s')$ and $\hat{t} \supseteq t$ holds, and we call such partial assignment $\hat{t}$ a partial state of t. Notice that $\hat{t}$ represents a set of states that can reach s, therefore we can avoid visiting states inside $\hat{t}$ separately by replacing t with $\hat{t}$, and the computation speeds up.

The GETPARTIAL($t, Input_t, s$) procedure (Algorithm 5) takes a state t, the input $Input_t$ corresponding to t and a state s reachable from t as inputs. Firstly it asserts the SAT query $SAT(Input_t \wedge T \wedge \neg s', t)$ is unsatisfiable, given that t can reach s, i.e., $SAT(t \wedge T, s')$ is satisfiable, therefore t can not reach $\neg s$ with the corresponding input $Input_t$, i.e, $SAT(Input_t \wedge T \wedge \neg s', t)$ is unsatisfiable. Under this assertion, an unsatisfiable core uc can be obtained in Line 3 and uc represents multiple states that can reach s. Analogously, we extract a MUC from uc (Line 4) to expand the partial states.

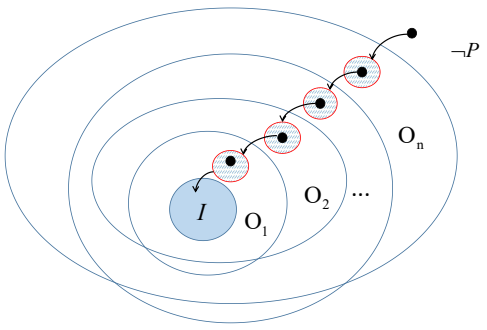


Fig. 3. An illustration of the Partial-state generation heuristic. Each red region represents a set of states which can be covered by a partial state.

Theorem 5.3: The correctness of CAR is preserved after integrating the Partial-state detection heuristic.

Proof: Assume $\hat{t}$ is the partial state of t. According to Algorithm 5, $SAT(Input_t \wedge T \wedge \neg s', \hat{t})$ is unsatisfiable, as $\hat{t}$

is the UC returned by SAT solver. Therefore, $\hat{t}$ represents a set of full state that can reach s (with the same input values). Note that s is a state which can reach the bad state and has already been added into the U-sequence (Line 16 of Algorithm 1). So $\hat{t}$ can also reach the bad state as $\hat{t}$ can reach s. Therefore adding $\hat{t}$ into the U-sequence does not change the meaning of the U-sequence. On the other hand, when we check whether certain O_i can reach $\hat{t}$, and assume that it turns out unsatisfiable. Then we can obtain a UC out of $\hat{t}$ and add the UC into O_{i+1}. The meaning of the O-sequence is also maintained as the UC represents a set of state that are not reachable for O_i. Therefore, the integration of the heuristic does not affect the correctness of the CAR framework.

■

Notably, the Partial-state generation heuristic is also integrated into IC3/PDR [7], [8], and it is one of the two main *state-generalization* techniques (The other one is called *ternary simulation* [8]). Also, this heuristic is only applicable to Forward CAR, because the assumption that the SAT query $SAT(Input_t \wedge T \wedge \neg s', t)$ is unsatisfiable does not hold for Backward CAR.

E. Motivating example of accelerating CAR by heuristics

Now we demonstrate how heuristics proposed in this paper affect the performance of CAR. To describe the idea of CAR in a simple way, the MUC-extraction heuristic has already been in consideration for the above workflow, as the UCs returned from SAT solver are minimal. One can see if the sizes of returned UCs are larger, more frames and states have to be involved before CAR is able to return safe.

For Dead-state detection, it is easy to see directly that the states 010, 011 are dead states (they both have no predecessors), which implies that state 101 is also a dead state (all its predecessors are dead state). Therefore, we can conclude that the bad state 111 is a dead state as well. And the correctness proof in CAR can be easier made.

For UC-propagation, in step 2 of Section IV-B, we can propagate the UC to different frames, so we propagate the previously obtained UC in O_1 to O_2 as states represented by the UC are not reachable from O_1, i.e., $O_2 = \{000,100,110,001\}$. In this way, O_2 is smaller than that shown above, which makes CAR converge more quickly.

For Partial-state generation, in step 3 of Section IV-B, the partial state that can reach 101 is $01x$, in which x can be either 0 or 1. By using the partial state $01x$ instead of the two independent states 010 and 011, CAR is able to save the effort of enumerating the states later and therefore accelerates the checking process.

VI. Experiments

A. Experiment Setup

We evaluate all tools upon 749 instances in the *aiger* format [33] from the benchmarks of the SINGLE safety property track of the HWMCC in 2015 and 2017. We perform the experiments on a cluster, which consists of 2304 processor cores in 240 nodes running RedHat 4.8.5 and running at 2.5GHz, 96GB of memory. We set the memory limit to 8GB and the time limit to 1 hour, following the same resource settings in [28] and [12]. When we perform experiments, each model-checking run has exclusive access to a dedicated node.

We integrate all proposed heuristics into the SimpleCAR open-source model checker which implemented the CAR algorithm [10]. We first demonstrate the overall performance of CAR with all those heuristics, and then we analyze the impact of each heuristic.

We consider pure Forward and Backward CAR without any heuristics as the baseline of our experimental evaluation, and denote them by using -f and -b flags in the following tables and graphs. Upon the baseline, we activate each of the heuristics to evaluate their impact. In the following tables and graphs, -m, -pr, -d and -pa respectively denote MUC-extraction, UC-propagation, Dead-state detection and Partial-state generation, e.g. -f-m-pr-d-pa means that we run Forward CAR with all those four heuristics, and -b-m-pr means that we run Backward CAR with the MUC and UC-propagation heuristic. All tools and algorithms evaluated in the experiments are listed in Table III. Since we evaluate 14 configuration combinations for CAR in the experiments, for convenience we only list 4 of them in the table, and the other combinations are analogous.

To check the correctness of the results from different tools (with different parameters), we compare results from other solvers to make sure they are consistent. Excluding timeout instances, all results among different checkers are consistent. The artifacts (including the source code and experimental results) are available at [34].

TABLE III
Tools and algorithms evaluated in the experiments.

Tool	Algorithm	Configuration Flags
ABC [35]	PDR (ABC-pdr)	-c 'pdr'
IIMC [36]	IC3 (iimc-ic3)	-t ic3
	IC3 (iimc-ic3r)	-t ic3r
IC3 Ref [37]	IC3 (ic3-ref)	-b
SimpleCAR [38]	Forward CAR (-f)	-f
	Forward CAR (-f-m-pr-d-pa)	-f -muc -propagate -dead -partial
	Backward CAR (-b)	-b
	Backward CAR (-b-m-pr)	-b -muc -propagate

B. Overall Results

We first present the overall experiment results, in which we compare CAR's performance to different implementations of IC3/PDR, which is often considered the most efficient complete model checking algorithm. CAR can perform in both the forward and backward directions, yet the IC3/PDR implementation of ABC only performs the forward direction. To be fair, we also involve the iimc checker that can perform the reverse IC3/PDR, i.e., the backward direction of IC3/PDR. Meanwhile, the ic3-ref implementation is selected so as to add diversity. Figure 4 shows the comparison of the overall performance among different approaches, depicting the growth of total solved instances (the y-axis) in the given CPU time (the x-axis). And the detailed experimental data is in Table IV and Figure 5.

As Figure 4 and Table IV show, when all heuristics are turned on, Forward CAR can solve 356 instances and Backward CAR can solve 355 instances. *Comparing to CAR*

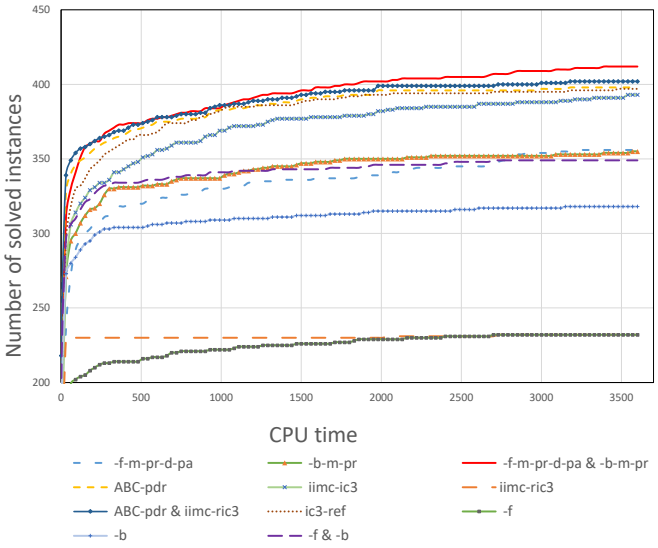


Fig. 4. The comparison of overall performance for different approaches.

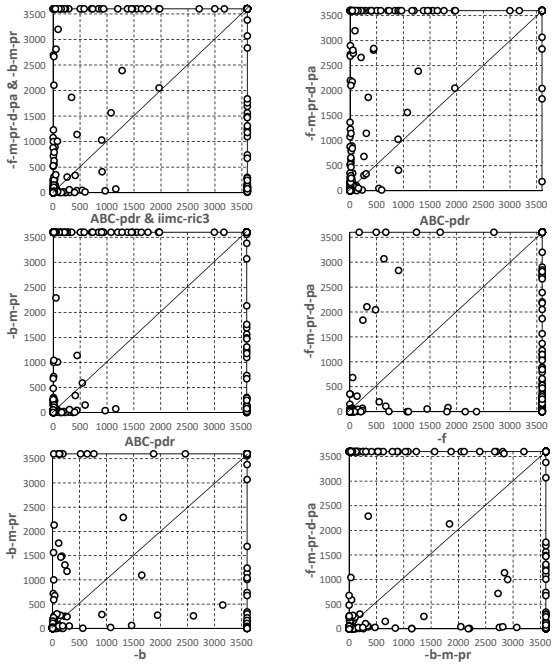


Fig. 5. Comparison between different approaches. Points above the diagonal represent that the approach of the x-axis has better performance, while points below the diagonal represent the opposite.

without any heuristics, there is a 53.4% boost in the number of solved instances for Forward CAR, and an 11.6% boost for Backward CAR. However, Backward CAR with all heuristics uniquely solves 43 instances compared to ABC-pdr & iimc-ic3r, while Forward CAR can only uniquely solve 5 instances. For different IC3/PDR implementations, although ABC-pdr solves 398 instances, which is the best result, the performances of other IC3/PDR implementations are fairly close to the best one.

Combining both Forward and Backward CAR together, integrating all heuristics presented in this paper (totally solve 412 instances) is able to outperform the combination of Forward and Backward IC3/PDR (totally solve 402 instances), which solves 10 more instances than the combined IC3/PDR and can uniquely solve 44 instances. There is also an 18% boost compared to the combination of the original Forward and Backward CAR. For the single direction, Forward CAR can solve $> 90\%$ of the instances that are solved by ABC-pdr (that is, 356 of 398), which performs slightly less than Forward IC3/PDR. However, Backward CAR performs much better than Backward IC3/PDR (iimc-ic3r), considering the amounts of solved instances are 355 and 232, respectively. Moreover, most cases solved by Backward IC3/PDR can also be solved by Forward IC3/PDR, which indicates that Backward IC3/PDR is not that useful when combined together with Forward IC3/PDR. Meanwhile, *Backward CAR can be the sufficient complement of Forward CAR, because it is able to contribute 44 uniquely-solved instances*.

TABLE IV
DETAILED EXPERIMENTAL RESULTS OF DIFFERENT APPROACHES. THE TIMEOUT INSTANCES ARE GIVEN A TIME OF 3600S TO CALCULATE THE AVERAGE TIME OF 749 BENCHMARKS.

Approaches	# Solved	#Uniquely solved (compare to ABC-pdr & iimc-ic3r)	Average time (sec)
CAR-f-m-pr-d-pa & CAR-b-m-pr	**412**	**44**	1710
ABC-pdr & iimc-ic3r	402	0	1728
CAR-f & CAR-b	349	47	1957
ABC-pdr	398	0	1745
ic3-ref	397	27	1760
iimc-ic3	393	22	1813
CAR-f-m-pr-d-pa	356	5	1988
CAR-b-m-pr	355	43	1952
CAR-f	232	8	2521
CAR-b	318	44	2106
iimc-ic3r	232	0	2493

We investigate the instances that Forward CAR cannot solve but IC3/PDR can solve. The observation is that IC3/PDR updates the over-approximate sequence more efficiently than (Forward) CAR. Take the instance *6s52.aig* for example, ABC-pdr extends 321 frames in the over-approximate sequence within 529.54s and then finds the invariant to return safe, while CAR only extends 9 frames within the 1h time limit. In principle, both IC3/PDR and CAR prove that the model satisfies the property within up to K steps if they successfully extend the size of the over-approximate sequence to K. The reason is that the elements of each frame computed by MUC in CAR are not as "good" as those computed by MIC in IC3/PDR, making CAR converge slower than IC3/PDR. It is interesting to explore further whether CAR can do better in refining the over-approximate sequence, which can influence the model-checking performance significantly.

In summary, we conclude that by integrating the four heuristics in this paper, CAR in the forward and backward direction is able to outperform IC3/PDR in both directions, though for the single direction CAR (in forward or backward) is still a bit less competitive as (Forward) IC3/PDR. We also observe that Backward IC3/PDR is far less useful than Backward CAR in a portfolio, which indicates that CAR is also a promising model-checking technique other than IC3/PDR.

C. Detailed Evaluation of Different Heuristics

Now we evaluate the impact of different heuristics by activating each of those heuristics separately in different configurations of CAR. For example, to find out whether the Dead-state detection heuristic is efficient, we first set up an experimental group with the configurations -f and -f-m-pr-pa , then we activate the Dead-state detection heuristic by setting up the -f-d and -f-d-m-pr-pa groups. Afterward, we can analyze the impact of the Dead-state detection heuristic by comparing the experiment results of those groups.

1) MUC Extraction: As is shown in Table V and Figure 6, *the MUC-extraction heuristic is quite effective as activating it can bring a boost in the number of solved instances when we run Forward CAR*, e.g. adopting the MUC-extraction heuristic on the basis of -f helps solve 29 more instances and it is the same for -f-d-pr-pa. However, MUC-extraction does not help as much when we run backward CAR. And we also notice that *the MUC-extraction heuristic helps solve the safe instances*, mainly because that MUC is used to refine the over-approximate sequence, which is responsible for proving correctness. And the reason why the MUC-extraction heuristic does not help solve unsafe instances is probably that it takes additional efforts to enumerate literals inside UC and invoke the corresponding SAT call, therefore CAR has less chance to search for a counterexample, and this can also be verified by the fact that in Figure. 6 spots representing jointly solved instances are more likely to be above the diagonal.

TABLE V
IMPACT OF THE MUC-EXTRACTION HEURISTIC.

Configurations	# Solved	# Safe	# Unsafe	Average time (sec)
-f-d-m-pr-pa	356	256	100	1988
-f-d-pr-pa	322	233	89	2120
-f-m	261	183	78	2425
-f	232	151	81	2521
-b-m-pr	355	219	136	1952
-b-pr	351	209	142	1956
-b-m	339	202	137	2035
-b	318	182	136	2106

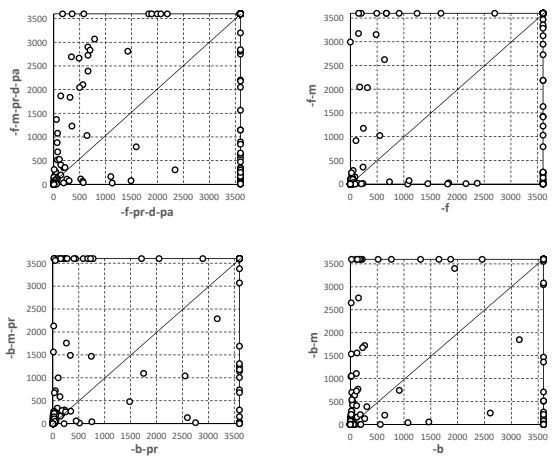


Fig. 6. Comparison among different approaches to evaluate the impact of the MUC-extraction heuristic.

2) UC Propagation: The experiment results are shown in Table VI and Figure 7. It turns out that *the UC-propagation heuristic leads to a boost in the number of solved instances*, for both Forward CAR and Backward CAR. Like the MUC-extraction heuristic, the UC-propagation helps expand the over-approximate sequence and therefore only contributes to solving safe instances. However, UC-propagation does not take as much time as the MUC-extraction heuristic does, since it requires fewer SAT queries.

TABLE VI
IMPACT OF THE UC-PROPAGATION HEURISTIC.

Configurations	# Solved	# Safe	# Unsafe	Average time (sec)
-f-d-m-pr-pa	356	256	100	1988
-f-d-m-pa	325	237	88	2120
-f-pr	272	190	82	2349
-f	232	151	81	2521
-b-m-pr	355	219	136	1952
-b-m	339	202	137	2035
-b-pr	351	209	142	1956
-b	318	182	136	2106

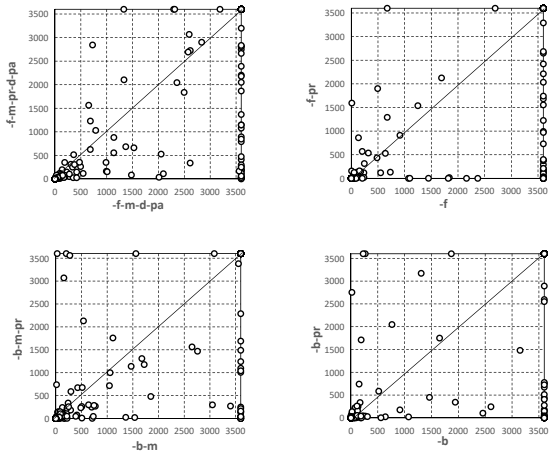


Fig. 7. Comparison among different approaches to evaluate the impact of the UC-propagation heuristic.

3) Dead State Detection: From the experiment results in Table VII, *the Dead-state detection heuristic seems not so efficient*, as there is no obvious increase in the number of solved instances after adopting such a heuristic. Moreover, there is a sharp decrease in the number of solved unsafe instances when we adopt the dead-state detection heuristic on the basis of pure Forward CAR, e.g. -f solves 81 unsafe instances while -f-d only solves 60 unsafe instances. But when we adopt the dead-state detection heuristic on the basis of -f-m-pr-pa, there is an increase of 5 newly solved unsafe instances. We looked into the instances that cannot be solved after adding the dead-state detection heuristic, and found that it failed to solve a large class of benchmarks (from oski). The reason is probably that dead-state detection costs too much on this kind of benchmark and therefore hurts the overall checking performance.

4) Partial State Generation: As is shown in table VIII and Figure 8, *the Partial-state generation heuristic is the most crucial one among the above four heuristics*, as there is a loss

TABLE VII
IMPACT OF THE DEAD-STATE DETECTION HEURISTIC.

Configurations	# Solved	# Safe	# Unsafe	Average time (sec)
-f-d-m-pr-pa	356	256	100	1988
-f-m-pr-pa	351	256	95	1978
-f-d	224	164	60	2563
-f	232	151	81	2521

of 86 instances in the total solved instances when we remove this heuristic from the configuration -f-d-m-pr-pa. Especially, the Partial-state generation heuristic helps solve both safe and unsafe instances, probably because it replaces a single state with a state set and helps refine both the over-approximate and the under-approximate sequences.

TABLE VIII
IMPACT OF THE PARTIAL-STATE GENERATION HEURISTIC.

Configurations	# Solved	# Safe	# Unsafe	Average time (sec)
-f-d-m-pr-pa	356	256	100	1988
-f-d-m-pr	270	210	60	2368
-f-pr	295	205	90	2349
-f	232	151	81	2521

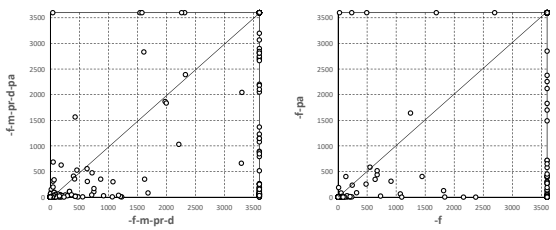


Fig. 8. Comparison among different approaches to evaluate the impact of the Partial-state generation heuristic.

In summary, our experiments show that by equipping the four heuristics proposed in this paper, CAR is able to outperform IC3/PDR by solving 10 more instances, as well as to complement IC3/PDR by uniquely solving 44 instances that IC3/PDR cannot solve. CAR is thus able to contribute to the current model-checking portfolio that provides the best result for practical purposes.

VII. CONCLUSION

In this paper, we present four kinds of heuristics to boost the performance of the CAR model checking algorithm. Our results show that CAR in both the forward and backward directions can outperform IC3/PDR in both directions. We also analyze the pros and cons from the results for readers who are interested to explore a better model-checking performance. Although computing MUC in CAR is not as efficient as computing MIC in IC3/PDR for proof correctness, we propose to combine both directions of CAR to mitigate this drawback as well as keep its previous advantage on bug-finding. This provides a new (potential) direction to explore efficient model-checking techniques other than IC3/PDR.

There are kinds of improvements that can be done in the future. For example, we will improve the implementation of our algorithms from the code level, as based on our experience, the code quality can significantly affect the performance. Also, the MUC heuristic is still a heavy process even though some limits to the procedure have been added. In the future, we need to find better ways to balance the sizes of unsatisfiable cores and the computation cost. More importantly, it is worth investigating the differences between the O-sequences of IC3/PDR and CAR, as they are probably the main reason causing the performance divergence of these two approaches.

ACKNOWLEDGMENT

We thank anonymous reviewers for their helpful comments. Geguang Pu is supported by National Key Research and Development Program (Grant #2020AAA0107800), and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software. Jianwen Li is supported by Shanghai Pujiang Talent Plan (Grant #20PJ1403500) and National Natural Science Foundation of China (Grant #62002118 and #U21B2015).

REFERENCES

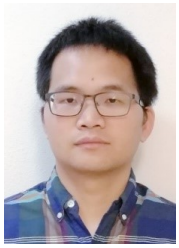
[1] A. Bernardini, W. Ecker, and U. Schlichtmann, "Where formal verification can help in functional safety analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.
[2] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.
[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.
[4] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proceedings of Design Automation Conference (DAC)*, 1999, pp. 317–320.
[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.
[6] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13.
[7] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.
[8] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, pp. 125–134.
[9] J. Li, S. Zhu, Y. Zhang, G. Pu, and M. Y. Vardi, "Safety model checking with complementary approximations," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD '17. IEEE Press, 2017, pp. 95–100.
[10] J. Li, R. Dureja, G. Pu, K. Y. Rozier, and M. Y. Vardi, "Simplecar: An efficient bug-finding tool based on approximate reachability," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 37–44.
[11] K. L. McMillan, "Circular compositional reasoning about liveness," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 342–345.
[12] R. Dureja, J. Li, G. Pu, M. Y. Vardi, and K. Y. Rozier, "Intersection and rotation of assumption literals boosts bug-finding," in *Verified Software. Theories, Tools, and Experiments*, S. Chakraborty and J. A. Navas, Eds. Cham: Springer International Publishing, 2020, pp. 180–192.
[13] "HWMCC 2015," http://fmv.jku.at/hwmcc15/.
[14] "HWMCC 2017," http://fmv.jku.at/hwmcc17/.
[15] E. Clarke and H. Schlingloff, "Model checking," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. MIT Press, 2001, pp. 1635–1790.
[16] O. Kupferman, N. Piterman, and M. Y. Vardi, "An automata-theoretic approach to infinite-state systems," in *Time for Verification: Essays in Memory of Amir Pnueli*, Z. Manna and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 202–259.

[17] K. L. McMillan, *Symbolic Model Checking*. Boston, MA: Springer US, 1993.
[18] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158.
[19] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.
[20] A. Griggio and M. Roveri, "Comparing different variants of the IC3 algorithm for hardware model checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 1026–1039, 2016.
[21] H.-J. Kang and I.-C. Park, "SAT-based unbounded symbolic model checking," pp. 840–843, 2003.
[22] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, ser. CAV '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 250–264.
[23] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 260–276.
[24] H. G. Vediramana Krishnan, Y. Vizel, V. Ganesh, and A. Gurfinkel, "Interpolating strong induction," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 367–385.
[25] A. Ivrii and A. Gurfinkel, "Pushing to the top," in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '15. Austin, Texas: FMCAD Inc, 2015, pp. 65–72.
[26] R. Dureja, A. Gurfinkel, A. Ivrii, and Y. Vizel, "Ic3 with internal signals," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 63–71.
[27] H. Zhang, A. Gupta, and S. Malik, "Syntax-guided synthesis for lemma generation in hardware model checking," in *Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 325–349.
[28] T. Seufert, C. Scholl, A. Chandrasekharan, S. Reimer, and T. Welp, "Making progress in property directed reachability," in *Verification, Model Checking, and Abstract Interpretation*, B. Finkbeiner and T. Wies, Eds. Cham: Springer International Publishing, 2022, pp. 355–377.
[29] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.
[30] "Minisat 2.2.0," https://github.com/niklasso/minisat.
[31] A. Belov, I. Lynce, and J. Marques-Silva, "Towards efficient MUS extraction," *AI Commun.*, vol. 25, no. 2, pp. 97–116, apr 2012.
[32] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '10. Austin, Texas: FMCAD Inc, 2010, pp. 221–229.
[33] A. Biere, "AIGER Format," http://fmv.jku.at/aiger/FORMAT.
[34] "Artifacts," https://drive.google.com/file/d/14kICTcwaukHcJY14lEqSUhwsNshHhXco/view?usp=sharing.
[35] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.
[36] "iimc," https://github.com/mgudemann/iimc.
[37] "IC3Ref," https://github.com/arbrad/IC3ref.
[38] "SimpleCAR," https://github.com/lijwen2748/simplecar/releases/tag/v0.1.

Xiaoyu Zhang received the B.S. degree from the School of Mathematical Sciences, East China Normal University, in 2018. He is pursuing the Ph.D. degree with the Software Engineering Institute, East China Normal University. His research interests include temporal logic and model checking.

Shengping Xiao received the B.S. degree from the Software Engineering Institute, East China Normal University, in 2021. He is currently working toward the Ph.D. degree in the Software Engineering Institute, East China Normal University. His research interests include temporal logic and model checking.

Yechuan Xia received the B.S. degree from the School of Computer Science and Technology, Donghua University, in 2019. He is currently working toward the Ph.D. degree in the Software Engineering Institute, East China Normal University. His research interests include model checking, formal methods and requirement engineering.



Jianwen Li received his Ph.D. from the Software Engineering Institute, East China Normal University, in 2014. He is now a research professor with the Software Engineering Institute, East China Normal University. His research interests includes formal verification, logic and automata theory.

Mingsong Chen (Senior Member, IEEE) received the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2010. He is currently a Professor with the Software Engineering Institute at East China Normal University. His research interests are in the area of design automation of cyber-physical systems, EDA, embedded systems, and formal verification techniques. Currently he serves as the director of Engineering Research Center of Software/Hardware Co-design Technology and Application affiliated to the Ministry of Education, China, and the vice director of technical committee of embedded systems of China Computer Federation (CCF). He is an Associate Editor of IET Computers Digital Techniques, and Journal of Circuits, Systems and Computers.

Geguang Pu received the B.S. degree in mathematics from Wuhan University in 2000 and the Ph.D. degree in mathematics from Peking University in 2005. He is currently a Professor with the Software Engineering Institute, East China Normal University. He has published over 100 publications on the topics of software engineering and system verification, including ICSE, FSE, ASE, and CAV. His research interests include program testing and reliable AI systems. He served as a PC member for more than 20 international conference committees.