# Revisiting Assumptions Ordering in CAR-Based Model Checking

Yibo Dong[†], Yu Chen[§], Jianwen Li [‡], Geguang Pu[‡], Ofer Strichman[¶]
[†]Each China Normal University, China, prodongf@gmail.com
[‡]Each China Normal University, China, {jwli,ggpu}@sei.ecnu.edu.cn
[§]Chuzhou University, China, chenyu@chzu.edu.cn
[¶]Technion, Isreal, ofers@technion.ac.il

***Abstract*—Model checking is an automatic formal verification technique that is widely used in hardware verification. The state-of-the-art complete model-checking techniques, based on IC3/PDR and its general variant CAR, are based on computing symbolically sets of under - and over-approximating state sets (called 'frames') with multiple calls to a SAT solver. The performance of those techniques is sensitive to the order of the assumptions with which the SAT solver is invoked, because it affects the *unsatisfiable cores* — which the solver emits when the formula is unsatisfiable — that crucially affect the search process. This observation was previously published in [15], where two partial assumption ordering strategies, *intersection* and *rotation* were suggested (partial in the sense that they determine the order of only a subset of the literals). In this paper we extend and improve these strategies based on an analysis of the reason for their effectiveness. We prove that intersection is effective because of what we call *locality* of the cores, and our improved strategy is based on this observation. We conclude our paper with an extensive empirical evaluation of the various ordering techniques. One of our strategies, Hybrid-CAR, which switches between strategies at runtime, not only outperforms other, fixed ordering strategies, but also outperforms other state-of-the-art bug-finding algorithms such as** ABC-BMC.

***Index Terms*—Hardware Verification, Formal Verification, Model Checking, Complementary Approximate Reachability, CAR**

## I. Introduction

*Model checking* is an automatic formal verification technique that is central in the hardware design community [3], [23]. Given a model $M$ and a temporal property $P$ over its variables, it checks whether all the behaviours of $M$ satisfy $P$, i.e., whether $M \models P$. Once a system behaviour is detected to violate $P$, the model checker returns a *counterexample* as the evidence, which demonstrates the execution of the system leading to the property violation. Such a process is called *bug-finding*. If $P$ is a *safety* property, the violation of $P$ is witnessed by a counterexample made of a finite number of states. It is well known that model checking on safety properties can be reduced to reachability analysis [12].

State-of-the-art safety model checking techniques include Bounded Model Checking (BMC) [4], [6], Interpolation Model Checking (IMC) [27], Property Directed Reachability (PDR) (also called IC3) [8], [16], and Complementary Approximate Reachability (CAR) [26], all of which integrate a SAT solver internally. BMC is an incomplete method (it is only used for finding bugs, not proving their absence) and is empirically very fast at finding relatively shallow bugs (i.e., after a relatively small number of steps from the initial state). IMC, PDR and CAR are complete but are generally not as fast as BMC in shallow bug-finding, and none of the existing implementations of those techniques dominates the other. In [24], [26], it was empirically shown that within a given time and hardware resources, CAR is able to solve unsafe instances that BMC cannot, and safety instances that IMC and PDR cannot, while the converse is true as well. Therefore, a portfolio consisting of different techniques is often maintained for different verification tasks. However, model-checking (a PSPACE problem), always falls short of the performance needs in the industry when it comes to verifying large designs [14], [34]. Indeed, performance optimisation of SAT-based model-checkers is an active research area. Some recent examples are [38], [37], [30] and [32].

In this paper, we focus on improving the performance of CAR. We will describe in detail how CAR works in Section II-D. It has many similarities to PDR, which is better known, but also several distinctive features. For now, let us just mention that, similar to PDR, it relies on many SAT calls over relatively easy formulas. One of its elements is a sequence of formulas $O_1 \dots O_k$, called the over-approximating frames (or *O-frames*, for short), where $O_i$, $1 \leq i \leq k$, over-approximates the states that can reach $\neg P$ within $i$ steps. CAR gradually makes these frames more precise, i.e., less over-approximating, by removing from them states that cannot reach $\neg P$ within the given number of steps. One of the critical elements of this process is *generalisation*, that is, the ability to remove many such states at once. This is done by finding the *unsatisfiable core* (UC) of unsatisfiable SAT calls. The research we report here focuses on improving the quality of those UCs, which accelerates the narrowing process of the $O$-frames. To explain our contributions, let us briefly recall how modern SAT solvers find UCs.

The input to every SAT call in CAR (and PDR) takes the form of $\bigwedge_{l \in \mathcal{A}} l \wedge \phi$, where $\phi$ is a Boolean formula in Conjunctive Normal Form (CNF) and $\mathcal{A}$ consists of a sequence of literals, called the *assumptions*. Almost all modern CDCL-based SAT solvers as of MINISAT [17] and GLUCOSE [1], [2] support assumptions. They position the literals in $\mathcal{A}$, in order, as their first decisions, and perform Boolean Constraint Propagation (BCP) as usual. Unsatisfiability is detected when the BCP of an assumption contradicts the value of another literal (because recall, all the literals in $\mathcal{A}$ and those that are implied by them via BCP are implied by the formula regardless of any decision). By analysing the trail, the solver can detect which of the assumptions contributed to the conflict and emit this list of assumptions as the UC, which is essentially a compact *reason* for the unsatisfiability. In other words, the UC is a subset of $\mathcal{A}$ that is sufficient for making $\phi$ unsatisfiable.

There can be multiple UCs in a given unsatisfiable formula, and the order of the assumptions may affect the UC that is found (and this, in turn, affects the overall performance of the model-checker, whether it is CAR or PDR). More explicitly, the literals that are propagated earlier are more likely to appear in the returned UC. Indeed, prior work leveraged this phenomenon to improve performance. Specifically, the IC3ref model checker [21], which implements the original IC3 algorithm, sorts the literals in $\mathcal{A}$ in descending order based on their appearance frequencies. The SIMPLECAR model-checker [25], which implements CAR, uses two different literal-ordering strategies, as reported in [15]: *Intersection*, which prioritizes literals that are both in the current state and the latest generated UC, and *Rotation*, which prioritizes literals that are present in all previously explored states. This makes these literals more likely to appear as part of the generated UC, and empirically improves bug-finding performance. Indeed, SIMPLECAR is one of the baseline implementations against which we compare our contributions. In addition, we compare ourselves against the best known BMC implementation, as well as previous optimizations that were applied to CAR [38].

Our contributions are:

1) We revisit one of the two heuristics proposed in [15], called *Intersection*, and suggest an explanation for its effectiveness. Briefly, we show that it leads to finding proofs of unsatisfiability faster because of what we call the *locality* of the cores. Based on this observation, we propose an extension of this technique that improves locality, and decides on the order of more literals comparing to the original version of *Intersection*. We also show how it affects a combination of *locality* with another strategy from [15] called *Rotation*. Our experimental results indicate that this leads to faster convergence and increases the number of cases that can be solved within a given timeout.
2) We define the *conflict literal* as the last literal found to be in the UC by the SAT solver, and observe that unlike the other literals in the core, it is *necessary* (without it, the other literals do not form a core). We show that by prioritizing these literals, proofs are found faster.
3) We suggest a method called Hybrid-CAR, which swaps *CoreLocality*'s configurations during the search (based on giving a time-limit to each configuration), which not only outperforms the previous 'static' ordering approaches but also any other bug-finding techniques, including ABC-BMC, the state-of-the-art BMC implementation [9] and previous published versions of CAR.
4) We provide an extensive empirical study about the influence of assumption ordering on the UC generated, with the existing and new strategies, thereby empirically demonstrating the significance of literal ordering in CAR-based model checkers.

We continue with preliminaries in the next section. In Section III we describe the prior work of [15] and explain why literal ordering matters. Section IV describes our contribution, and Section V describes the results of our empirical evaluation. Our conclusions are summarised in Section VI.

## II. 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 $s_1 \cup s'_2 \models T$, which is also denoted by $(s_1, s_2) \in T$. A *path* of length $k$ is a finite state sequence $s_1, s_2, \dots, s_k$, where $(s_i, s_{i+1}) \in T$ holds for $(1 \le i \le k-1)$. 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 $i \ge 0$, 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 Model 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 condition in the safe/unsafe checking is satisfied, the search process terminates. Intuitively, unsafe checking $F_i \cap \neg P \neq \emptyset$ indicates that some bad states are within $F_i$ and safe checking

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 | $F_{i+1} \subseteq \bigcup_{0 \le j \le i} F_j$ | $B_{i+1} \subseteq \bigcup_{0 \le j \le i} B_j$ |
| Unsafe Check | $F_i \cap \neg P \neq \emptyset$ | $B_i \cap I \neq \emptyset$ |

$F_{i+1} \subseteq \bigcup_{0 \le j \le i} F_j$ indicates that all the reachable states from $I$ have 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.

### C. SAT Solving and Unsatisfiable Cores

In propositional logic, a *literal* is an atomic variable or its negation. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals. 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 $\phi$ as a set of clauses. 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 $\phi$ is satisfiable if there exists an assignment of each Boolean variable in $\phi$ such that $\phi$ is true; otherwise, $\phi$ is unsatisfiable. A SAT solver can decide whether a CNF formula $\phi$ is satisfiable or not. It emits a Boolean assignment to the variables, called a model of $\phi$, if $\phi$ is satisfiable. Otherwise, it emits an unsatisfiable core as explained in the introduction, based on a subset of the assumptions.

### D. Complementary Approximate Reachability (CAR)

CAR is a relatively new SAT-based safety model checking approach that is essentially a reachability-analysis algorithm, inspired by PDR [26]. Unlike BMC [4], [6], CAR is complete, i.e., it can also prove correctness. CAR maintains two sequences of state sets (also called 'frames'), that are defined as follows:

**Definition 1** (Over/Under Approximating State Sequences)**.** *Given a transition system $Sys = (V, I, T)$ and a safety property $P$, the over-approximating state sequence $O \equiv O_0, O_1, \dots, O_i$ $(i \geq 0)$, and the under-approximating state sequence $U \equiv U_0, U_1, \dots, U_j$ $(j \geq 0)$ are finite sequences of state sets such that, for $k \geq 0$:*

| | $O$-sequence | $U$-sequence |
|---|---|---|
| Base: | $O_0 = \neg P$ | $U_0 = I$ |
| Induction: | $O_{k+1} \supseteq R^{-1}(O_k)$ | $U_{k+1} \subseteq R(U_k)$ |
| Constraint: | $O_k \cap I = \emptyset$ | $--$ |

*These sequences determine the termination of CAR as follows:*

- *Return 'Unsafe' if $\exists i \cdot U_i \cap \neg P \neq \emptyset$.*
- *Return 'Safe' if $\exists i \geq 1 \cdot (\bigcup_{j=0}^{i} O_j) \supseteq O_{i+1}$.*

Notably, CAR can also use the over and under approximating sequences reversed, i.e., use the over-approximating sequence in the forward direction, from the initial state towards the negated property, while using the under-approximating sequence from the negated property towards the initial state. In this paper, we only consider the direction as stated in Definition 1 (this was called 'backward CAR' in [24], [26]).

At the high level, CAR can be considered a general version of PDR, as the O-sequence in CAR is not necessarily monotone, while that in PDR is. As a result, CAR can have a more flexible methodology for the *state generalization*, i.e., directly using the UC from the SAT solver rather than computing the *relative inductive clauses*. However, CAR needs to invoke additional SAT queries to find the invariant (checking safety), while PDR can do it with a simple syntactic check.

**Algorithm 1:** Complementary Approximate Reachability (CAR).

```
Input: A transition system $Sys = (V, I, T)$ and a
       safety property $P$
Output: 'Safe' or ('Unsafe' + a counterexample)
1  if $SAT(I \wedge \neg P)$ then return 'Unsafe'
2  $U_0 := I$, $O_0 := \neg P$
3  while true do
4      $O_{tmp} := \neg I$
5      while $state := pickState(U)$ is successful do
6          $stack := \emptyset$
7          $stack.push(state, |O| - 1)$
8          while $|stack| \neq 0$ do
9              $(s, l) := stack.top()$   // Assume $s \in U_j$
10             if $l < 0$ then return 'Unsafe'
11             $\hat{s}$ = Reorder (s, l + 1)
12             if $SAT(\hat{s}, T \wedge O'_l)$ then
13                 $t := GetModel()|_{V'}$
14                 $U_{j+1} := U_{j+1} \cup t$   // Widening $U$
15                 $stack.push(t, l - 1)$
16             else
17                 $stack.pop()$
18                 $uc := getUC()$
19                 if $l + 1 < |O|$ then
                       $O_{l+1} := O_{l+1} \wedge (\neg uc)$
20                 else $O_{tmp} := O_{tmp} \wedge (\neg uc)$
21                 while $l + 1 < |O|$ and $\neg s \in O_{l+1}$ do
                       $l := l + 1$
22                 if $l + 1 < |O|$ then $stack.push(s, l)$
23     if $\exists i \geq 1$ s.t. $(\bigcup_{0 \le j \le i} O_j) \supseteq O_{i+1}$ then return
         'Safe'
24     Add a new state-set to $O$ and initialize it to $O_{tmp}$
```

Algorithm 1 describes CAR. It progresses by widening the $U$ sets, and narrowing the $O$ sets, which are initialised at Line 2 to $I$ and $\neg P$, respectively. The algorithm maintains a stack of pairs $\langle state, level \rangle$ where $level$ refers to an index of an $O$ frame. $O_{tmp}$, initialised to $\neg I$ in Line 4 and later updated, represents the next frame to be created.

Initially, a state from the $U$-sequence is heuristically picked (Line 5) – by default from the end to the beginning – and pushed to the stack. In each iteration of the internal loop, CAR checks whether the state at the top of the stack, call it $s$, can transition to the $O_l$ frame. This is done by checking if $s \wedge T \wedge {O_l}'$ is satisfiable (Line 12, $\hat{s}$ is exactly $s$ if literal-ordering is not invoked, otherwise a reordered version). If yes, a new state $t \in O_l$ is extracted from the model to update the $U$-sequence (Line 13-15), effectively *widening* it; Otherwise, the negation of the unsatisfiable core is used to constrain the $O$ frame of $s$ (level $l+1$), effectively *narrowing* it (Lines 17-19), and pushing $s$ back to the stack. In Line 21, CAR skips frames that already block $s$.

CAR returns 'Unsafe' as soon as the working level $l$ is less than 0, which indicates that a bad state in $\neg P$ is reached (line 10). Otherwise, CAR returns 'Safe' if the $O$ sequence includes all the states that can reach $\neg P$ – this is checked via the condition in Line 23, which was also mentioned as part of Definition 1.

## III. Literal Reordering Strategies: prior work and insights

**Example III.1** (Prior literals are more likely to appear in the UC)**.** *Let the clauses and assumptions be*

$$\begin{aligned} \phi &\doteq \{(a_1 \vee \neg a_4 \vee \neg a_5), (a_3 \vee \neg a_4 \vee \neg a_5), (a_2 \vee a_4)\} \\ \mathcal{A} &\doteq (\neg a_1, a_2, \neg a_3, a_4, a_5)\ . \end{aligned}$$

*Different literal orderings generate different UCs:*

**Order 1:** $Assum_1 = (\neg a_1, a_2, a_4, a_5, \neg a_3)$

$$\begin{aligned} BCP(\neg a_1) :&\{(False \vee \neg a_4 \vee \neg a_5), \\ &(a_3 \vee \neg a_4 \vee \neg a_5), (a_2 \vee a_4)\} \\ BCP(a_2) :&\{(False \vee \neg a_4 \vee \neg a_5), \\ &(a_3 \vee \neg a_4 \vee \neg a_5), (True \vee a_4)\} \\ BCP(a_4) :&\{(False \vee False \vee \neg a_5), \\ &(a_3 \vee False \vee \neg a_5), (True \vee a_4)\} \\ BCP(a_5) :&\{(False \vee False \vee False), \\ &(a_3 \vee False \vee False), (True \vee a_4)\} \\ \longrightarrow UC_1 &= (\neg a_1, a_4, a_5). \end{aligned}$$

**Order 2:** $Assum_2 = (a_5, a_4, \neg a_3, a_2, \neg a_1)$

$$\begin{aligned} BCP(a_5) :&\{(a_1 \vee \neg a_4 \vee False), \\ &(a_3 \vee \neg a_4 \vee False), (a_2 \vee a_4)\} \\ BCP(a_4) :&\{(a_1 \vee False \vee False), \\ &(a_3 \vee False \vee False), (a_2 \vee a_4)\} \\ BCP(\neg a_3) :&\{(a_1 \vee False \vee False), \\ &(False \vee False \vee False), (a_2 \vee a_4)\} \\ \longrightarrow UC_2 &= (a_5, a_4, \neg a_3). \end{aligned}$$

As mentioned in the introduction, modern CDCL-based SAT solvers such as the derivatives of MINISAT [17] take as input, in addition to the formula $\phi$, a vector of literals $\mathcal{A}$, called the assumptions, and checks whether $\bigwedge_{l \in \mathcal{A}} l \wedge \phi$ is satisfiable.

The SAT solver chooses the assumption literals to be the first decisions in the order they are given. As usual, after each such decision, it invokes BCP. Suppose there is already a conflict in the first $|A|$ ($A \subseteq \mathcal{A}$) decision levels (recall that this can happen after learning and backtracking to those levels). In that case, the search is terminated – the formula is declared unsatisfiable under $A$. The solver can be asked to analyse the cause of the conflict and return it in the form of a subset of $A$. This implies that assumption literals after $|A|$ in the predefined order cannot be part of the generated UC. As a result, prior assumption literals have a higher probability of appearing in the UC. That is why literal ordering matters. Example III.1 illustrates this point.

**The Intersection strategy and locality of cores:** The *Intersection* strategy [15] places the intersection with the last UC in the beginning of the assumptions sequence – see Algorithm 2. It is an implementation of the *Reorder* function that is called in line 11 of Algorithm 1 with level $l+1$, namely the previous level. In line 5 of Algorithm 2, the literals from this UC are placed first in the order, which makes them more likely to appear in the new core, hence make consecutive cores similar. This is what we call 'the *locality* of the cores'.

The term *locality* is used, among other places, in describing decision heuristics in SAT solving. All CDCL solvers use decision heuristics that prioritize variables that participated in recent conflicts, hence they focus the search. Although this is not directly related to the current paper, our hypothesis is that this decision strategy is effective because it generates proofs faster: similar clauses are necessary for constructing a resolution proof (for satisfiable cases, learning has little effect to begin with [29]). And if there is a small core, it is better to focus the search and hopefully find it rather than generating unrelated clauses.

Our argument is that finding cores in CAR that are similar should have a similar effect: it makes proofs involving the $O$ frames easier and hence faster. In other words, every time that we check whether a state can reach an $O$ frame, if that frame contains apriori many of the clauses that are necessary for the proof that the state is not reachable, the proof will converge faster. We tested this hypothesis empirically, and the results appear in Table II. While the first row shows the effect of locality on the average run time of UNSAT cases, the second row is the average overall time for proving that a state cannot reach an $O$ frame, i.e., the average time of an iteration of the loop in line 5 of Algorithm 1. The evaluation is based on benchmarks from the single safety property track of the 2015 [18] and 2017 [19] Hardware Model Checking Competition (HWMCC [7])[1].

**The Rotation strategy:** The *Rotation* technique — demonstrated in Algorithm 3 — maintains a vector *common* for each level $l$ to track the similarity among recent failed states, i.e.,

[1]The experiment setup in this section is the same as that in Sec. V.

TABLE II
THE INTERSECTION STRATEGY, THROUGH WHAT WE CALL LOCALITY, ACCELERATES UNSAT CALLS AND PROOF FINDING

| Strategy | Natural | Intersection |
|---|---|---|
| Average time of UNSAT calls(s) | 0.0132 | 0.0105 |
| Average time of finding proofs(s) | 0.9541 | 0.6287 |

**Algorithm 2:** Reordering algorithm: *Intersection*

```
Input: A vector of literals $s$ representing a state, and
       the frame level $l$
Output: $\hat{s}$: the reordered $s$
1  $\hat{s} = \emptyset$;
2  $lastUC \coloneqq getLastUC(l)$;        ▷ A vector of literals
3  for each $lit \in lastUC$ do
4      if $lit \in s$ then
5          $\hat{s}.pushBack(lit)$
6  for each $lit \in s \wedge lit \notin \hat{s}$ do
7      $\hat{s}.pushBack(lit)$
8  return $\hat{s}$
```

the *common* vector is a reordered version of the last failed state, with the intersection of the failed states in the front.

The key insight behind *Rotation* [15] is that in cases where the solver consistently returns similar states that share common literals but fail to explore deeper levels, the search process may be trapped within a specific sub-space. Consequently, Rotation prioritises the common part in the front, intending to generate a UC from it, thus facilitating an exit from the problematic sub-space.

**Algorithm 3:** Reordering algorithm: Rotation

```
Input: A vector of literals $s$ representing a state, and
       the frame level $l$
Output: $\hat{s}$: the reordered $s$
1  $\hat{s} = \emptyset$;
2  $cVec \coloneqq getCommonVector(l)$; ▷ get common vector
3  for each $lit \in cVec$ do
4      if $lit \in s$ then
5          $\hat{s}.pushBack(lit)$
6  for each $lit \in s \wedge lit \notin \hat{s}$ do
7      $\hat{s}.pushBack(lit)$
8  return $\hat{s}$
9  // Future updates:
10 if $\neg SAT(\hat{s}, T \wedge O'_l)$        ▷ Fail to reach
11  then
12     $setCommonVector(l, \hat{s})$ ▷ Update common vector
```

While maintaining the consecutive intersection of all the recent failed states is feasible, the length of the joint part decreases, consequently containing diminishing information. As a remedy, it was suggested in [15] to preserve a fixed length of the *common* vector. As demonstrated in Fig. 1, *common* always preserves the last failed state – and is reordered to encode the historical information. While the first segment can be regarded as the intersection of all states ($\bigcap_{s_1}^{s_n}$, as shown in the figure), the combination of the first two segments can be seen as all the states except one ($\bigcap_{s_2}^{s_n}$), and so forth.

The rational behind *Rotation*, is that it encourages finding UCs made of literals that appeared in many recent failed states, hence it diverts the search from areas that seem to lead nowhere. This, in turn, should reduce the number of states that are checked and the corresponding number of SAT calls. Indeed, our results in Table III show a reduction in the number of SAT calls with *Rotation* when proving that a state from a $U$ frame cannot reach any of the $O$ frames (i.e., a single iteration of the loop starting in line 5 of Algorithm 1). The basis of the evaluation is the same as in Table II.

TABLE III
THE ROTATION STRATEGY HELPS ESCAPING BAD AREAS OF THE SEARCH, AND CONSEQUENTLY IT LOWERS THE NUMBER OF SAT QUERIES AND THE TIME OF PROOFS.

| Strategy: | Natural | Rotation |
|---|---|---|
| Average #SAT calls to find proofs | 207.13 | 190.25 |
| Average time to find proofs (s) | 0.9541 | 0.7277 |

TABLE IV
THE COMBINATION OF INTERSECTION AND ROTATION CAN FIND PROOFS FASTER.

| Strategy: | Natural | Combination (I+R) |
|---|---|---|
| Average time of UNSAT calls(s) | 0.0132 | 0.0137 |
| Average #(SAT Query) to find proof | 207.13 | 173.57 |
| Average time of finding proofs(s) | 0.9541 | 0.5990 |

**Combining *Intersection* and *Rotation*:** When it comes to combining these two algorithms, it should be noted that the latest UC selected in *Intersection* is derived from the last failed state. This observation leads to the conclusion that the $iCube$ (the cube generated via *Intersection*) is a subset of the $rCube$ (the cube generated via *Rotation*). As shown in Fig.2, to integrate the two algorithms is merely to position the literals produced by *Intersection* ahead of those generated by *Rotation* while eliminating duplicate literals in the latter.

The results in Table IV show that indeed the combination finds proofs faster on average.

## IV. LITERAL ORDERING STRATEGIES: NEW APPROACHES

As discussed in the previous section, *Intersection* and *Rotation*, either separately or combined, determine the position of only a limited portion of the whole literal set, whereas the position of other literals is determined by what we called the 'natural' order, which is arbitrary.

A motivating example is depicted in Fig.1. As shown, the segment $S_3 - S_2$ within $\hat{S_3}$ remains unaffected by both literal ordering strategies, i.e., it just remains the natural order, yet it constitutes more than half of the state's length.

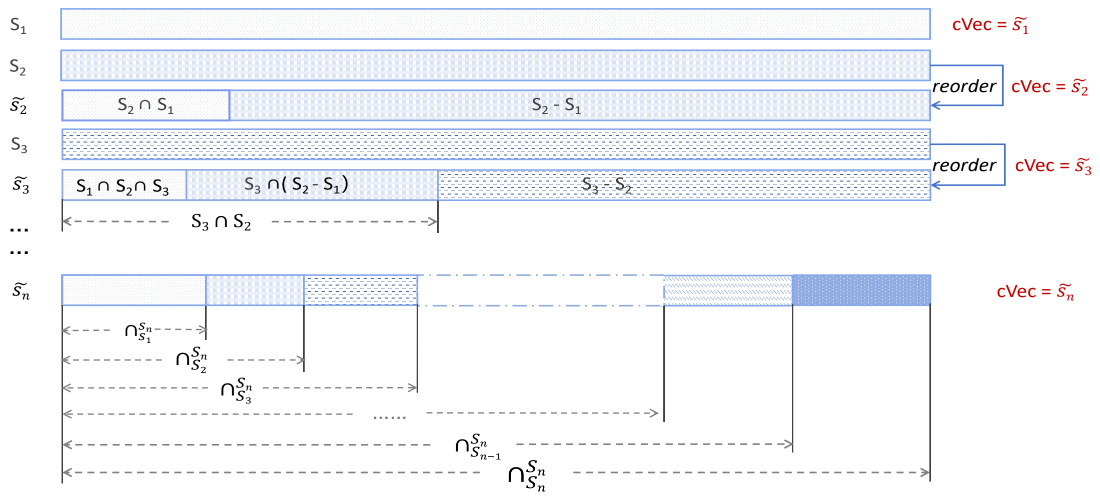
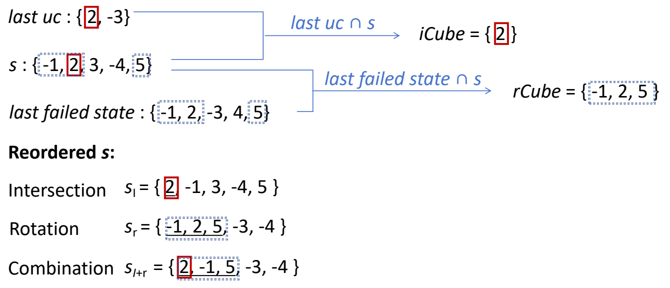


Fig. 1. The updating process of $cVec$, which is the core of *Rotation*. The consecutive intersection of recent failed states ($\bigcap s_i$) is in the very beginning.



last uc : {2, -3}
s : {-1, 2, 3, -4, 5}
last failed state : {-1, 2, -3, 4, 5}
last uc ∩ s
last failed state ∩ s
iCube = {2}
rCube = {-1, 2, 5}
Reordered s:
Intersection $s_I$ = {2, -1, 3, -4, 5}
Rotation $s_r$ = {-1, 2, 5, -3, -4}
Combination $s_{I+r}$ = {2, -1, 5, -3, -4}

Fig. 2. An example of the reordering process, where $s_i$, $s_r$ and $s_{i+r}$ each represents the reordered state $\hat{s}$ using only *Intersection*, only *Rotation* and their combination, respectively.

**Algorithm 4:** Reordering algorithm: *CoreLocality*

```
Input: A state $s$, frame level $l$, configuration $iLimit$
Output: $\hat{s}$: The reordered $s$
1  $\hat{s} := \emptyset$
2  for $k : 0 \to iLimit$ do
3      Let $ref_k = getTheLast\_kth\_UC(l)$
4      if $ref_k \neq \emptyset$ then
5          for each $lit \in ref_k$ do
6              if $lit \in s \wedge lit \notin \hat{s}$ then
7                  $\hat{s}.pushBack(lit)$
8                  ▷ Literals added here form the $iCubes$
9  $cVec := getCommonVector(l)$
10 for each $lit \in cVec$ do
11     if $lit \in s \wedge lit \notin \hat{s}$ then
12         $\hat{s}.pushBack(lit)$
13             ▷ Literals added here form the $rCube$
14 for each $l \in s \wedge l \notin \hat{s}$ do
15     $\hat{s}.pushBack(l)$
16 return $\hat{s}$
```

In this section we will show a way to increase the portion of literals that their position is determined, utilising more historical information on the cores, and consequently improving the overall runtime.

## A. Literal reordering with CoreLocality

Stemming from the intuition that incorporating recent UCs beyond the most recent one could help by improving locality, we propose a new literal ordering strategy *CoreLocality*, which is outlined in Algorithm 4. By expanding the scope of considered UCs, intersecting with each and organizing the results chronologically (with the intersection with newer UCs placed earlier), *CoreLocality* facilitates sorting a greater number of literals, thereby refining the guidance of the search.

As shown in the algorithm, in addition to the state $s$ and frame level $l$, a new parameter $iLimit$ is introduced to denote the limit on the amount of UCs to utilise. The *for* block at Line 2-8 computes the intersection according to the corresponding UC, and pushes them into $\hat{s}$ in order. For the *if* block at Line 11-13, it is similar to *Rotation* .

**Example IV.1.** *Fig. 3 illustrates a computational process for the* CoreLocality *strategy with several different $iLimit$ values. The upper dashed box in the figure shows the last 3 UCs in chronological order (1st being the most recent one), along with the last failed state, and the current state s. Next, $iCubes$ and $rCube$ are computed based on the above data, similar to the calculation in Fig. 2, as shown in the lower left dashed box. Finally, in the lower right dashed box, s is reordered by $iCubes$ and $rCube$ based on different choices of $iLimit$. As is shown, by incorporating the 2nd UC, the literal '-4' is successfully impacted.* □

The distinction between *CoreLocality* and *Intersection* is demonstrated in Fig. 4). It prioritises literals that would otherwise be relegated to the rear of $rCube$, or even after $rCube$.

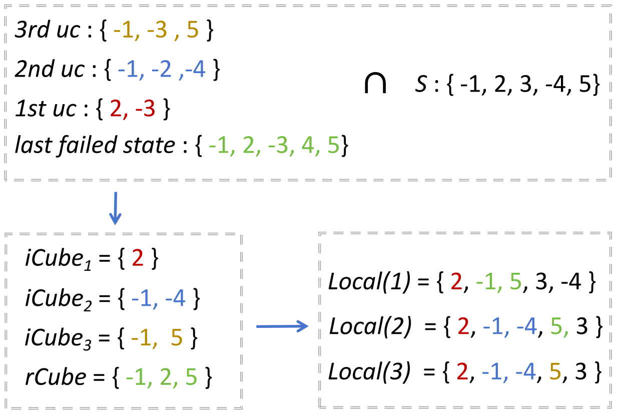


Fig. 3. An example of the *CoreLocality* strategy. $Local(k)$ denotes reordered state utilizing $k$ UCs, i.e., $iLimit = k$.

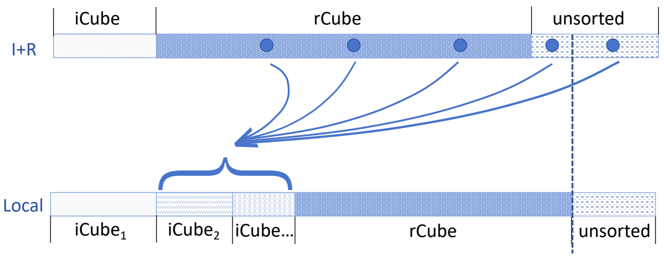


Fig. 4. The difference between *CoreLocality* and combination of *Intersection* and *Rotation*. Some literals (blue dots) from $rCube$ and $unsorted$ are prioritised.

**Tuning *CoreLocality*:** In the *CoreLocality* strategy, the parameter $iLimit$, which denotes the maximum number of utilised UCs, serves as a metric of the 'local' scope, defining the range within which a UC is considered 'recent'. In other words, given that the relevance of a UC to the current query diminishes as it becomes more distant, setting a limit excludes prior outdated UCs from current consideration. While increasing the value of $iLimit$ allows for the inclusion of additional information, it also diminishes the impact of *Rotation* due to the precedence of $iCubes$ over the $rCube$. Furthermore, while it is feasible to set the $iLimit$ large enough to order all the literals, this approach is observed to be highly inefficient. The considerable increase in cost to get one more literal reordered, i.e., one that appears in a subsequent UC, but not in any previous one, often necessitates thousands or even tens of thousands of UCs. This phenomenon is demonstrated in Fig.5 for a particular formula.

The optimal value of $iLimit$ depends, of course, on the specific problem context and constitutes a trade-off between literal coverage and the computational cost to achieve it. Indeed, the results in Table V (middle column) show that the speed to find proofs of *CoreLocality* depends on $iLimit$ but as expected, it is not monotonic. They also show that with these low values of $iLimit$ we are able to find proofs faster than the previous methods. In practice the best $iLimit$ value can be found based on experiments, but there is also an option to change it during run time, as we will explain later.

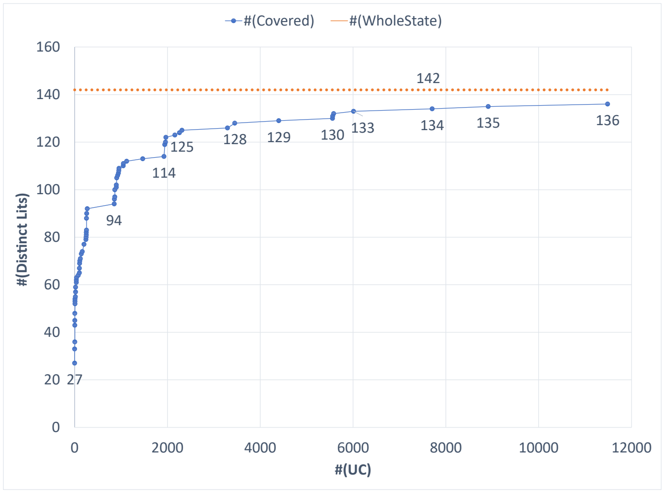


Fig. 5. Increasing the number of UCs that are intersected has a diminishing effect on the number of distinct literals that are covered (this figure was created based on a formula from 6s33.aig).

TABLE V
CORELOCALITY CAN FIND PROOFS EVEN FASTER. 'WITH' AND 'WITHOUT' REFERS TO THE 'CONFLICT LITERAL' OPTIMIZATION.

| Strategy | Average time of finding proofs(s) | |
|---|---|---|
| | Without | With |
| Natural | 0.9478 | 0.9541 |
| Combination(I+R) | 0.7728 | 0.5990 |
| Local(2) | 0.6191 | 0.5887 |
| Local(3) | 0.6476 | 0.5566 |
| Local(4) | 0.5721 | 0.4836 |
| Local(5) | 0.6413 | 0.5078 |

## B. Moving forward the conflict literals

Not all literals in a UC are equal. Specifically, the last literal added to the core, by definition, was *necessary* for that proof (in other words, it is part of a *minimal* core). We call it the *conflict* literal. During the process of getting UCs, we give such conflict literals a higher priority by placing them in the front of the core. For example, suppose that $iCube = (1, 2, 3)$ and literal 3 is the conflict literal of this clause. Then we re-order $iCube$ to $iCube = (3, 1, 2)$, before we proceed with building the assumption literal order as described earlier (Algorithm 4). As shown in the right column of Table. V, this small optimization accelerates, on average, the process of finding proofs.

From here on, when we say *CoreLocality*, we mean *CoreLocality* together with this optimization.

## C. Hybrid-CAR: Combining different orderings

Empirically, the best configuration of *CoreLocality* varies according to the specific problem and is hard to predict. It is often observed that a model checking problem that can be solved easily with one literal ordering strategy will timeout with another. This encouraged us to research a dynamic strategy, by which the configuration is periodically switched.

TABLE VI
THE U-SEQUENCE EXPANDS QUICKLY AND SLOWS DOWN THE ITERATION TIME IN 6S33.AIG

| Round | TimeForThisRound(s) | size(U) |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0 | 2 |
| 3 | 0.011 | 15 |
| 4 | 0.007 | 20 |
| 5 | 0.763 | 823 |
| 6 | 7.392 | 5625 |
| 7 | 12.264 | 9632 |
| 8 | 43.158 | 19285 |
| 9 | 356.92 | 57263 |
| 10 | >3000(Timeout) | 110235 |

We coupled this direction with a new restart mechanism for CAR. The U-sequence in CAR is observed to expand quickly, resulting in increasingly longer time to extend a new $O$ frame.

Taking a closer look at the particular case shown in Table VI, it can finish the first five rounds within 1 second when the size of the U-sequence is moderate, but then gets stuck in the 10th round, where it spends more than 50 minutes. Indeed, all the states in the $U$ frame are explored (see line 5 in Algorithm 1) before a new $O$ frame can be opened in line 24. Perhaps resetting the $U$ frame and progressing with a different literal ordering can converge faster.

Algorithm 5 shows our implementation of Hybrid-CAR, based on the observations above. The only difference between Hybrid-CAR and CAR falls in the dotted rectangle. A timer is kept in the SAT solver, starting at the beginning, to calculate the time consumption. Once it exceeds the time limit, it triggers the restart procedure. In the *restart* procedure, it only keeps the states in the lowest level in the $U$-sequence, clears all the others, and switches the reordering configuration. After that, it resets the timer, jumps out of the current loop, and starts searching with a new configuration from the beginning. In the *getNextConfig* procedure, we employ a simple strategy to change the demarcation of locality: increase the number of $iLimit$ by one. Finally, to preserve completeness, we give the option to increase the time limit each time restart is called.

## V. EXPERIMENTS

### A. Setup

Our experiments focused on bug-finding only, and accordingly we implemented our suggested algorithms on top of SIMPLECAR [24], [33], which is an implementation of the CAR algorithm, in its best version for bug-finding [15]. We compared ourselves to the best public BMC implementation (the one in ABC-BMC [9]), and the best combination of CAR and BMC in [38]. Our evaluation was based on 438

**Algorithm 5:** Hybrid-CAR.

```
Input: A transition system $Sys = (V, I, T)$ and a
       safety property $P$, time limit to restart
       $TimeLimit$
Output: 'Safe' or ('Unsafe' + a counterexample)
1  if $SAT(I \wedge \neg P)$ then return 'Unsafe'
2  $U_0 := I$, $O_0 := \neg P$;
3  while true do
4      $O_{tmp} := \neg I$
5      while $state := pickState(U)$ is successful do
6          $stack := \emptyset$
7          $stack.push(state, |O| - 1)$
8          while $|stack| \neq 0$ do
9              $(s, l) := stack.top()$
               if $timeExceed(TimeLimit)$ then
                   $conf := getNextConfig()$
10                 $restart()$;
11             if $l < 0$ then return 'Unsafe'
12             $\hat{s}$ = Reorder (s, l + 1, conf)
13             if $SAT(\hat{s}, T \wedge O'_l)$ then
14                 ...                ▷ Same as in original CAR
15     ...
```

benchmarks[2] in the Aiger [11] format from the single safety property track of the 2015 [18] and 2017 [19] Hardware Model Checking Competition (HWMCC [7])[3], which is consistent with the benchmark set of [38]. All the counterexamples found were successfully verified with the third-party tool *aigsim* that comes with the Aiger package [5]. All the artifacts are available in Github [13].

We ran the experiments on a cluster of Linux servers, each equipped with an Intel Xeon Gold 6132 14-core processor at 2.6 GHz and 96 GB RAM. The version of the operating system is Red Hat 4.8.5-16. For each running instance, the memory was limited to 8 GB; if not otherwise specified, the time was limited to 1 hour.

The following questions guided our evaluation:

- Q1: How does *CoreLocality* perform when compared to the present best reordering strategy in CAR, i.e., *Intersection + Rotation*?
- Q2: How useful can it be to integrate *CoreLocality* to the best CAR variants for bug-finding, i.e., the three presented in [38]?
- Q3: How does Hybrid-CAR perform when compared to the state-of-the-art bug-finding (unsafe checking) algorithms?

[2]Results of these benchmarks are either known to be unsafe or remain unknown.

[3]These are the last two years of HWMCC using the AIGER format. Since 2019 [20], the official format switched to a word-level format BTOR [10], [28].

**A1. *CoreLocality* VS. *Intersection* + *Rotation*.** The previous literal-ordering strategy for CAR, namely the combination of *Intersection* and *Rotation* as published in [15], is very close to *CoreLocality* when $iLimit$ is set to one ('Local-1'), except that *CoreLocality* introduces a reordering inside the UCs (see Sec. IV-A). Recall that in Sec. III we presented empirical evidence that confirms independently of [15] that these two strategies improve the empirical results.

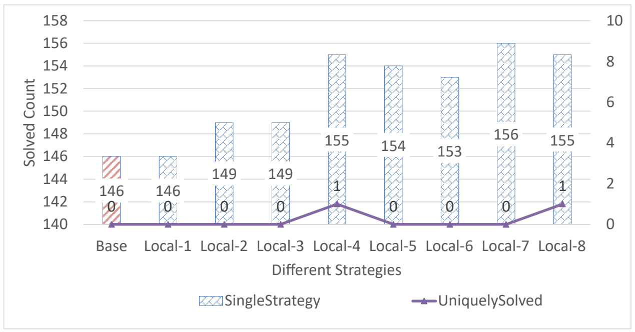


Fig. 6. Results on different reordering strategies, in terms of the total solved instances. In the figure, 'Base' refers to the combination of *Intersection* and *Rotation*, 'Local-$i$' ($1 \le i \le 8$) represents the *CoreLocality* strategy with $iLimit = i$.

As is shown in Figure 6, the performance of *CoreLocality* with $1 < iLimit \le 8$ outperforms that of the base strategy. The peak performance occurs with $iLimit = 7$, which solves 156 cases in total and obtains a 7% improvement compared to the prior best strategy (Base) in [15]. The various strategies solve different cases, as is evident by looking at the graphs depicting the virtual best solver, with and without the base. No instance is uniquely solved by Base, indicating that a *Local* portfolio can cover all the cases.

Increasing the number of solved cases, even by a few instances, is important in light of the decades of research and development of model checkers.

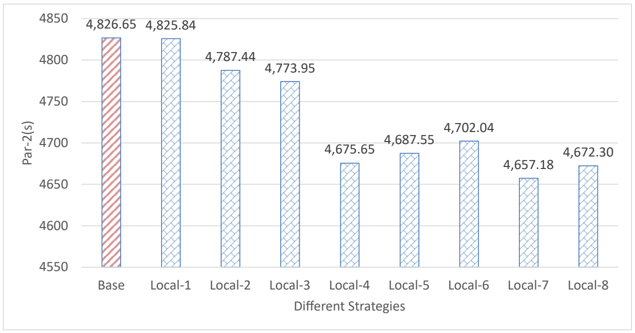


Fig. 7. Results on different reordering strategies, using the Par-2 score.

A comparison of the run time of the different strategies is shown in Figure 7. We rank them using the Par-2 score, which is calculated by the average time consumption of all cases while doubling the run time of instances that timed out. This is a common factor measured in the SAT community [31]. The figure illustrates that *CoreLocality* with $iLimit > 1$ consumes less time than Base. It is also apparent that there is a correlation between the number of solved instances and the total time consumption. *CoreLocality* outperforms the Base strategy on both the number and time of solved cases. A detailed pairwise comparison between the peak and Base is shown on the right.

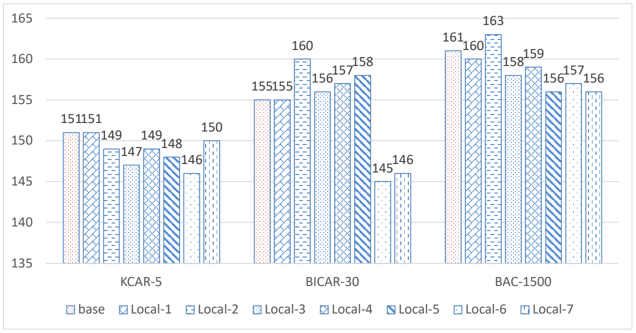
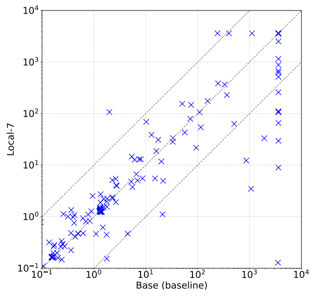


Fig. 8. Results of integrating *CoreLocality* into KCAR-5, BICAR-30, and BAC-1500 with $iLimit$ ranging from 1 to 7.



Notably, the performance of *CoreLocality* with different configurations is not correlated to the value of $iLimit$. This is consistent with our discussion in Section IV-A, that increasing the limit does not have a monotonic effect.

**A2. The effect of *CoreLocality* on the CAR +BMC combination of [38].** We also implemented and evaluated *CoreLocality* on top of the three best combinations between CAR and BMC, i.e., BAC-1500, BICAR-30, and KCAR-5, from [38]. The results are shown in Figure 8. Generally speaking, *CoreLocality* can be helpful for BICAR-30 and BAC-1500 to solve more instances when $iLimit = 2$ (2 and 5 more instances, respectively), though it seems to be detrimental in other cases.

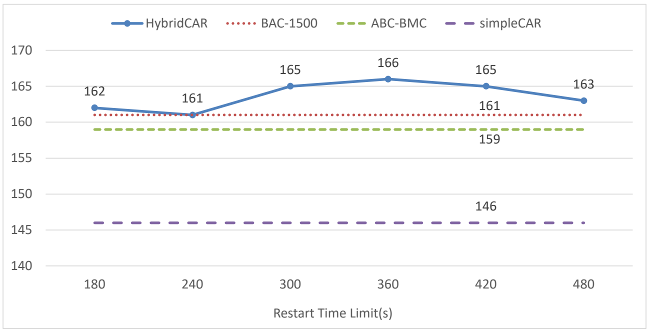


Fig. 9. Comparison on Hybrid-CAR with different restarting limits to BAC-1500, ABC-BMC, as well as the original SIMPLECAR. Time limit is 1 hour. The X-axis represents different restarting limits.

**A3. Hybrid-CAR VS. state-of-the-art bug-finding algorithms.** We compared Hybrid-CAR to the original SIMPLE-

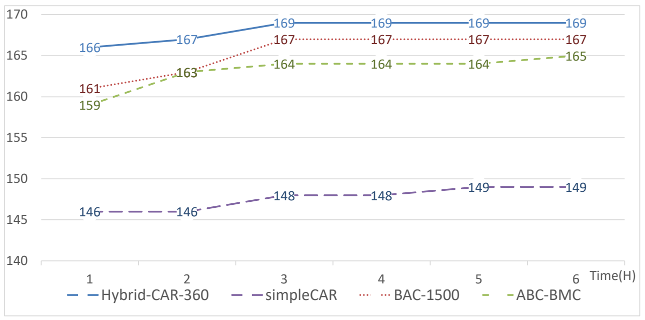


Fig. 10. Comparison on Hybrid-CAR with different restarting limits to the original SIMPLECAR, ABC-BMC, and BAC-1500, the latter two of which are shown to be the state of the art in [38]. In the figure, Hybrid-CAR-360 refers to Hybrid-CAR with the restart limit set to 360 seconds. Timeout is up to 6 hours. The X-axis represents CPU running time.

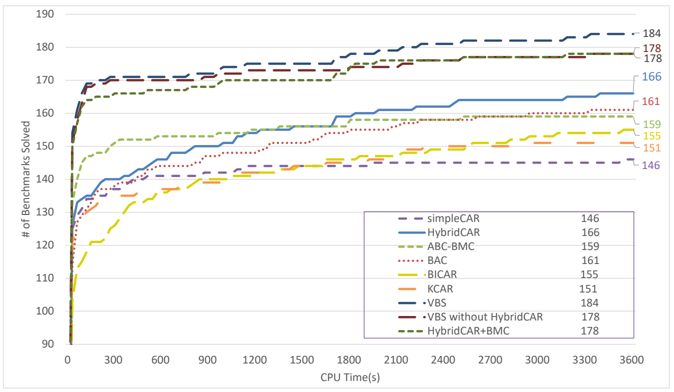


Fig. 11. Comparison of run-time performance among different model checkers. VBS represents the virtual best, i.e., parallel running all and taking the best.

CAR, BAC-1500 (the best solution shown in [38]) and ABC-BMC on bug finding[4]. To fully evaluate the scalability of these different approaches, we ran the experiments with two separate time limits: 1 hour and 6 hours. The corresponding results are shown in Figure 9 (1 hour) and Figure 10 (6 hours).

It turns out that regardless of the time limit for restart, Hybrid-CAR performs better than the competitors. In the one-hour setting, the best version of Hybrid-CAR, in which the restart is invoked every 6 minutes, solves 166 cases in total, which is seven more than that solved by ABC-BMC, and 20 more than that solved by the original SIMPLECAR. In the 6-hour setting, this version of Hybrid-CAR solves 169 cases, which is 20 more than the original SIMPLECAR, and outperforms the competition. Note that Hybrid-CAR solves more instances in just one hour (166) than that solved by ABC-BMC in 6 hours (165).

Figure 11 also includes the comparison among the best version of Hybrid-CAR and the other two combinations of BMC and CAR presented in [38], i.e., BICAR and KCAR. The

[4]While PDR is good at proving safe, it is not as good in finding bugs. The best implementations of PDR, to our knowledge, namely ABC-PDR and NUXMV-IC3, cannot solve more than 140 cases within the time limit. So do other variants such as AVY [35], [36] and QUIP [22]. For this reason they are not included in the comparison.

TABLE VII
UNIQUELY SOLVED INSTANCES FOR EACH ALGORITHM.

| | UNIQUELY SOLVED | UNIQUELY SOLVED (COMPARED TO ABC-BMC) | UNIQUELY SOLVED (COMPARED TO BAC) |
|---|---|---|---|
| simpleCAR | 0 | 13 | 4 |
| **Hybrid-CAR** | **6** | **19** | 11 |
| BAC | 4 | 17 | 0 |
| ABC-BMC | 5 | 0 | **15** |
| BICAR | 0 | 5 | 10 |
| KCAR | 1 | 10 | 6 |

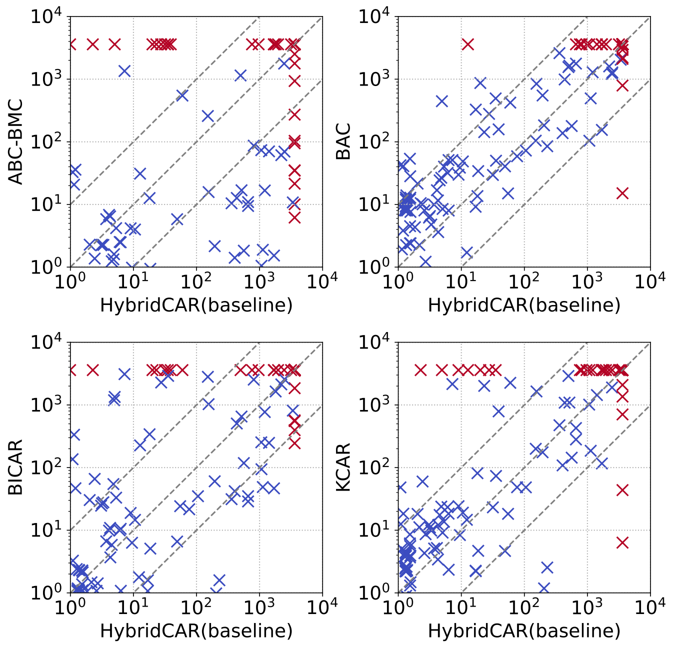


Fig. 12. Pairwise Comparison of Hybrid-CAR and competitors. Timeout instances in either are marked in red.

timeout here is one hour. Hybrid-CAR performs better than all the other methods. In terms of the number of solved unsafe instances, Hybrid-CAR is 166, followed by BAC (161), ABC-BMC (159), BICAR (155), KCAR (151), and the original CAR (146). In particular, Hybrid-CAR can solve six *unique* benchmarks, i.e., benchmarks that cannot be solved by all the other methods.

Table VII shows the uniquely solved instances of each technique (i.e. that no other tool can solve), and, in parenthesis, in comparison to ABC-BMC and BAC, e.g., Hybrid-CAR solves 19 and 11 cases that cannot be solved by these two tools, respectively. Moreover, we note that a portfolio of only Hybrid-CAR and ABC-BMC can solve 178 instances, almost reaching the virtual best results (184) that a portfolio of all these algorithms can solve. A detailed pairwise comparison is shown in Fig. 12.

## VI. Conclusion

In this paper, we revisited the assumption literal ordering strategies presented in [15]. We hypothesized that *Intersection* works because of what we call *core locality*, which means that similar cores help the SAT solver find proofs faster. Our empirical data, as we showed, supports this claim. Both *Intersection* and *Rotation* determine only a part of the literal order, hence the order of most of the assumptions is left arbitrary. Our improved strategy, CoreLocality (Sec. IV-A), generalizes *Intersection* and orders a larger part of the assumptions sequence, while improving the core locality. Together with prioritizing *conflict literals* (Sec. IV-B) they shorten rather significantly the time it takes the SAT solver to find proofs. We also presented a hybrid approach called Hybrid-CAR (Sec. IV-C), which switches between different configurations of CoreLocality during run time, while resetting the $U$ sequences. Our results show that these strategies perform better on average than the reordering strategies of [15] and also better than the various integrations of CAR with BMC [38]. In particular, Hybrid-CAR is able to outperform all bug-finding model-checking algorithms off-the-shelf. It is left for future work to try these strategies on PDR.

## REFERENCES

[1] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to mus extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013*, pages 309–317, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[2] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[3] Alessandro Bernardini, Wolfgang Ecker, and Ulf Schlichtmann. Where formal verification can help in functional safety analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[4] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC)*, pages 317–320, 1999.

[5] Armin Biere. AIGER Format. http://fmv.jku.at/aiger/FORMAT.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[7] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 9–9, 2017.

[8] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[9] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Robert Brummayer, Armin Biere, and Florian Lonsing. Btor: bit-precise modelling of word-level problems for model checking. In *Proceedings of the joint workshops of the 6th international workshop on satisfiability modulo theories and 1st international workshop on bit-precise reasoning*, pages 33–38, 2008.

[11] Robert Brummayer, Alessandro Cimatti, Koen Claessen, Niklas Een, Marc Herbstritt, Hyondeuk Kim, Toni Jussila, Ken McMillan, Alan Mishchenko, Fabio Somenzi, et al. The aiger and-inverter graph (aig) format version 20070427. In *The AIGER And-Inverter Graph (AIG) Format Version 20070427*, 2007.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[13] Artifacts. https://github.com/AnonymousAccO-O-O/HybridCAR.

[14] Yibo Dong, Xiaoyu Zhang, Yicong Xu, Chang Cai, Yu Chen, Weikai Miao, Jianwen Li, and Geguang Pu. Lightf3: A lightweight fully-process formal framework for automated verifying railway interlocking systems. New York, NY, USA, 2023. Association for Computing Machinery.

[15] Rohit Dureja, Jianwen Li, Geguang Pu, Moshe Y. Vardi, and Kristin Y. Rozier. Intersection and rotation of assumption literals boosts bug-finding. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 180–192, Cham, 2020. Springer International Publishing.

[16] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 125–134, Austin, Texas, 2011. FMCAD Inc.

[17] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[18] HWMCC 2015. http://fmv.jku.at/hwmcc15/.

[19] HWMCC 2017. http://fmv.jku.at/hwmcc17/.

[20] HWMCC 2019. https://fmv.jku.at/hwmcc19/.

[21] IC3Ref. https://github.com/arbrad/IC3ref.

[22] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 65–72, Austin, Texas, 2015. FMCAD Inc.

[23] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[24] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y. Vardi. SimpleCAR: An efficient bug-finding tool based on approximate reachability. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 37–44, Cham, 2018. Springer International Publishing.

[25] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y Vardi. SimpleCAR: An efficient bug-finding tool based on approximate reachability. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 37–44. Springer, 2018.

[26] Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, and Moshe Y. Vardi. Safety model checking with complementary approximations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, pages 95–100. IEEE Press, 2017.

[27] K. L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[28] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 587–595, Cham, 2018. Springer International Publishing.

[29] Chanseok Oh. Between sat and unsat: The fundamental difference in CDCL SAT. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 307–323, Cham, 2015. Springer International Publishing.

[30] Kristin Y. Rozier, Natarajan Shankar, Cesare Tinelli, and Moshe Vardi. Developing an open-source, state-of-the-art symbolic model-checking framework for the model-checking research community. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–1, 2023.

[31] The international SAT competitions. http://www.satcompetition.org/.

[32] Tobias Seufert, Christoph Scholl, Arun Chandrasekharan, Sven Reimer, and Tobias Welp. Making progress in property directed reachability. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 355–377, Cham, 2022. Springer International Publishing.

[33] SimpleCAR. https://github.com/lijwen2748/simplecar/releases/tag/v0.1.

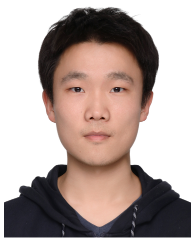
[34] Muralidhar Talupur. Hardware model checking: Status, challenges, and opportunities. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 154–154, 2011.

[35] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 367–385, Cham, 2019. Springer International Publishing.

[36] Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 260–276, Cham, 2014. Springer International Publishing.

[37] Yechuan Xia, Anna Becchi, Alessandro Cimatti, Alberto Griggio, Jianwen Li, and Geguang Pu. Searching for i-good lemmas to accelerate safety model checking. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 288–308, Cham, 2023. Springer Nature Switzerland.

[38] X. Zhang, S. Xiao, J. Li, G. Pu, and O. Strichman. Combining bmc and complementary approximate reachability to accelerate bug-finding. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.



**Yibo Dong** received the B.S. degree from Shanghai Jiao Tong University, Shanghai, China, in 2021. He is pursuing an M.S. degree with the Software Engineering Institute at East China Normal University, Shanghai. His main research interest lies in formal verification, especially model checking.

**Yu Chen** received her B.S. degree from Shanghai University, Shanghai, China, in 2020 and her M.S. degree from East China Normal University, Shanghai, China, in 2023. She is currently a teaching assistant at Chuzhou University, Anhui, China. Her main research interest lies in temporal logic and model checking.

**Jianwen Li** He received his Ph.D. degree from the Software Engineering Institute, East China Normal University, Shanghai, China, in 2014. He is currently a Research Professor at the Software Engineering Institute, East China Normal University. His research interests include formal verification, logic and automata theory.

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

**Ofer Strichman** Prof. Ofer Strichman earned his PhD in 2001 from the Weizmann Institute, where he worked, under the supervision of Amir Pnueli, on translation validation for compilers, Bounded Model Checking, and other topics in formal verification. He then was a post-doc in Carnegie Mellon University in Ed Clark's group, where he mostly worked on model-checking, learning, predicate abstraction and decision procedures. Prof. Strichman published two books: "Decision procedures – an algorithmic point of view" together with Daniel Kroening, and "Efficient decision procedures for validation", edited two others and coauthored more than 100 peer-reviewed articles, mostly in formal verification and SAT. In the SAT field he is mostly known for his contributions to linear-time proof manipulations, exploiting symmetries and incremental satisfiability. In formal verification he is mostly known for his invention of 'regression verification' and various decision procedures, mostly for equalities with uninterpreted functions.

Prof. Strichman won the 2021 CAV award "for pioneering contributions to the foundations of the theory and practice of satisfiability modulo theories (SMT)".