Unleash the Hidden Power of CAR-based Model Checking through Dynamic Traversal

Yibo Dong[1], Yu Chen[2], Jianwen Li[1⋆], and Geguang Pu[1]

[1] Software Engineering Institute, East China Normal University, China
[2] The School of Computer and Information Engineering, Chuzhou University, China

Abstract. Complementary Approximate Reachability (CAR) is a leading SAT-based model checking algorithm that combines under- and over-approximating state sequences to verify safety properties. However, its performance is hindered by redundant computations caused by the fixed-order traversal of the under-approximating sequence. To address such a limit, in this paper, we propose a dynamic traversal strategy to optimize CAR. By identifying common inefficiency patterns, we introduce heuristic methods and a scoring mechanism to prioritize states that are more likely to advance verification. We also prove that the correctness of the CAR algorithm can be preserved while exploring only a subset of the U-sequence, enabling partial traversal strategies that significantly reduce computational overhead. Experimental results demonstrate that our approach could solve 10% more cases than the previous best CAR implementation [17] and outperform state-of-the-art IC3 model checkers, e.g., `IC3-REF` [4, 11]. Our method bridges the gap between CAR's theoretical potential and practical scalability, offering a more efficient solution for industrial-scale verification.

Keywords: Model Checking, Formal Verification, Complementary Approximate Reachability

1 Introduction

Model checking has long been a cornerstone of formal verification, offering rigorous guarantees for the correctness of both hardware and software systems. Given a system model M and a temporal property P, model checking automatically verifies whether all behaviors of M satisfy P. Despite its widespread adoption, scalability remains a critical challenge, particularly for large, industrial-scale systems where the state space grows exponentially. In these settings, traditional methods often struggle to meet the time and memory constraints required for practical use, making the quest for scalable, efficient model checking methods a vital area of research.

[⋆] We thank the anonymous reviewers for their insightful feedback. Jianwen Li is the corresponding author. This work is supported by NSFC Grant #62372178 and #U21B2015.

State-of-the-art safety model checking techniques, including Bounded Model Checking (BMC) [2, 3], Property Directed Reachability (PDR/IC3) [4, 9], and Complementary Approximate Reachability (CAR) [15], rely fundamentally on SAT solvers but adopt divergent strategies. BMC prioritizes shallow bug detection through bounded path exploration, offering speed at the cost of incompleteness. In contrast, PDR and CAR provide completeness but are generally slower for shallow bug-finding. Therefore, a combination of techniques is often used depending on the verification task.

Among these, the CAR framework uniquely combines over-approximation (O-sequence) and under-approximation (U-sequence), efficiently narrowing the search space and accelerating the verification process through a balance of bug-finding and proof capabilities. However, its practical scalability remains constrained by a rigid traversal of the U-sequence. This fixed-order strategy forces CAR to process states sequentially, irrespective of their individual utility, This results in two predominant inefficiency patterns: (1) *redundant states*, where multiple states generate identical unsatisfiable cores (UCs), and (2) *misleading states*, where a number of states repeatedly fail to transition into deeper O-frames, yet still consuming computational cycles due to their position in the traversal order. Consequently, even state-of-the-art implementations struggle with large-scale systems.

While prior optimizations like clause generalization [17] and assumption ordering [6–8] have sought to mitigate some of CAR's inefficiencies, they fail to address the core issue, i.e., the inflexible traversal order. In this paper, we propose a dynamic traversal strategy to improve CAR 's performance by addressing these inefficiencies. Our approach introduces two key heuristics: *PickUC*, which prioritizes states that contribute more to narrowing the O-sequence, and *PickChildren*, which favors states with higher branching factors to avoid unproductive paths. Both heuristics are integrated into a unified scoring mechanism that ranks states based on their potential to advance the verification process. These strategies are part of a broader optimization we term *Dynamic Traversal (DT)*, which refines the traversal of the U-sequence.

While reordering improves state prioritization, it still involves full traversal of the U-sequence, meaning that even lower-priority states are still explored. However, we demonstrate that full traversal is not necessary for correctness. Specifically, we prove that CAR can retain its correctness while exploring only a subset of the U-sequence. Building on this insight, we propose a further optimization of dynamic traversal: instead of fully traversing, we selectively explore only a subset of states. This approach, which we call `CAR-DT` (CAR with the dynamic traversal optimization), focuses on the most promising states and therefore reduces unnecessary exploration and accelerating convergence, especially for large-scale verification tasks.

We implement our approaches on the best variant of CAR and conduct an extensive evaluation using all the 318 benchmarks from the HWMCC'24 competition [10] to assess its performance. The experimental results demonstrate that `CAR-DT` outperforms the original CAR, solving 13 more out of 145 cases. In comparison to state-of-the-art tools like `IC3-REF` [4, 11], `CAR-DT` also demonstrates superior performance. These improvements bridge the gap between CAR's theoretical potential and its practical applicability, offering a scalable and efficient solution for industrial-scale verification tasks.

Our contributions can be summarized as follows:

- **Prioritized Traversal Strategy:** We propose a prioritized traversal strategy that optimizes U-sequence processing by prioritizing states based on their potential to refine O-frames or explore new transitions.
- **Theoretical Insight:** We prove that CAR's correctness is maintained while exploring only a subset of the U-sequence. This insight challenges the conventional wisdom that full traversal is necessary for correctness and lays the foundation for more efficient exploration strategies.
- **Dynamic Traversal:** Based on the theoretical insight, we introduce the *Dynamic Traversal* optimization, which focuses on high-potential states, reducing computation and accelerating convergence.
- **Empirical Validation:** We implement `CAR-DT` on the best variant of CAR and validate it on 318 benchmarks from the HWMCC'24 competition. Our results show that `CAR-DT` outperforms the original CAR and state-of-the-art tools like `IC3-REF`, offering a scalable solution for large-scale verification.

The remainder of this paper is organized as follows: In Section 3, we provide motivating examples. Section 4 then outlines our methodology, followed by the experimental results presented in Section 5.

2 Preliminaries

2.1 Boolean Transition System

A *Boolean transition system* Sys is defined as a tuple (V, I, T), where V is a set of Boolean variables, and each state s is a truth assignment to variables in V. I is a Boolean formula corresponding to the set of initial states. The transition relation T is a Boolean formula over $V \cup V'$, where V' is the set of primed variables. A 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 in Sys is a sequence $s_1, s_2, \ldots, s_k$ of states connected by transitions. A state t is reachable from s in k steps if there is a path of length k from s to t. Let S be a set of states in Sys. We denote the set of successors of states in S as $R(S) = \{t \mid (s, t) \in T, s \in S\}$. Conversely, we define the set of predecessors of states in S as $R^{-1}(S) = \{s \mid (s, t) \in T, t \in X\}$. Recursively, we define $R^0(S) = S$ and $R^i(S) = R(R^{i-1}(S))$ where $i \geq 0$, and the notation $R^{-i}(S)$ is defined analogously. In short, $R^i(S)$ denotes the states that are reachable from S in i steps, and $R^{-i}(S)$ denotes the states that can reach S in i steps.

2.2 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 [2]. 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 $R^i(S)$ with increasing values of i, while backward search begins with states in $\neg P$ and computes $R^{-i}(S)$ with increasing values of i to search for states reaching I. Table 1 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 $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ 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.

Table 1. 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	$F_i \cap \neg P \neq \emptyset$	$B_i \cap I \neq \emptyset$

2.3 Complementary Approximate Reachability (CAR)

CAR is a recently proposed SAT-based model checking algorithm inspired by IC3/PDR [4, 9]. CAR performs reachability analysis in both forward and backward directions by maintaining over- and under- approximate state sequences, which is 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 $O \equiv O_0, O_1, \ldots, O_i$ $(i \geq 0)$, and the under-approximating state sequence $U \equiv U_0, U_1, \ldots, U_j$ $(j \geq 0)$ are finite sequences of state sets, defined as shown in Table 1, where k denotes the frame index in the induction process, and $k \geq 0$.*

Table 2. Definition of over-/under-approximate state sequences in CAR

	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:	$O_k \cap I = \emptyset$	$--$

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)
if SAT $(I \wedge \neg P)$ then return Unsafe
$U_0 := I$, $O_0 := \neg P$
while True do
    $O_{tmp} := \neg I$
    while state :=pickState (U) is successful do
        stack := $\emptyset$
        stack.push (state, $|O| - 1$)
        while stack.size $\neq 0$ do
            (s,l) := stack.top()                        // Assume s $\in U_j$
            if l $< 0$ then return Unsafe
            if isBlockedAt (s, l) then
                backtrack (s,l)
                Continue
            if SAT $(s, T \wedge O'_l)$ then
                t := getModel()
                $U_{j+1} := U_{j+1} \cup$ t                   // Widening $U$
                stack.push (t,l-1)
            else
                stack.pop()
                uc := getUC()
                if $l + 1 < |O|$ then $O_{l+1} := O_{l+1} \wedge (\neg$uc )
                else $O_{tmp} := O_{tmp} \wedge (\neg$uc )
                backtrack (s,l)
    if $\exists i \geq 1$ s.t. $(\bigcup_{0 \leq j \leq i} O_j) \supseteq O_{i+1}$ then return Safe
    Add a new state-set to $O$ and initialize it to $O_{tmp}$
```

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* $\exists i \geq 1 \cdot (\bigcup_{j=0}^{i} O_j) \supseteq O_{i+1}$.

CAR can be implemented in both forward (ForwardCAR) and backward (BackwardCAR) modes. BackwardCAR is advantageous for finding unsafe bugs, while ForwardCAR is effective for proving safety [14, 15].

As shown in Algorithm 1, CAR works by progressively widening the U sets and narrowing the O sets. These sets are initialized at Line 2 with U set to the state I and O set to $\neg P$. The algorithm maintains a stack of pairs $\langle state, level\rangle$, where each state is associated with an index corresponding to an O frame. The temporary frame O_{tmp}, initialized to $\neg I$ at Line 4, is updated during the process to represent the next frame to be created.

In each iteration, CAR picks a state from the U-sequence, by default from the beginning to the end, as shown in Line 5, and pushes it to the stack. Then, for the state s at the top of the stack, CAR first checks if it is already blocked at this frame(Line 11). If so, CAR backtracks this state to a higher level(Line 12), skipping over frames that already block this state. Otherwise, CAR goes on to check if it can reach the O_l frame by checking whether $s \wedge T \wedge O_l'$ (Line 14) is satisfiable. If yes, a new state $t \in O_l$ is extracted from the model and added to the U-sequence, thereby widening it(Lines 15-17). Otherwise, the algorithm uses the unsatisfiable core (denoted as UC) to constrain the O frame for the next level, narrowing it (Lines 19-21), and pushes s back onto the stack. Afterward, CAR backtracks this state at Line 23.

Finally, CAR terminates the checking procedure with either 'Safe' or 'Unsafe'. The unsafe check attempts to find a path from I to $\neg P$ while the working level l is less than 0(Line 10) and provides a counterexample. The safe check propagates clauses from O_i to O_{i+1}, checking for a fixpoint. If reached, the algorithm returns 'Safe' based on the check in line 24.

3 Motivating Examples

Before delving into the examples, let us briefly review how CAR operates. The CAR algorithm relies on SAT queries as its core mechanism to advance its search process (see Line 14 of Algorithm 1). The SAT query checks whether a state s from the U-sequence can transition to a particular O-frame O_i. If the query is successful (lines 15–17), a new reachable state is identified, and CAR uses this state to expand the U-sequence. Conversely, if the query fails, CAR refines the O-sequence by incorporating the negation of the identified UC. The search process of CAR is illustrated in Figure 1.

CAR enhances search efficiency by retaining the reachable states in the U sequence for future reuse, distinguishing it from the PDR algorithm. However, an overabundance of remembered states could occasionally pose challenges. In practice, we have observed that the CAR algorithm can sometimes be stuck during the search process in certain situations, causing timeouts. Specifically, the search process can get stuck within a subspace, consuming a significant amount of time during a single iteration. This prompted us to investigate these occurrences further, where we identified two recurring patterns that frequently contribute to inefficient search progress.

Redundant States During the execution of the CAR algorithm, we observed that when multiple states in the U-sequence attempt to reach a specific O-frame, the UC generated by one state often blocks the others. For instance,

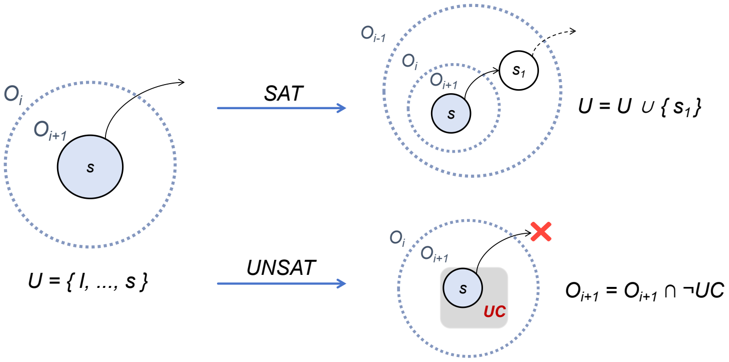

Fig. 1. The CAR algorithm searches for reachable states by leveraging SAT queries. When a state s in the U-sequence can transition to an O-frame O_i, it expands the U-sequence. Otherwise, it refines the O-sequence.

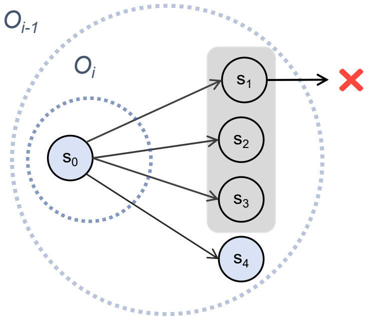

Fig. 2. Redundant states in the same subspace could generate identical UCs when attempting to reach O_{i-1}. Only one state (e.g., s_1) needs to be explored, while others (e.g., s_2, s_3) are redundant.

as depicted in Figure 2, state s_0 has four successor states in O_i. The shaded area indicates that s_1, s_2, and s_3 are within the same state space. When these states attempt to transition to a specific O-frame O_{i-1}, the UC generated by one of them (e.g., s_1) blocks the others (e.g., s_2 and s_3). This means that searching for s_2 and s_3 is redundant because their outcomes are already determined by s_1.

The root cause of this redundancy lies in the nature of SAT queries and the behavior of SAT solvers. When a SAT query is satisfiable, there may be multiple distinct assignments that satisfy the query. However, the SAT solver only returns one of these assignments. Often, some variables in the query are free, meaning their values do not affect the satisfiability of the query. Different assignments of these free variables can lead to multiple distinct states being generated. For example, if the SAT solver returns a state s_1 with a specific assignment of free variables, other states s_2 and s_3 that differ only in the assignments of these free variables will also be generated. When these states later attempt to transition to O_{i-1}, they are all blocked by the same UC generated for s_1, making their search redundant. Although CAR has a pre-check(Line 11 of Alg. 1), the overhead is not negligible, especially when with a large amount of accumulated UCs.

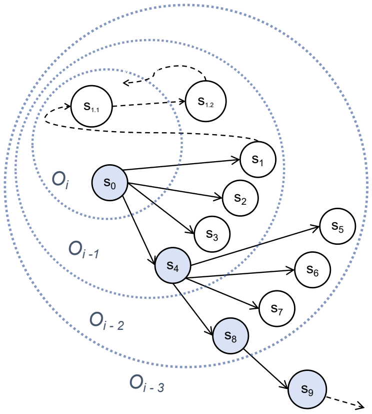

Fig. 3. Misleading states(e.g. s_1, s_2, s_3) cannot transition to lower frames, leading to the inefficient search.

Misleading States. We also observed an inefficiency in the search process when a state attempts to transition multiple steps. Some states returned by the SAT solver are not part of the actual transition trail and are added to the U-sequence after being accessed once. These states may repeatedly fail when revisited in the future. As shown in Figure 3, s_0 in O_i can transition to O_{i-3} in 3 steps. During the search process, s_0 first reaches s_1 but fails to progress further and backtracks to $s_{1.1}$. Subsequently, s_0 reaches s_2 and s_3, each of which also fails to go further. After multiple SAT queries, s_0 finally reaches s_4, and through the trail $s_0 \to s_4 \to s_8 \to s_9$, it reaches O_{i-3}.

This pattern is caused by the nature of SAT solver encoding. The SAT solver encodes single-step transition relations, so the states it returns for each step may not be the intermediate states required for a multi-step transition. For example, even though s_0 can reach s_9 in 3 steps; when we ask the SAT solver whether s_0 can reach O_{i-1}, it returns SAT but provides a state (s_1) that cannot reach O_{i-2}. Only after several rounds of queries is s_4 found, allowing the transition to proceed. This results in misleading states such as $s_1 - s_3$ and $s_5 - s_7$ in the figure, which are products of the SAT solver's single-step encoding and the different possible assignments to the constraints. Moreover, the presence of the backtracking mechanism in the CAR algorithm exacerbates the issue, as s_1 can introduce even more spurious states through backtracking, such as $s_{1.1} - s_{1.2}$.

4 Implementing Dynamic Traversal in CAR

Motivated by the observations in Section 3, we first introduce heuristic methods to optimize the traversal order of the U-sequence in CAR. These methods aim to address the inefficient search patterns in the original algorithm. Then, we propose the algorithm of CAR-DT (CAR with the dynamic traversal optimization), which can be regarded as a more flexible variant of CAR. Finally, we explore dynamic traversal approaches based on CAR to further enhance the efficiency.

4.1 Heuristic Methods for Optimizing U-Sequence Traversal

By recognizing the patterns observed in the CAR algorithm, we propose strategies to prioritize states more intelligently. Given that CAR is highly performance-sensitive, it is crucial to identify these patterns using methods that do not introduce additional computational costs. Fortunately, feasible approaches exist for both patterns. We introduce two heuristic methods, **PickUC** and **PickChildren**, corresponding to Redundant States and Misleading States, respectively. Additionally, we propose a combined scoring mechanism to dynamically prioritize states based on their potential to advance the search process.

PickUC: Distinct States First. For Redundant States, the number of UCs generated by a state can serve as an indicator. In the CAR algorithm, if a state is blocked by existing UCs, it will not generate a new UC despite consuming some checking time (the 'isBlockedAt' check, which can be generally understood as a subsumption test). Consequently, states that are 'always blocked by UCs generated by other states' will have a low number of generated UCs. This characteristic can be leveraged to efficiently identify Redundant States.

To address this, PickUC prioritizes states that generate more UCs in the previous round, as they are more likely to contribute to refining the O-sequence. Conversely, states blocked by existing UCs – therefore generating no new UCs – are deprioritized, reducing redundant computations.

PickChildren: Branching States First. For Misleading States, the number of children states can be used as a criterion. Misleading states, which repeatedly fail to transition further, tend to have a small number of children states (often zero). In contrast, correct states that can continue the transition process have more children (at least one). This difference in the number of children states can be used to distinguish between misleading and correct states.

To mitigate this issue, PickChildren prioritizes states with more successors (i.e., higher branching factors). These branching states are more likely to lead to productive transitions and help advance the search. By focusing on such states, the algorithm reduces time spent on unproductive leaf nodes.

Scoring mechanism: Combination of Both Criterions. The generation of UCs relies on the SAT solver returning 'UNSAT', while the generation of successors depends on 'SAT' queries. Although these processes may initially seem contradictory, they are actually complementary. Specifically, the same $(state, level)$ pair(see Line 9 of Alg 1) could be queried multiple times, yielding one or more SAT results, but ultimately resulting in a single UNSAT result (since the updated UCs prevent further successors from being found). For a specific state, the number of UCs reflects its ability to refine the O-sequence by eliminating redundant paths, while the number of successors indicates its potential to explore diverse transitions.

Algorithm 2: The CAR algorithm with Dynamic Traversal

```
   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 := pickStateDynamically (U) is successful do
6          stack := $\emptyset$
7          stack.push (state, $|O| - 1$)
8          while stack.size $\neq 0$ do
9              (s,l) := stack.top()                    // Assume s $\in U_j$
10             ...
11         ...
12     if $\exists i \geq 1$ s.t. $(\bigcup_{0 \leq j \leq i} O_j) \supseteq O_{i+1}$ then return Safe
13     Add a new state-set to $O$ and initialize it to $O_{tmp}$
```

Building on these insights, we introduce a scoring mechanism that ranks states within the U-sequence. By balancing the factors, we can prioritizes states that efficiently prune the search space and explore deeper levels.

Specifically, each state is assigned a score based on a weighted sum of its UC count and number of successors:

$$\text{score} = w \cdot \text{num(UCs)} + (1 - w) \cdot \text{num(successors)} \tag{1}$$

where w is a tunable weight balancing the contributions of these two factors. States are sorted in descending order of their scores in each iteration, with the initial state always preserved to ensure correctness. This mechanism directs the traversal toward states that either block redundant explorations through UCs or offer diverse transitions through high branching factors, thereby enhancing the efficiency of the CAR algorithm.

4.2 CAR with Partial Traversal

Prioritized traversal reduces the priority of low-value states, yet these states can still be visited. To avoid unnecessary exploration of less promising states, we propose the **Partial Traversal** strategy to enhance dynamic traversal. This strategy explores only a selected portion of the U-sequence to improve efficiency. Alg. 2 shows the implementation of the updated CAR-DT. The only difference between CAR-DT and the original CAR algorithm lies in the state selection process(see Line 5). Specifically, CAR-DT only visits a subset of U-sequence. We then prove that as long as the initial state is not excluded by the partial traversal, the completeness of CAR-DT is maintained, as stated in the following theorem:

Theorem 1 (Completeness). *Given a Boolean transition system Sys and a safety property P,* `CAR-DT` *terminates with UNSAFE if* $Sys \not\models P$ *and terminates with SAFE if* $Sys \models P$*.*

Before proving the theorem, we first prove two lemmas separately. [3]

Lemma 1 (Completeness-UNSAFE). *Given a Boolean transition system Sys and a safety property P,* `CAR-DT` *terminates with UNSAFE if* $Sys \not\models P$*.*

Proof. If the given problem is UNSAFE, there exists a finite path ρ from I to $\neg P$. Let the length of ρ be $n+1$, and label the states on the path as $\rho[j]$ for $0 \leq j \leq n$, where $\rho[n] = I$ and $\rho[0] \in \neg P$.

Assume the O-sequence has grown to size $n+1$[4], i.e., we have $O_0, O_1, \ldots, O_n$. Since ρ is a valid path, for each j from 0 to $n-1$, $\rho[j+1] \wedge T \wedge O'_j$ is SAT, where O'_j is the predicate for the next frame.

In `CAR-DT`, when the initial state I is selected (guaranteed not to be missed by **pickStateDynamically**), the algorithm checks if $I \wedge T \wedge O'_n$ is SAT. Since I can reach $\rho[n-1]$ in one step, this query will be SAT. And with finite steps[5], $\rho[n-1]$ will be found and added to the working stack.

By induction, assume that $\rho[k]$ has been found and pushed to the working stack for some $k < n$. When $\rho[k]$ is selected, the algorithm checks if $\rho[k] \wedge T \wedge O'_{k-1}$ is SAT. Similarly, this query will be SAT, and $\rho[k-1]$ will be pushed to the stack.

Finally, $\rho[0] \in \neg P$ will be found, and the algorithm will terminate with UNSAFE, returning the counterexample ρ. □

Lemma 2 (Completeness-SAFE). *Given a Boolean transition system Sys and a safety property P,* `CAR-DT` *terminates with SAFE if* $Sys \models P$*.*

Proof. By the design of `CAR-DT`, the O-sequence is monotonically increasing, i.e., once the negation of a UC is added to an O-frame, it will never be removed. With this continuous refinement, the O-sequence becomes increasingly precise, and will eventually converge to the real reachable set R.

Since the system is SAFE, there exists no path from I to $\neg P$. As a result, the O-sequence will eventually reach a fixpoint where no further states can be added to the over-approximation. Formally, there exists a minimal k such that: $\left(\bigcup_{0 \leq j \leq k} O_j\right) \supseteq O_{k+1}$ This implies that the O-sequence has stabilized and no new states can be reached beyond this point.

In `CAR-DT`, once the O-sequence reaches this fixpoint, the algorithm checks whether the condition $\exists i \geq 1$ such that $\left(\bigcup_{0 \leq j \leq i} O_j\right) \supseteq O_{i+1}$ holds. If this condition is satisfied, the algorithm terminates and returns 'Safe'.

Since the O-sequence is guaranteed to converge to the true reachable set R, the algorithm will eventually detect this fixpoint and correctly terminate with 'Safe'. □

[3] Due to space constraints, we only demonstrate the correctness of backward-CAR, which involves a forward search. Forward-CAR can be proved similarly.

[4] This is guaranteed to happen within finite time. The proof is the same as the original CAR algorithm.

[5] This is guaranteed by the termination of the original CAR algorithm.

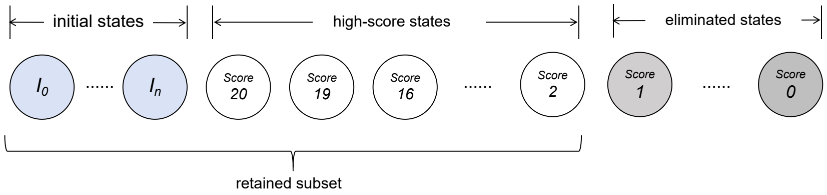

Fig. 4. Reordered U-sequence after scoring and truncation.

And finally, we prove Theorem 1:

Proof. An input problem could either be SAFE or UNSAFE. With the proved lemmas Lemma 1 and Lemma 2, the completeness of `CAR-DT` is proved. ■

Building on the insight that full traversal of the U-sequence is not necessary for correctness, we now present the details of the `CAR-DT` implementation. The partial traversal focuses on the initial state and the most promising states in the U-sequence, identified by their scores from Equation 1. Figure 4 illustrates the reordered U-sequence when applying partial traversal in `CAR-DT`. The traversal begins with the initial state, followed by states sorted according to their scores. States beyond the retained subset are excluded from further traversal. This approach reduces computational overhead by focusing on high-potential states while ensuring algorithm correctness by always preserving the initial states.

A special case of the partial traversal strategy retains only the initial state in the U-sequence. This is motivated by the fact that initial states typically exhibit high scores due to their lack of blocking constraints and high branching potential. By focusing exclusively on the initial states, this variant prevents the accumulation of redundant or misleading states and eliminates the need to manage a large U-sequence, resulting in $U = \{I\}$.

5 Evaluation

We implemented the proposed method on the state-of-the-art CAR-based single-core model checker SimpleCAR[6] [17], which incorporates the latest optimization of the CAR algorithm.[7]

5.1 Evaluation Setup

We conducted the experiments on a cluster, consisting of 240 nodes with 6720 processor cores altogether (14 processor cores per node) and running at 2.6GHz with 96GB of RAM per node. The operating system is RedHat 4.8.5-16.

[6] It is a core component of the SuperCAR model checker [16], which received the bronze award in recent HWMCC competition [10].

[7] All the artifacts are available at [1].

Table 3. Comparison of solved cases with different picking strategies. 'Basic' refers to the default traversal setting of the CAR algorithm. 'Par-2 Score' refers to the calculation of the average time consumption across all cases, where the run time of timed-out cases is counted as double.'VirtualBest' refers to taking the best result of each checker.

Method	Safe			Unsafe			Total	Par-2 Score
	Gain	Loss	Solved	Gain	Loss	Solved		
Basic	-	-	106	-	-	26	132	4299.81
PickUC	7	1	**112**	0	0	26	**138**	**4204.39**
PickChildren	6	1	111	1	2	25	136	4249.70
VirtualBest	-	-	115	-	-	27	142	-

Table 4. Performance of CAR with the scoring mechanism. VBS refers to Virtual Best.

Strategies	PickChildren	Scoring with different w				PickUC	VBS
	$(w = 0)$	$w = 0.3$	$w = 0.5$	$w = 0.7$	$w = 0.9$	$(w = 1)$	
Cases Solved	136	131	136	**139**	134	138	146
Safe Solved	111	107	111	**112**	109	112	118
Unsafe Solved	25	24	25	**27**	25	26	28
Par-2 Score	4249.70	4326.27	4247.25	**4184.82**	4266.53	4204.39	-

We evaluated our method using all 318 AIGER-format benchmarks from the HWMCC 2024 competition, including both safe and unsafe cases, to show the effectiveness of `CAR-DT` in proving safety and finding counterexamples. For each running instance, the memory was limited to 8 GB; if not otherwise specified, the time was limited to 1 hour.

5.2 Evaluation results

RQ1: How effective is the prioritized traversal strategy? To compare the effectiveness of different picking methods on the CAR algorithm, we first evaluated the performance of SimpleCAR under different state prioritizations, the corresponding results are shown in Table. 3.

The data reveals that both the PickUC and PickChildren strategies outperform the Basic traversal method. Specifically, PickUC gains 7 additional cases compared to Basic, despite incurring 1 loss. PickChildren gains 7 cases, which also contributes to the virtual best. Interestingly, virtual best solves 142 cases, with an increase of 10 cases, which is equivalent to the combined improvements of PickUC (6 cases) and PickChildren(4 cases) over Basic. This suggests that the strategies are complementary, indicating that integrating them into the scoring mechanism could further enhance the results.

To further explore the effects of combining the two picking strategies, we conducted experiments using various values of w, which represents the weight that balances the preference for avoiding either Redundant States or Misleading States. When $w = 1$, the method defaults to PickUC, while $w = 0$ corresponds

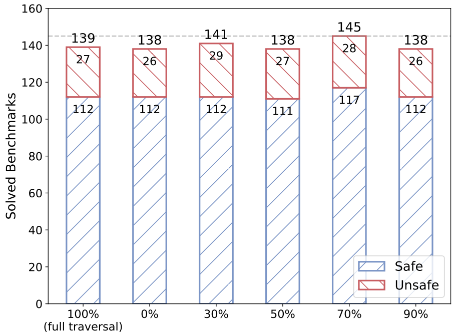

Fig. 5. The impact of different portion settings on the performance of the `CAR-DT` algorithm. The X-axis represents the portion setting. All variants are with the best Scoring strategy where $w = 0.7$.

Table 5. Par2-Score of `CAR-DT` with different configurations.

Strategies	Scoring method	`CAR-DT` (x)				
		$x = 0$	$x = 0.3$	$x = 0.5$	$x = 0.7$	$x = 0.9$
Par2-Score	4184.82	4190.66	4179.08	4201.00	**4074.68**	4212.43

to PickChildren. The experimental results, shown in Table 4, indicate that the optimal performance occurred when $w = 0.7$, which solved 112 safe cases and 27 unsafe cases. The virtual best of these scoring strategies reaches a total of 146 solved cases, showing that balancing the two patterns during the search process uncovers additional potential for solving even more cases.

Thus, the superior performance compared to the Basic method highlights the potential of **prioritized traversal** for advancing the verification process.

RQ2: Can CAR with Partial Traversal achieve better performance? To evaluate whether searching on a subset of the U-sequence can further enhance performance, we conducted experiments based on partial traversal. Specifically, in each iteration, we eliminated a certain proportion of states from the end of the ordered U-sequence, which corresponded to the least promising states. We denote this approach as `CAR-DT` (x), where only the top x states are retained. For example, `CAR-DT` (0%) reduces to the case where only the initial state is preserved, while `CAR-DT` (100%) retains all states, equivalent to the scoring method.

Building upon the optimal scoring strategy identified in RQ1, we conducted experiments with several portion settings. The corresponding results are presented in Figure 5. `CAR-DT` outperforms the optimal scoring method when the portion is set to 30% and 70%, whereas for other values, `CAR-DT` remains comparable to the scoring method. `CAR-DT` (70%) achieves a peak value of 145, yielding an improvement of 10% compared to the unoptimized Basic traversal strategy of CAR, representing a significant enhancement.

Table 5 compares the Par-2 score of `CAR-DT` across different portion settings. The results show that some variants achieves lower Par-2 scores compared to

Table 6. The number of instances solved with a 1-hour timeout by different model checkers. 'Gain' and 'Loss' refers to the comparison with CAR-DT. 'VBS' means the virtual best.

	Safe			Unsafe			Total
	Gain	Loss	Solved	Gain	Loss	Solved	
CAR-DT	-	-	117	-	-	28	145
ABC-PDR	15	3	129	0	11	17	146
IIMC-PDR	8	24	101	2	8	22	123
IC3-REF	2	12	107	11	12	27	134
AVY	11	13	115	12	6	34	149
VBS	-	-	137	-	-	47	184

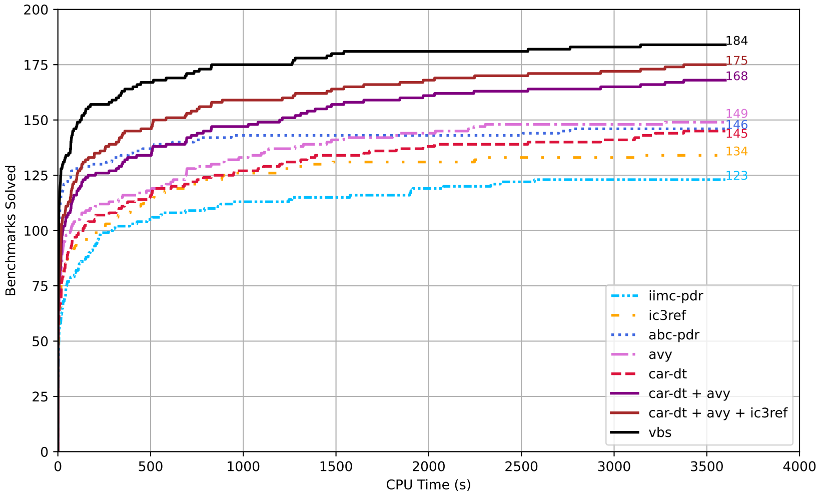

Fig. 6. Comparison of run-time performance among different model checkers.

the scoring strategy. This indicates that dynamically discarding an appropriate proportion of states from the U-sequence can significantly reduce the computational overhead and improve the overall efficiency of the CAR algorithm. For instance, CAR-DT (70%) achieved a Par-2 score of 4074.68, which is much lower than the score of 4184.82 obtained by the optimal weight of the scoring method, demonstrating a notable improvement in verification efficiency.

RQ3: How does CAR-DT perform when compared to the state-of-the-art model checking algorithms? To evaluate the performance of CAR-DT against state-of-the-art model checking algorithms, we conducted a comprehensive comparison with several leading tools, including ABC-PDR [5], IIMC-PDR [12], IC3-REF [11] and AVY [13].

As shown in Table 6, CAR-DT outperforms IC3-REF and IIMC-PDR, solving more instances within the given time limit. When compared to ABC-PDR and AVY, CAR-DT exhibits competitive performance, with a slight difference in the number of solved instances. It should also be noted that CAR-DT can solve more safe cases

(117) than AVY (115). This indicates that our approach is particularly effective in proving safety properties, which is a crucial aspect of model checking.

Moreover, CAR-DT demonstrates notable complementarity with other algorithms. For example, it can solve 14 instances that ABC-PDR cannot, while ABC-PDR can solve 15 instances that CAR-DT cannot. Similarly, CAR-DT has a gain of 32 instances over IIMC-PDR and a gain of 24 instances over IC3-REF. These results highlight the unique strengths of CAR-DT in certain types of verification tasks. The virtual best solver of all these tools can resolve a total of 184 cases, which underscores the potential for further improvement through the combination of different algorithms. The high complementarity among these tools suggests that integrating CAR-DT with other state-of-the-art model checkers could lead to a more comprehensive and efficient verification framework. For instance, combining CAR-DT and AVY could leverage the strengths of both tools to solve a broader range of benchmarks more effectively.

Figure 6 provides a more detailed comparison of the run-time performance of these model checkers. It shows that CAR-DT consistently solves a significant number of benchmarks faster than IC3-REF and IIMC-PDR. The performance gap between CAR-DT and ABC-PDR is relatively small, indicating that both tools are effective for a wide range of verification tasks. AVY, while having the highest overall performance, does not exhibit a clear dominance over our approach in terms of run-time efficiency for individual benchmarks.[8]

In conclusion, CAR-DT demonstrates strong performance compared to state-of-the-art model checking algorithms. It outperforms some leading tools and shows competitive performance with others, while also exhibiting notable complementarity. These results highlight the effectiveness of our proposed dynamic traversal approach in enhancing the efficiency and scalability of CAR-based model checking.

6 Conclusion

In this paper, we proposed a dynamic traversal strategy for optimizing CAR-based model checking. By implementing prioritized traversal on a subset of U-sequence, we introduced an efficient approach that reduces redundant computations and improves scalability. The experimental results show that the application of our method, CAR-DT, significantly outperforms the original CAR algorithm and achieves competitive performance compared to state-of-the-art model checkers.

This work enhances CAR's practical applicability, offering a more efficient solution for large-scale verification tasks. In future work, we will focus on developing more intelligent methods to dynamically adjust the balance between these components.

[8] Notably, with only 3 cores (CAR-DT, AVY and IC3-REF), the combined performance already surpasses that of many candidates with 16 cores in HWMCC 2024.

References

1. CAR-DT artifact. https://doi.org/10.5281/zenodo.15165732.
2. 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.
3. 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.
4. 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.
5. 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.
6. Yibo Dong, Yu Chen, Jianwen Li, Geguang Pu, and Ofer Strichman. Revisiting assumptions ordering in CAR-based model checking (long version). https://doi.org/10.48550/arXiv.2411.00026.
7. Yibo Dong, Yu Chen, Jianwen Li, Geguang Pu, and Ofer Strichman. Revisiting assumptions ordering in car-based model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2025.
8. 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.
9. 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.
10. HWMCC 2024. https://hwmcc.github.io/2024/.
11. IC3Ref. https://github.com/arbrad/IC3ref.
12. iimc. https://github.com/mgudemann/iimc.
13. 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.
14. 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.
15. 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.
16. Supercar. https://github.com/lijwen2748/hwmcc24.
17. 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.