Accelerating CAR-based Model-Checking with Multiple Unsatisfiable Cores

Yibo Dong[1], Xiwei Wu[2], Jianwen Li[1⋆], Geguang Pu[1], and Ofer Strichman[3]

1. Software Engineering Institute, East Chine Normal University
2. Department of Computer Science and Engineering, Shanghai Jiao Tong University
3. Faculty of data and decision sciences, Technion

Abstract. Model checking is a framework for automated formal verification of transition systems, like hardware designs. Two leading model-checking techniques, PDR and CAR, are based on multiple calls to a SAT solver, for the purpose of finding a path to a bug or proving its absence. They build sequences of formulas, called *frames*, that represent approximated sets of states, e.g., a frame can contain an over-approximation of the states that can reach the negated property within a given number of steps. Part of the process is to make these sets more precise, i.e., less approximating, and this is done by strengthening the frames with negation of states that were proven to be spurious. A key component for performance is the ability of these engines to generalize those states and thus accelerate the process of making the frames more precise. This is done by strengthening them with the *unsatisfiable cores* that the SAT solver returns. In this work, we suggest several performance improvements to this process, most notably a technique for generating multiple such cores in linear time and adding them simultaneously to the frames. Our results show that our implementation of these techniques, on top of SimpleCAR, not only improves its performance, but also solves more (unsafe) cases than any other model-checker in the public domain, and solves instances from the HWMCC that no other model checker can solve, hence it contributes to the state-of-the-art.

1 Introduction

Model checking is an automatic formal-verification technique that is central in the hardware design community [3, 15]. Given a model M and a temporal property P over its variables, it checks whether all the behaviors of M satisfy P, i.e., whether $M \models P$. Once a system behavior 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

⋆ 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.

counterexample made of a finite number of states. It is well known that model checking on safety properties can be reduced to reachability analysis [8].

State-of-the-art safety model checking techniques include Bounded Model Checking (BMC) [4, 6], Interpolation Model Checking (IMC) [18], Property Directed Reachability (PDR) (also called IC3) [7,11], and Complementary Approximate Reachability (CAR) [17], 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 as such, 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 dominate the other. In [16, 17] it was empirically shown that within a given time and hardware resources, CAR is able to solve unsafe (i.e., instances in which the property fails) 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, hardware model-checking (a Pspace problem) always falls short of the performance needs in the industry when it comes to verifying large designs. Indeed, performance optimization of SAT-based model checkers is an active research area. Some recent examples are [27], [10] and [22].

In this paper, we focus on improving the performance of CAR. We will describe in detail how CAR works in Section 3. It has many similarities to PDR, which is better known, but also several distinctive features. For now, let us 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 key elements of this process is *generalization*, 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 that we report here is focused on finding multiple such UCs, which accelerates the narrowing process of the O-frames. To explain our contribution, let us first describe briefly 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 ϕ 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 [12] 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 analyzing 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 ϕ 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 [14], which implements the original IC3 algorithm, sorts the literals in $\mathcal{A}$ in descending order based on their appearance frequencies. The SimpleCAR model-checker, which implements CAR, uses two different literal-ordering strategies, as reported in [10]: *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 the performance of bug-finding. Indeed, SimpleCAR is one of the baseline implementations against which we compare our contributions.

The research that we report here, is focused on computing and adding more than one UC at a time, hence accelerating the narrowing of the O-frames. We prove that once the SAT solver proved that the formula is unsatisfiable, it is a *linear-time* operation to find multiple UCs, simply by rerunning the solver incrementally, with a different assumptions order. Since all the learned clauses remain from the initial run, it is guaranteed that the solver will detect unsatisfiability solely based on deciding the assumptions (or a subset thereof) and applying BCP. However, our experiments with variants of this approach demonstrated the difference between asymptotic complexity and actual run-time: although this linear-time operation is supposed to improve the search and thus lead to less exponential-time SAT solving, in practice the formulas given to the SAT solver in CAR are so easy that they are solved in fractions of a second. Thus, the trade-off between more cores and fewer SAT calls in practice is not at all an obvious win. We will describe several algorithmic steps that we took in order to make this technique cost-effective. Using this combination of techniques, we not only improved the average performance of SimpleCAR but also reached the point that it solves more unsafe cases from the HWMCC15 + HWMCC17 benchmark sets than any other model checker, and furthermore it can solve several cases that have never been solved before by any model checker, thus contributing to the state-of-the-art.[1]

We continue with preliminaries in the next section. In Section 3, we recall the CAR model checking algorithm. In Section 4, we present our new methods in detail, including experiments, and in Section 5, we compare our results to other tools. We conclude and suggest topics for future research in Section. 6.

[1] CAR with the improvements reported here has recently won the 3rd place in bit-level model-checking competition(safe + unsafe), and 2nd place in unsafe cases [13].

2 Preliminaries

2.1 Boolean Transition System

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

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., $t \models \neg P$, 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 a 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 I. Table 1 gives the corresponding formal definitions.

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	$F_i \cap \neg P \neq \emptyset$	$B_i \cap I \neq \emptyset$

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 and unsafe checking. If the condition in the safe/unsafe checking is satisfied, the search process terminates. Intuitively, if the unsafe checking passes, then some bad state is reachable, and if the safe checking passes, then all the reachable states from I have been checked, and none of them can reach a bad state. 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.

2.3 SAT Solving and Unsatisfiable Cores

In propositional logic, a *literal* is an atomic variable or its negation. A *cube* is a conjunction of literals, and a clause is a 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 ϕ 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 that a CNF formula ϕ is satisfiable if there exists an assignment of each Boolean variable in ϕ such that ϕ is true; otherwise, ϕ is unsatisfiable. Generally, it is an NP-complete problem to decide whether a given CNF formula is satisfiable. A SAT solver can decide whether a CNF formula ϕ is satisfiable or not. It emits a Boolean assignment to the variables, called a model of ϕ, if ϕ is satisfiable. Otherwise, some SAT solvers can emit an unsatisfiable core, as explained in the introduction, based on a subset of the assumptions.

3 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 [17]. 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 (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 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 [16, 17]).

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 describes CAR. It progresses by widening the U sets and narrowing the O sets, which are initialized 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}, initialized 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 transit to the O_l frame. This involves two steps. It first conducts a 'blockedIn' check (Line 11, to be discussed later) to check if the state is blocked at level $l+1$, i.e., whether $O_{l+1} \rightarrow \neg s$. If yes, backtrack is initiated; otherwise, CAR checks if $SAT(s, T \wedge O_l{}')$ in Line 18, i.e., whether $T \wedge O_l{}'$ is satisfiable while taking the literals in s as assumptions (recall from Sec. 2.1 that prime variables, such as O_l' here, denote next-state variables).

If yes, a new state $t \in O_l$ is extracted from the model, to update the U-sequence (Line 19-21), effectively *widening* it; Alternatively, the negation of the unsatisfiable core is used to constrain the O frame of s (level $l+1$), effectively *narrowing* it (Lines 23-30), and pushing s back to the stack if possible.

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 31, which was also mentioned as part of Definition 1.

Let us go back to '$blockedIn(s, l)$'. It can be thought of as an optimization – a linear-time test that saves calls to a SAT solver in line 18. It returns true if there exists a clause $cl \in O_l$ that subsumes (and hence implies) $\neg s$. Although each pair (s, l) should adhere to the condition that s is not blocked at level $l+1$ when the pair is pushed into the stack (line 21), the successors of s might later backtrack to a higher level, giving rise to a UC that blocks s. In the algorithm, if s is already blocked in O_{l+1}, then according to the definition, it cannot reach O_l within one step, and there is no need for the SAT call in line 18.

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 $U_0 \coloneqq I$, $O_0 \coloneqq \neg P$
3 **while** *true* **do**
4 $O_{tmp} \coloneqq \neg I$
5 **while** $state \coloneqq pickState(U)$ *is successful* **do** ▷ Heuristic choice
6 $stack \coloneqq \emptyset$
7 $stack.push(state, |O| - 1)$
8 **while** $|stack| \neq 0$ **do**
9 $(s, l) \coloneqq stack.top()$ ▷ Assume $s \in U_j$
10 **if** $l < 0$ **then return** 'Unsafe'
11 **if** $blockedIn(s, O_{l+1})$ **then** ▷ true if $O_{l+1} \rightarrow \neg s$
12 $stack.pop()$ ▷ note line 9
13 **while** $l + 1 < |O|$ ***and*** $blockedIn(s, O_{l+1})$ **do**
14 $l := l + 1$ ▷ Backtrack to a higher level
15 **if** $l + 1 < |O|$ **then**
16 $stack.push(s, l + 1)$
17 **continue**
18 **if** $SAT(s, T \wedge O'_l)$ **then**
19 $t \coloneqq GetModel()$
20 $U_{j+1} \coloneqq U_{j+1} \vee t$ ▷ Widening U. j is s's frame – see Line 9
21 $stack.push(t, l - 1)$
22 **else**
23 $stack.pop()$
24 $uc \coloneqq getUC()$
25 **if** $l + 1 < |O|$ **then**
26 $O_{l+1} \coloneqq O_{l+1} \wedge (\neg uc)$
27 $l' = minNotBlockedIn(s)$
28 $stack.push(s, l' - 1)$
29 **else**
30 $O_{tmp} \coloneqq O_{tmp} \wedge (\neg uc)$
31 **if** $\exists i \geq 1$ *s.t.* $(\bigcup_{0 \leq j \leq i} O_j) \supseteq O_{i+1}$ **then return** 'Safe'
32 Add a new state-set to O and initialize it to O_{tmp}

4 Adding multiple unsatisfiable cores

4.1 Theory and initial results

Recall that in CAR, the O frames are narrowed (i.e., become less overapproximating) by constraining them with negations of UCs — see line 30 in Alg. 1. We now suggest a method by which each time the formula in line 18 turns out to be unsatisfiable, more than one UC is computed and added. Constraining

further the frames in this way accelerates the narrowing of the O-frames and hence blocks more spurious states from being explored. It is guaranteed, by construction, that each new core is *not* implied by previous (negations of) UCs in the frame[2], as stated in the following theorem.

Theorem 1 (Additional UCs are not redundant). *Let (s, l) be a pair checked in line 9 of Alg. 1. If s is not blocked in O_{l+1} and $SAT(s, T \wedge O'_l)$ returns 'unsatisfiable', the UC retrieved from the SAT call, call it uc, satisfies $O_{l+1} \not\rightarrow \neg uc$.*

Proof. uc is a core of $SAT(s, T \wedge O'_l)$, hence $uc \subseteq s$, or, equivalently, $\neg uc \rightarrow \neg s$. Hence if $O_{l+1} \rightarrow \neg uc$, then $O_{l+1} \rightarrow \neg s$, which implies that s has already been blocked in O_{l+1} before the SAT call. A contradiction. □

We observe that once the SAT solver proved that the formula is unsatisfiable, it is a *linear-time* operation to find multiple UCs simply by rerunning the solver incrementally, with a different assumptions order. To be specific, since all the learned clauses remain from the initial run, it is guaranteed that the solver will detect unsatisfiability solely based on deciding the assumptions (or a subset thereof) and applying BCP. More formally:

Theorem 2 (The complexity of computing additional UCs is linear). *Let $\mathcal{A}$ be the assumptions (a vector of literals) and f be a formula. In a modern CDCL SAT Solver, if $SAT(f, \mathcal{A})$ is unsatisfiable, then a subsequent incremental call $SAT(f, reorder(\mathcal{A}))$, where reorder is some reordering of the assumptions, takes time which is linear in the number of f literals.*

Proof. In a Minisat-like SAT solver, the assumptions are selected as decision literals in the order in which the assumptions vector is given to the solver, followed each time by BCP. A formula is declared unsat when a conflict occurs in some decision level d such that $d \leq |\mathcal{A}|$. Let $uc \subseteq \mathcal{A}$ denote the UC of the original SAT call. In consequent incremental runs of the solver (recall that the checked formula includes all the learned clauses from the previous run that were still present, i.e., not deleted, at the time that the conflict was detected) with any order of assumptions, a conflict must occur at or before the point in which this subset is 'covered', hence before any real decision is made. Therefore, only decisions and BCP are applied, which is linear in the number of literals. □

Theorem 2 gives us hope that finding multiple UCs in a single run will be cost-effective. We call this method mUC, where m stands for multiple. Generally, we do not need to commit to a constant number of UCs, as we can heuristically decide how many to add. For exposition purposes, however, we will denote by mUC (n) a process of adding exactly n cores. mUC (1) is, therefore, the baseline of adding a single core.

[2] However, it is possible that the additional cores are subsumed by previous ones in the same formula. Indeed, we experimented with applying a subsumption test, but it turned out not to be cost-effective.

There is a large space for tuning this approach, including a decision on which order of assumptions to use in order to produce more UCs, how many UCs to add, whether always to apply it or just selectively, and so forth. We drove this tuning process with the 337 unsafe benchmarks [3] in the Aiger [5] format from the single safety property track of the 2015 and 2017 Hardware Model Checking Competition (HWMCC) [4] All the counterexamples found were successfully verified with the third-party tool AIGSIM that comes with the AIGER package.

Our initial experiments with variants of this approach demonstrated the difference between asymptotic complexity and actual run-time. Although this linear-time operation is supposed to improve the search and thus lead to fewer exponential-time SAT calls, in practice, the formulas given to the SAT solver in CAR are so easy that they are solved in fractions of a second. Thus, it is not obvious at all that more cores and fewer SAT calls in practice improve the total running time.

To test the actual difference in running time, we changed the algorithm. Each time the result of the SAT call was 'unsat', we ran it once more, incrementally, with the order of the literals reversed. This is not only cheap to compute, but also guarantees that the literals that have been picked in the previous run are in the back, which promotes diversity of the cores. We shall later show that this is a good choice empirically. Recall that this second run is linear according to Theorem 2, while the first run is worst-case exponential. Across all benchmarks, the ratio between the run times of the second and first runs turned out to be 0.65. In other words, if we always apply it, then each time the result of the SAT call is 'unsat', we pay a penalty of 65% in running time, with the hope that this will improve the search and consequently reduce the overall number of SAT calls and the total run-time. There are other overheads associated with this method: the frames can become bigger and hence slow down the SAT solver, and the check in line 11, which has a complexity linear in the size of the frame, can also be negatively affected. All these factors show that without careful tuning, this technique is not likely to succeed.

We implemented our suggested algorithms on top of SIMPLECAR [16, 23], which is an implementation of the CAR algorithm. The following table shows the effect of applying this technique on the total running time and the total number of SAT calls (in line 18). The timeout here and in the rest of the article was set to one hour.

[3] Results of these benchmarks are either known to be unsafe or remain unknown. Those that are already proven to be safe in the competition are excluded, as we will explain in Sec. 5.

[4] The focus on these relatively old benchmarks stems from the fact that since HWMCC 2019, the official format shifted to BTOR, which operates at the word level. Although some bit-level benchmarks were provided, they relied on a newer version of the Aiger format (> 1.9). However, not all the model checkers that we compared against support this format, and some emit wrong results. Consequently, we excluded those benchmarks and focused on HWMCC 2015 and HWMCC 2017 instead. We note that the most recent bit-level model checking article that was published to the best of our knowledge [26] also uses the 2015 and 2017 benchmarks.

Table 2: The effect of adding more UCs using different ordering of assumptions. The subscript in mUC_r indicates that the assumptions orderings for generating the additional cores were selected randomly.

	mUC (1)	mUC (2)	$\text{mUC}_r(2)$	$\text{mUC}_r(3)$	$\text{mUC}_r(4)$	$\text{mUC}_r(5)$
Solved	146	**150**	146	142	142	139
Avg. # of (first-only) SAT calls	137k	103k	146k	120k	107k	97k
Avg. total time of 1st SAT calls	1171.2s	944.0s	1138.7s	1014.0s	879.7s	827.9s
Avg. total time of subsequent SAT calls	0.0s	367.3s	741.4s	916.7s	1041.2s	1145.6s
Avg. UC redundant rate	0	4.53%	7.09%	8.97%	10.70%	11.15%

As we can see from Tab. 2, the number of cases solved by mUC (2) increased, which is a success. Furthermore, the number of (first) SAT calls and their accumulated run-time indeed drops, which implies that we are able to narrow down the over-approximation by adding more UCs, as expected. Note that the total SAT run-time (944+367.3 sec.) of mUC (2) is larger than that of the baseline mUC (1) – 1171.2 sec. On the other hand, the table shows that adding cores based on random assumptions ordering does not help. We will therefore continue from here on with mUC (2) based on the reverse order.

The results in the table may be misleading due to the time-out cases. For example, if a run with a single UC terminates on time and manages to make many fast SAT calls, while a run with two UCs times out and, due to overhead, completes fewer SAT calls before timing out, the table would count more calls under the single UC column. This could create the false impression that the single UC approach is less efficient than the two UCs approach. To address this, we also calculated the number of initial SAT calls after excluding 173 cases where at least one engine timed out:

	mUC (1)	mUC (2)
Avg. # of (first-only) SAT calls	2960	1747

We can see, then, that for this set of instances, the number of (first) SAT calls also drops. Notably this filtered view can also distort the full story, exactly because it ignores the cases in which one terminates and the other does not.

When comparing the overall runtime, the individual results of each benchmark instance in Fig. 1 show that most of the plots are above the diagonal, indicating a positive impact of adding the second UC.

4.2 Re-tuning the solver for multiple UCs

Selective application. To try to improve the method further, we experimented with various heuristics that control when to apply it. Specifically, in which frames to apply it: the high or low $x\%$ of the frames. For example, if we choose 30% from the low-index frames, it means that if the current number of O frames is n, then for frame i, $i \leq 0.3n$, we add multiple cores. Tab. 3 shows our results with different such values.

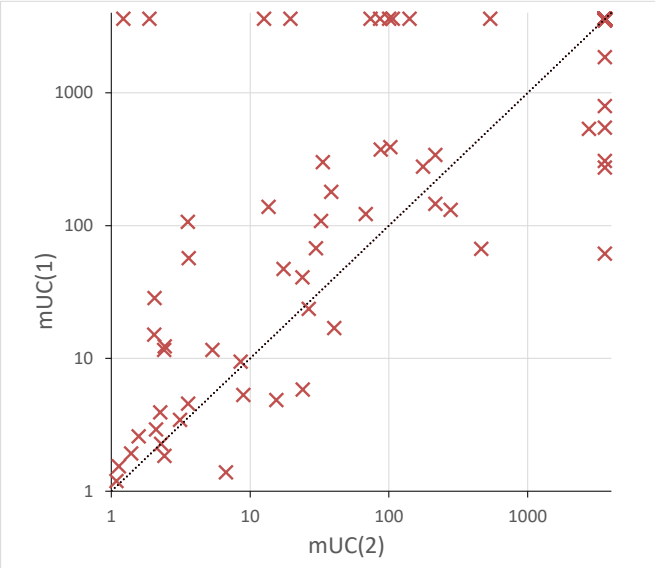

Fig. 1: Per-instance comparison of total time where X-axis and Y-axis represents mUC(2) and mUC(1) respectively.

Table 3: Different options to add multiple UCs

	mUC (1)	mUC (2)	Low-*				High-*			
			50%	33%	25%	20%	50%	33%	25%	20%
Solved	146	150	151	**152**	150	149	149	150	147	144

Hence, by adding a UC only for the 33% low-index frames we are able to solve two more cases.

Detecting blocked states efficiently. Next, through profiling we learned that a large part of the run-time is spent in line 11 of Algorithm 1, which tests whether the state s is blocked by the UCs in the frame O_{l+1}. Recall that this step is an optimization that saves SAT calls in line 18. The time spent in this line is proportional (i.e., linear) to the size of the frame, and hence adding more UCs to it potentially increases the overhead of this test. We measured the portion of running time dedicated to this process in the time-out cases. On average, it was 11% (407 out of 3600 sec). In 14% of the cases it was more than 30% of the running time. Alg. 2 shows the implementation of 'blockedIn'. It iterates the clauses in the corresponding frame and checks if one of them blocks the state, which amounts to a subsumption test.

Algorithm 2: The Basic Implementation of 'blockedIn'

Input: A state s, an O frame O_l
Output: 'BLOCKED' or 'NOT BLOCKED'

```
1 for each cl ∈ O_l do
2     if cl → ¬s then
3         return 'BLOCKED'
4 return 'NOT BLOCKED'
```

Going over all clauses in sequence might not be the best strategy. We experimented with using BCP instead, which amounts to calling a SAT solver over the formula O_l and the assumptions s. Since s spans the entire set of variables, this SAT call is determined without real decisions (i.e., only decisions of the assumptions themselves) and is hence linear. Furthermore, we maintain a separate solver object for each frame, and hence we activate this step incrementally, i.e., we do not need to read the frame into the SAT solver each time.

While Alg. 2 can be thought of as clause-driven DFS (checking each clause to completion each time), the BCP method can be thought of as literal-driven BFS. The 2-watch literal scheme [19] in modern SAT solvers makes it highly efficient, as most clauses are not even visited, so we expect this route to be faster, at least when the frames are large. Our experiments confirmed this hypothesis, which led us to a *hybrid* approach based on a threshold: we activate the BCP method only when the number of clauses in the frame is larger than 10k.

With this threshold, our results improved by three solved cases, to 155 — see Tab. 4. Two things to note in this table: first, the hybrid approach of invoking Alg. 2 for frames with 10k clauses or less, and BCP otherwise, improves the results regardless of the number of added UCs; Second, the time per call decreased by a factor of 2 (from 0.53 to 0.26) with the latest configuration, owing to this method.

Table 4: Results with different 'blockedIn' approaches. The 'Basic' rows refer to Alg. 2.

UC method	blockedIn method	Avg time per call (ms)	Solved
mUC (1)	Basic	0.56	146
	BCP	0.81	145
	Hybrid (10k)	0.4	148
mUC (2)	Basic	0.96	150
	BCP	0.85	150
	Hybrid (10k)	0.52	**153**
mUC (2) (Low 33%)	Basic	0.53	152
	BCP	0.58	153
	Hybrid (10k)	0.26	**155**

Restarts. Finally, we tested whether restarting periodically with different mUC strategies can help. Each such strategy likely leads to a different search. At each restart, we preserve some data from the previous run. It is important to note that we cannot maintain arbitrary portions of the frames, because it may break the invariants listed in Definition 1. For example, if we remove a clause from O_{k_o}, then we can no longer guarantee that the induction condition is maintained, namely that $O_{k+1} \supseteq R^{-1}(O_k)$. If we arbitrarily remove a state from a U frame, say U_k, we cannot guarantee that the invariant $U_{k+1} \subseteq R(U_k)$ is maintained, although this invariant is not necessary for the proof of correctness [17].

The only correct and computationally easy way that we found to remove data from the O sequence is to decide on an index k_o such that $O_0, \ldots, O_{k_o}$ are maintained whereas O frames with a larger index are removed. One exception that can be made to this rule is with the border frame, O_{k_o}, that can also be *partially* removed without violating the invariant, because after resetting O_{k_o+1} it is equal to *true* and trivially maintains the invariant. Similarly, for the U sequence, we can decide on an index k_u, maintain $U_0, \ldots, U_{k_u}$ and remove frames with an index larger than k_u.

We experimented with changing five dimensions:

1. The values of k_o, k_u, namely the border frames that define which data to maintain between restarts,
2. the portion of the border frames to maintain,
3. the clauses to be maintained in the border frames,
4. the selective application strategy (see the beginning of this subsection)
5. the time to restart.

The best configuration that we found is with $k_o = 1, k_u = 0$. As for the second question, we settled on maintaining at the nth restart $\frac{n}{n+1}$ of O_{k_o}. Hence, we begin with half of O_{k_o}, then two-thirds, and so on, namely we increase it from one restart to the next. Note that O_{k_o} likely changes each time. As for the third question, we sort O_{k_o}'s clauses and take the shortest ones. As for the fourth question, we begin with full activation and then begin to only activate at the low indices, where the activation threshold is updated at restart n, for $n > 1$, according to

$$threshold[n] = \frac{threshold[n-1]}{threshold[n-1] + 1} \tag{1}$$

Hence the activation ratio progresses as follows: $1, 1/2, 1/3, 1/4, \ldots$. Finally, as for the fifth question, we experimented with different time constants.

Table 5: The number of solved cases by mUC (2) with different restart strategies and restarting periods.

Restart Strategy	t=180s	t=300s	t=450s	t=600s	t=900s	t=1200s
$k_o = 0, k_u = 0$	155	157	160	156	155	160
$k_o = 1, k_u = 0$	154	157	157	159	161	158
$k_o = 1, k_u = 0$, forget parts of O_1	156	**164**	161	157	158	160

Tab. 5 summarizes some of our experiments with these dimensions. We can now solve more than 160 cases with many different configurations, peaking at 164 cases.

5 A comparison to other model checkers

Finally, we compared mUC to the leading model checkers in the public domain: ABC-BMC [1], ABC-PDR [1], NUXMV-IGOOD [21, 26], AVY [2, 24], NAVY [20, 25], BAC [27] and our baseline tool SIMPLECAR [17, 23]. Details about the configuration of these tools in our experiments appear in the appendix. The benchmark set is the same as described in Sec. 4.1.

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 number of CPU cores was limited to 1, the memory was limited to 8 GB, and the time to 1 hour.

Tab. 6 shows a summary of this comparison. The last column refers to mUC in its best configuration as described in the previous section. Fig. 2 and Tab. 7 give more details about the running time.

Table 6: The number of unsafe instances solved with a 1-hour timeout, by different model checkers.

	ABC-BMC	ABC-PDR	NUXMV-IGOOD	AVY	NAVY	BAC	SIMPLECAR	mUC
Solved	159	109	137	128	138	160	146	**164**

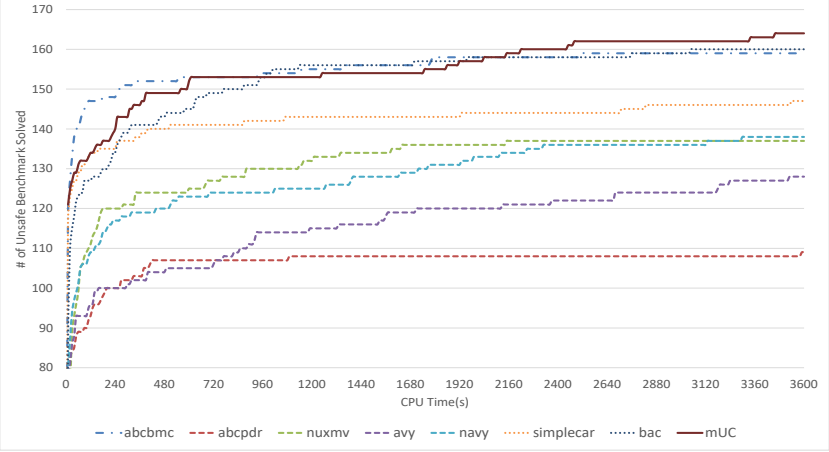

Fig. 2: Comparison of run-time performance among different model checkers (best seen in color).

Table 7: Comparing mUC to other model checkers in terms of the percentage of benchmarks that they each solve at least 5 seconds faster than the other.

	ABC-BMC	ABC-PDR	NUXMV-IGOOD	AVY	NAVY	BAC	SIMPLECAR
mUC is faster	34.0%	**63.4%**	**71.3%**	**58.2%**	**48.2%**	**61.2%**	**35.5%**
mUC is slower	**43.3%**	9.2%	13.3%	13.5%	20.6%	12.7%	14.0%

During our experiments with the various configurations of mUC, we were able to solve 7 benchmarks that were in an *unknown* status so far, that no *current version* of the tools in Tab. 6 can solve within the time limit, thus this can be seen as a contribution to the state-of-the-art. The depths of the counterexamples that we found, compared to the maximal depths that were proven safe by the virtual best solver of the other tools, appear in Tab. 8. We note that in some of the cases, the configuration at the left column is only one of several configurations that solved the benchmark.

Table 8: Benchmarks that only mUC can solve, albeit under different configurations (in some cases by multiple mUC configurations). The third column represents the maximum depth that other tools reached together (vbs), i.e., were able to prove that there is no counterexample up to that depth. The right column is the length of the counterexample that mUC found.

mUC (2) configuration	Benchmark	Bound	Counterexample length by mUC
low-20%	oski15a01b76s	14	143
low-25%	oski15a01b62s	16	95
low-33%	oski15a01b70s	18	125
low-33%	6s329rb20	47	1417
low-50%	intel012	100	1962
high-33%	6s329rb19	45	1295
full	oski15a01b36s	14	262

For completeness, we also report the results for safe cases in Tab. 9, despite the fact that CAR in general is not competitive for safety checking. The results show that our technique also improves the results for those cases — in particular from 166 to 177 solved instances.

To summarize, our experiments show that our implementation of these techniques, on top of SimpleCAR, not only improves its performance, but also solves more (unsafe) cases than any other model-checker in the public domain, and solves instances that no other model checker can.

Table 9: The number of safe instances solved (out of the 749 instances of HWMCC15-17) with a 1-hour timeout, by different model checkers.

	ABC-BMC	ABC-PDR	NUXMV-IGOOD	AVY	NAVY	BAC	SIMPLECAR	mUC
Solved	-	294	304	296	310	-	166	177

6 Conclusion

SAT-based model checkers like CAR and PDR rely heavily on unsatisfiable cores to generalize from failed attempts to progress from a given state. While previous work [10] focused on improving the core by controlling the literal order, here we suggested to add several cores (although in practice we found that just adding one more core, and even that only selectively, is best). Our initial core was already the best one (as a single core!) according to [10], hence our technique improves it. We discussed the many dimensions of this technique and presented our (limited) experience with tuning them, although there is plenty left for future research. For example, what are good literal orders for achieving high-quality cores in the context of adding multiple cores (we simply took the order suggested in [10] and its reversal for the 2nd core)? What is the best set of U frames and partial frames to leave when restarting? Can this method also improve PDR?

We showed that despite the fact that additional cores can be computed in linear time, and it indeed saves normal, worst-case exponential SAT runs, owing to the fact that the latter process is fed with relatively easy formulas, it is not at all trivial that the saving will be larger than the overhead of computing more cores and the overhead associated with having larger frames. We suggested a method for improving the linear-time test called 'blockedIn' (line 11 of Alg. 32) which initially was made slower by the increased frames, and also showed that restarting the solver with different strategies improves the run time.

At the bottom line, we were able to improve CAR from 146 to 164 solved cases. The next in line is BAC, which is also based on CAR, and after that the bounded model checker ABC-BMC, which can only solve 160 and 159 cases, respectively. Furthermore, we improved the state-of-the-art in the sense that we solved 7 cases that no other solver can. Such an improvement is significant in light of the many years of research on these tools.

All the artifacts described in this article are available at [9].

References

1. ABC: System for Sequential Logic Synthesis and Formal Verification, https://github.com/berkeley-abc/abc
2. AVY, https://arieg.bitbucket.io/avy/
3. Bernardini, A., Ecker, W., Schlichtmann, U.: Where formal verification can help in functional safety analysis. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 1–8 (2016). https://doi.org/10.1145/2966986.2980087
4. Biere, A., Cimatti, A., Clarke, E., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of Design Automation Conference (DAC). pp. 317–320 (1999). https://doi.org/10.1109/DAC.1999.781333
5. Biere, A.: AIGER Format. http://fmv.jku.at/aiger/FORMAT
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
7. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992). https://doi.org/https://doi.org/10.1016/0890-5401(92)90017-A
9. Dong, Y.: muc-artifact (Apr 2025). https://doi.org/10.5281/zenodo.15164945
10. Dureja, R., Li, J., Pu, G., Vardi, M.Y., Rozier, K.Y.: Intersection and rotation of assumption literals boosts bug-finding. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 180–192. Springer International Publishing, Cham (2020)
11. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, Texas (2011)
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
13. HWMCC 2024. https://hwmcc.github.io/2024/
14. IC3Ref. https://github.com/arbrad/IC3ref
15. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) **41**(4), 1–54 (2009). https://doi.org/10.1145/1592434.1592438
16. Li, J., Dureja, R., Pu, G., Rozier, K.Y., Vardi, M.Y.: Simplecar: An efficient bug-finding tool based on approximate reachability. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 37–44. Springer International Publishing, Cham (2018)
17. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 95–100 (2017). https://doi.org/10.1109/ICCAD.2017.8203765
18. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification, pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

19. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. Design Automation Conference (DAC'01) (2001)
20. NAVY, https://bitbucket.org/arieg/extavy/downloads/fib_cav15.tar.gz
21. nuXmv with i-good Lemmas, https://github.com/youyusama/i-Good_Lemmas_MC
22. Seufert, T., Scholl, C., Chandrasekharan, A., Reimer, S., Welp, T.: Making progress in property directed reachability. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 355–377. Springer International Publishing, Cham (2022)
23. SimpleCAR. https://github.com/lijwen2748/simplecar/releases/tag/v0.1
24. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: International Conference on Computer Aided Verification. pp. 260–276. Springer (2014)
25. Vizel, Y., Gurfinkel, A., Malik, S.: Fast interpolating bmc. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27. pp. 641–657. Springer (2015)
26. Xia, Y., Becchi, A., Cimatti, A., Griggio, A., Li, J., Pu, G.: Searching for i-good lemmas to accelerate safety model checking. In: International Conference on Computer Aided Verification. pp. 288–308. Springer (2023)
27. Zhang, X., Xiao, S., Li, J., Pu, G., Strichman, O.: 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, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3508352.3549393, https://doi.org/10.1145/3508352.3549393

A The configurations of the competing model checkers

The table below gives more details about the configuration of each of the tools. Unless otherwise stated, these flags represent the default. In the case of AVY and NAVY the flags were copied from the corresponding scripts in their respective repository.

Tool	Configuration Flags
ABC-BMC	-c "bmc2"
ABC-PDR	-c "pdr"
SIMPLECAR	-b -e
BAC	-b -e -1500
AVY	–reset-cover=1 -a –kstep=1 –shallow-push=1 –tr0=1 –min-suffix=1 –glucose –glucose-inc-mode=0 –min-core=1 –glucose_itp=1 –stick-error=1 –sat-simp=1
NAVY	–reset-cover=1 –opt-bmc –kstep=1 –shallow-push=1 –min-suffix=1 –glucose –glucose-inc-mode=0 –sat-simp=1 –glucose_itp=1
NUXMV-IGOOD	-a ic3 -s cadical -W -m 1 -u 4 -I 1 -D 0 -g 1 -X 0 -c 0 -p 1 -d 2 -G 1 -P 1 -A 100 -O 3
MUC	–vb –inter 1 –rotate –raw –imp 5 –rem 1 –convMode 1 –convParam 0 –restart 300