Combining BMC and Complementary Approximate Reachability to Accelerate Bug-Finding

Xiaoyu Zhang[1], Shengping Xiao[1], Jianwen Li[1*], Geguang Pu[1,2], Ofer Strichman[3]
[1]Software Engineering Institute, East China Normal University [2]Shanghai Trusted Industrial Control Platform Co., Ltd
[3]Information System Engineering, IE, Technion

ABSTRACT

Bounded Model Checking (BMC) is so far considered as the best engine for bug-finding in hardware model checking. Given a bound K, BMC can detect if there is a counterexample to a given temporal property within K steps from the initial state, thus performing a global-style search. Recently, a SAT-based model-checking technique called Complementary Approximate Reachability (CAR) was shown to be complementary to BMC, in the sense that frequently they can solve instances that the other technique cannot, within the same time limit. CAR detects a counterexample gradually with the guidance of an over-approximating state sequence, and performs a local-style search. In this paper, we consider three different ways to combine BMC and CAR. Our experiments show that they all outperform BMC and CAR on their own, and solve instances that cannot be solved by these two techniques. Our findings are based on a comprehensive experimental evaluation using the benchmarks of two hardware model checking competitions.

1 INTRODUCTION

Model checking [5, 6] is an automatic technique for formal verification of hardware and software designs, see e.g., [10] and [7]. Given a formal model M and a temporal property P, model checking answers the question whether $M \models P$, i.e., whether all the state sequences in M that begin from an initial state satisfy P. If the answer is negative then a counterexample to the property, namely a bug, is detected by the checker and presented to the user. If P is restricted to be a *safety* property [22], the counterexample is a loop-free state sequence starting at the initial state and ending with a state that violates P. In this paper, we only focus on such properties.

A bit of history: While explicit-state model checking of a safety formula was used in AT&T already in the 1980's [21, 23], it was the introduction of symbolic model checking [15, 27] based on Bryant's OBDDs [14] in the early 1990's that pushed several EDA companies to develop this technique, initially for internal use and later as products. OBDDs can require exponential space in the number of inputs, which limits the size of verified models. A major breakthrough was made with the introduction of SAT-based bounded model checking – BMC – in 1999 [8, 9]. SAT [18], unlike BDDs, do not suffer from the same practical memory bottleneck of BDDs , and can be remarkably fast in practice. On the other hand, BMC in its basic form is not complete - it can only detect bugs up to a given bound, and cannot prove that they do not exist in deeper cycles. Since BMC is based on unrolling the transition relation, the size as well as the number of variables in the CNF formula grow linearly with the bound k, which limits the capability of BMC to finding relatively shallow bugs.

There are also SAT-based model-checking techniques that *are* complete, most notably Interpolation-based model checking (IMC) [26], IC3/PDR [12, 17, 19] and CAR [16, 24, 25]. IC3/PDR and CAR do not rely on unrolling and are hence potentially better at detecting deep bugs. These techniques produce a much larger number of SAT queries compared to BMC, but each of them is based on a single copy of the transition relation, which is solved very fast by modern SAT solvers (typically thousands of such queries can be solved in a second). From previous evaluation studies, BMC and CAR perform better when it comes to bug-finding than IMC and IC3/PDR [24]. In the EDA industry, typically BMC has the best performance as a bug-finder, comparing to the complete model-checking techniques[1]. CAR is rather recent and was not experimented with in the industry.

Notably, there are several remarkable improvements on IC3/PDR such as Avy [29, 30] and QUIP [20], to name a few. However, they are not considered in this article because they cannot perform better on bug-finding than BMC and CAR based on a previous evaluation [16, 24].

CAR is able to complement BMC by solving instances that cannot be solved by the latter within a given time limit, though it does not have a competitive overall performance as BMC in bug detection. A model checker that is complementary to another one is important in a parallel portfolio setting (i.e., running several engines in parallel), which is a common practice in the industry. The natural question that we address here, is whether it is possible to find a faster bug-finding algorithm, by inheriting the advantages of both of these techniques. This paper addresses this question, by presenting three integration techniques of BMC and CAR. We will describe CAR in detail in the next section. For now it suffices to say that it is similar to the more well-known IC3/PDR technique in the sense that it maintains symbolically sets of states that are being updated with easy-to-solve SAT queries. Specifically it maintains a sequence of state sets U that underapproximate reachable states, and a separate sequence O of state sets that overapproximate the states that can reach a bad state, i.e., a state that satisfies $\neg P$. It searches a U state that can reach an O state, and then attempts to progress from the O state all the way to $\neg P$. CAR uses a backtracking mechanism, which iteratively widens the U sets and narrows the O sets, until either a path to $\neg P$ is found (i.e., a counterexample), or it is proven that no prefix from an initial state can be extended to such a path.

The first integration techniques that we will present is called BICAR, short for CAR-aided BMC. In the first stage of BICAR, we run BMC up to a given time limit (alternatively, as long as it makes progress with the bound). Suppose that no counterexample was found within this limit, and that the last bound that it was able to prove safe is k. In each of the k unsatisfiable SAT queries we save the unsatisfiable core. In the second phase of BICAR, we use the

* Jianwen Li is the corresponding author (lijwen2748@gmail.com).

[1]Based on a discussion with industrial partners.

negation of these cores to initialize the overapproximating sequence of CAR, and then run CAR.

The second integration technique is called BAC, short for BMC-aided CAR. It leverages BMC to try and escape from regions in the search space in which CAR is struggling to make progress. The original CAR uses a depth-first strategy to find new states. We learned that in most instances that cannot be solved by CAR, it is because it 'gets stuck' while backtracking within a sub-sequence of O, say from O_j to O_i, for $j \geq i \geq 0$ (CAR attempts to progress from a large index set to a small index set, where $O_0 = \neg P$). Inspired by this observation, BAC identifies such scenarios by monitoring the search depth that it already reached (i in this case) and then invokes BMC to check whether a state s in O_j has a successor in O_{i-1} (i.e., one frame closer to $\neg P$) in $j - i + 1$ steps, namely by solving

$$s^{(0)} \wedge \left(\bigwedge_{0 \leq m \leq j-i)} T^{(m)}\right) \wedge O_{i-1}^{(j-i+1)} . \quad (1)$$

The superscripts represent the cycle index to which the predicate's variables are renamed, e.g., $T^{(m)}$ is the transition relation where the variables are renamed to those that represent cycles $m, m + 1$.

The last integration technique is called K-CAR, a simple variant of CAR that allows the given state to find its successors in up to k steps. The original CAR only uses $k = 1$, hence the main SAT query input has the form of $s \wedge T \wedge O_i'$, where O_i' represents the next-state version of O_i. If this query is satisfiable, a successor of s in O_i can be computed from the satisfying assignment. Unlike CAR, K-CAR checks whether a successor of s in k (≥ 1) steps is in O_i, i.e., the corresponding query is generalized to

$$\bigvee_{1 \leq j \leq k} \left(s^{(0)} \wedge \left(\bigwedge_{0 \leq m \leq j-1} T^{(m)}\right) \wedge O_i^{(j)}\right) . \quad (2)$$

It should be highlighted that our suggested algorithms only work as bug-finders, i.e., they are not complete.

We conducted a comprehensive experimental evaluation of our implementation of these algorithms inside SimpleCAR [24], a tool that implements CAR. For this we also implemented BMC inside SimpleCAR. Hence these implementation of CAR and BMC served as our baseline for the comparison. As benchmarks, we took the instances from the hardware model checking competitions in 2015 and 2017 that have a failing safety property. Our results show that all three algorithms are able to outperform both CAR and BMC, and solve instances that cannot be solved by both. Our implementation of BMC inside SimpleCAR is rather naive, so we also compared our results to the state-of-the-art BMC solver inside ABC [13] (henceforth, ABC-BMC). It turns out that from our three algorithms only BAC can outperform it, but all three algorithms are able to solve instances that it cannot within a given timeout. In the future we intend to integrate ABC-BMC inside SimpleCAR to enjoy its superior implementation of BMC.

We continue in the next section with preliminaries. Sec. 3 presents the three new bug-finding algorithms in detail. Sec. 4 shows the experimental results. Finally, Sec. 5 summarizes the contributions and discusses future work.

2 PRELIMINARIES

2.1 Notation and SAT Solving

We assume the reader is familiar with standard propositional logic terminology and SAT. For each variable x, there exists a corresponding variable x' (the *primed version* of x). If V is a set of variables, V' is the set obtained by replacing each element in V with its primed version. Given a formula φ, φ' is the formula obtained by replacing each variable occurring in φ with the corresponding primed variable, whereas $\varphi^{(i)}$ denotes the formula obtained by i consecutive applications of priming, i.e., $\varphi^{(0)} = \varphi$ and $\varphi^{(i)} = (\varphi^{(i-1)})'$. Given two formulas φ and ψ, we denote with $\varphi \models \psi$ if all the models of φ are also models of ψ.

A SAT *solver* is a procedure for deciding the satisfiability of a given propositional formula φ in Conjunctive Normal Form (CNF), i.e., it returns *true* iff φ has at least one model. We will assume that our SAT solver supports at least the following API:

- IsSat(ϕ) checks the satisfiability of the input formula ϕ;
- SatAssume($assumptions, \phi$) checks the satisfiability of ϕ under the given additional assumptions (a list of literals) $assumptions$, as in [3, 28]
- GetModel() retrieves the model computed by a previous SatAssume() or a IsSat() call, and GetModel()$|_V$ returns the projection of that model to V;
- GetUnsatAssumptions() retrieves an unsatisfiable core (UC) of the assumption literals of the previous SatAssume(*assumptions*, ϕ) call, i.e., a subset of the assumption literals that is enough to make the formula unsatisfiable.

2.2 Safety Model Checking

Definition 1 (Boolean Transition System). *A Boolean transition system is a tuple (V, I, T) where*

- *V and V' denote the set of variables in the present state and the next state, respectively;*
- *I is a propositional formula corresponding to the set of initial states;*
- *T is a propositional formula over $V \cup V'$ for the transition relation.*

Given a transition system $Sys = (V, I, T)$, its state space is the set of possible variable assignments, which is a subset of 2^V. Also, a state of Sys is a cube over V. A *path* of length n in system Sys is a finite state sequence $s_1, \ldots, s_n$ such that $s_1 \in I$ and $s_i \cup s_{i+1}' \models T$ holds for $0 \leq i \leq n-1$. Let $X \subseteq 2^V$ be a set of states in Sys. The set of X's successor states is defined as $R(X) = \{t \mid (s \cup t') \models T, s \in X\}$. Conversely, the set of predecessors of states in X is defined as $R^{(-1)}(X) = \{s \mid (s \cup t') \models T, t' \in X'\}$. 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. Intuitively, $R^i(X)$ denotes the states reachable from X in i steps, and $R^{-i}(X)$ denotes the states that can reach X in i steps.

Given a transition system $Sys = (V, I, T)$ and a safety property P, 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*, which is a finite path from one of the initial states to a state violating P. In the former case, we say that the system is *safe*, and in the latter case, we say that the system is *unsafe*.

In this article, we frequently use a Boolean formula φ to represent symbolically the set of states $S = \{s \mid s \models \varphi\}$, and a set of states S to represent the Boolean formula $\bigvee_{s \in S} s$.

2.3 BMC **and** CAR

2.3.1 Bounded Model Checking. (BMC) is a well-known incomplete model checking algorithm. Given a bound K, BMC tries to find a counterexample whose length is K (alternatively, up to K), by a reduction to a sequence of SAT queries. More specifically, given a transition system $Sys = (V, I, T)$ and a safety property P, BMC calls a SAT solver with the formula

$$I^{(0)} \wedge \bigwedge_{0 \leq i < K} T^{(i)} \wedge (\neg P^{(K)}) \ . \tag{3}$$

If this formula is satisfiable, the corresponding model represents a counterexample to the property P. Otherwise, the user may decide to increase K and retry – the only limit being the run time. BMC is famous for its efficiency on bug-finding, especially if the bug is relatively shallow.

2.3.2 Complementary Approximate Reachability. (CAR), is a relatively new SAT-based safety model checking approach that is essentially a reachability-analysis algorithm, inspired by IC3/PDR [25]. Unlike BMC, 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 2 (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 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* $\exists i \geq 1 \cdot (\bigcup_{j=0}^{i} O_j) \supseteq O_{i+1}$.

We note that 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. However, in this article we will only use the direction as stated in Definition 2 (this was called 'backward CAR' in [24]).

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.

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 IsSat(I ∧ ¬P) then return "Unsafe"
 2 U_0 := I, O_0 := ¬P
 3 while true do
 4 |  O_tmp := ¬I
 5 |  while state := PickState(U) is successful do
 6 |  |  stack := ∅
 7 |  |  stack.push(state, |O| − 1)
 8 |  |  while |stack| ≠ 0 do
 9 |  |  |  (s, l) := stack.top()
10 |  |  |  if l < 0 then return "Unsafe"
11 |  |  |  if SatAssume(s, T ∧ O_l') then // Assume s ∈ U_j
12 |  |  |  |  t^(1) := GetModel()|_V'
13 |  |  |  |  U_{j+1} := U_{j+1} ∪ t^(1)        // Widening U
14 |  |  |  |  stack.push(t, l − 1)
15 |  |  |  else
16 |  |  |  |  stack.pop()
17 |  |  |  |  uc := GetUnsatAssumptions()
18 |  |  |  |  if (l + 1 < |O|) then O_{l+1} := O_{l+1} ∧ (¬uc)
19 |  |  |  |  else O_tmp := O_tmp ∧ (¬uc)
20 |  |  |  |  while l + 1 < |O| and s ∉ O_{l+1} do l := l + 1
21 |  |  |  |  if l + 1 < |O| then stack.push(s, l + 1)
22 |  if ∃i ≥ 1 s.t. (⋃_{0≤j≤i} O_j) ⊇ O_{i+1} then return "Safe"
23 |  Add a new state-set to O and initialize it to O_tmp
```

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 is done by checking if $s \wedge T \wedge O_l{}'$ is satisfiable (Line 11). If yes, a new state $t \in O_l$ is extracted from the model, to update the U-sequence (Line 12-14), effectively *widening* it; Otherwise, the negation of the unsatisfiable core is used to constrain the O frame of s (level $l+1$), effectively *narrowing* it (Lines 16-18), and pushing s back to the stack. In Line 20 CAR skips frames already block s.

Fig. 1 shows a high-level depiction of the search in lines 5-23 of Algorithm 1. In this figure and those after

$$\varphi_i = \neg O_0 \wedge \neg O_1 \wedge \ldots \wedge \neg O_i \ , \tag{4}$$

hence φ_i represents an underapproximating set of states that cannot reach $\neg P$ within i steps. Obviously, $\varphi_i \supseteq \varphi_{i+1}$ and

$$s \in \varphi_i \setminus \varphi_{i+1} \quad \Rightarrow \quad s \in O_{i+1} \ . \tag{5}$$

Note that narrowing O_{i+1} by blocking s is equivalent to moving s from φ_i to φ_{i+1}. Also note that while the φ sequence is convenient for understanding CAR, CAR does not compute it explicitly.

At Line 5, every state $s \in U$ is guaranteed to be in some φ_i, for $0 \leq i \leq n$, where $n = |O| - 1$. Starting from some state in φ_n, CAR searches a new state that is in φ_{n-1} but not in φ_n. If it succeeds, the

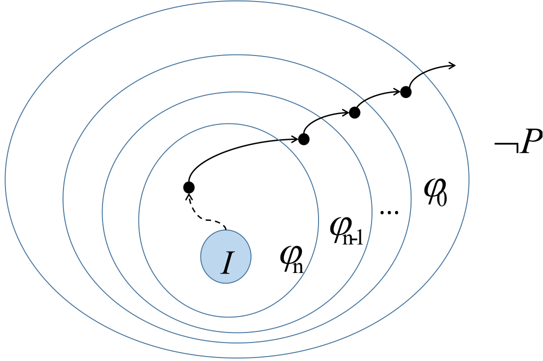

Figure 1: A high-level depiction of the search in CAR (Algorithm 1). At each step it attempts to transit to a lower O frame, which, according to (4), is the same as transiting to a lower φ layer, towards $\neg P$, as shown here. This search process involves backtracking (not shown here), and is restricted to states in the U sequence, namely states that are reachable from I.

process is repeated from that state, the stopping condition being that we reached $\neg P$ (line 10). Otherwise, a state s in φ_i may be moved to φ_{i+1} (or even higher because of line 20) because of the narrowing of the O-sequence (see (5) and the discussion that follows it). Then, a successor to s in φ_{i+1} will be searched (Line 21), and if successful the process continues based on the new successor. We call this step *backtracking*. Lastly, CAR returns "Safe" if the O sequence includes all the states that can reach $\neg P$ – this is checked via the condition in line 22, which was also mentioned as part of Definition 2.

Although CAR is prior work, and its correctness was proven in [4], we prove here completeness, to show later on why our modifications break this proof and make the algorithm incomplete.

Proposition 1 (Completeness). *If $Sys \not\models P$, then* CAR *returns "unsafe".*

Proof. Note that the alternative, namely that CAR returns "safe", can only occur in line 22, hence we should prove that this does not happen in the presence of a counterexample. Let $s_0, ..., s_n$ be the shortest counterexample ($n+1$ states). We will focus in this proof on the case that CAR reaches line 22 with $|U| + |O| \leq n$, because otherwise it is not hard to see that CAR will find the counterexample. When CAR reaches line 22, there is no 1-step transition from U to O. Let $|O| = t$, for $0 \leq t \leq n$. To simplify the rest of the proof, we will consider concrete numbers, $n = 10$ and $t = 3$, without loss of generality. We know that

- the suffix s_8, s_9, s_{10} is contained in O_2, O_1, O_0, respectively (by definition of the O sequence), and that
- the U sequence does not contain s_7 (otherwise there would be a transition to O_2).

Let s_k, $0 < k < 8$ be the state in the counterexample with the highest index such that s_k is not in the O sequence (recall that because the O sequences are overapproximating, they can contain more than just the above-mentioned $s_8...s_{10}$). There must be such an index k, because otherwise the U sequence, which includes the initial state I, would be able to reach the O sequence in one step — a contradiction to our observation above that there is no transition between the U and O sequences at this point.

If the condition in line 22 holds, then $O_2 \subseteq O_1 \cup O_0$, and so we have

$$\begin{aligned} & s_k \in R^{-1}(O_0 \cup O_1 \cup O_2) \wedge O_2 \subseteq O_1 \cup O_0 \Rightarrow \\ & s_k \in R^{-1}(O_1 \cup O_0) \Rightarrow s_k \in (O_2 \cup O_1)\ , \end{aligned} \tag{6}$$

which contradicts our assumption that s_k is not in the O sequence. Hence, the condition in line 22 does not hold, which implies that CAR returns "unsafe". □

Note that the last implication in (6) is correct because of the induction condition of the O sequence, namely $O_{k+1} \supseteq R^{-1}(O_k)$, for $k \in [0..|O|-1]$. We will see later in the article that this condition no longer holds with our new methods, which breaks our proof above and forces us to give up completeness. Subsection 3.4 will be dedicated to this issue.

3 THREE WAYS TO COMBINE BMC AND CAR

In this section, we present three algorithms that we experimented with, which can be seen as integrations of BMC and CAR.

3.1 BICAR: BMC-Initialized CAR

Recall from Sec. 2.3 that given a bound $k \geq 1$, BMC checks if there exists a counterexample of length k by solving the query IsSat($I \wedge T_k \wedge \neg P$). If the query is satisfiable, a counterexample can be extracted from the satisfying assignment and the algorithm terminates. Otherwise, k is increased by one, and the procedure repeats. In the latter case, an unsatisfiable core UC can be obtained from the SAT solver, which is not used in standard BMC but can be used to initialize the O-sequence of CAR and hence potentially make it run faster. More specifically, the UC encompasses the reason that the initial state cannot reach $\neg P$ in k steps. Therefore, the negation of the UC is an *over-approximation* of the states that *can* reach $\neg P$, and we can leverage it in CAR to initialize the O-sequence. In this way, CAR can be seen as an aid to BMC.

There is another way in which the initial BMC run can accelerate CAR. If it times-out after proving that there is no counterexample up to depth k, then CAR can be invoked by calling IsSat($I \wedge T \wedge O_k$), in contrast to CAR which invokes IsSat($I \wedge T \wedge O_0$). The reason is that BMC already proved that a counterexample, if there is one, is at least of length $k+1$.

The implementation of BICAR is described in Algorithm 2. It takes a transition system $Sys = (V, I, T)$ and a safety property P as the inputs. At Line 1, BICAR checks whether the set of initial states intersects with the bad states. Then the over- and under-approximating sequences of CAR as well as the bound k in BMC are initialized (Line 2). In the main loop (Lines 4-11), it checks whether I can reach $\neg P$ in k steps (Line 5). If not, a UC is obtained from the SAT solver (Line 7) and its negation is assigned to the corresponding frame O_k of CAR (Line 8). If BMC's run-time exceeds the predefined time constraint *Max_time*, the loop breaks (Line 10) and CARcheck() is invoked (Line 12). As mentioned above, CARcheck()starts by checking whether I can reach O_{n-1}.

Algorithm 2: BICAR: BMC-Initialized CAR

```
Input: A transition system Sys = (V, I, T) and a safety
       property P
Output: if terminates, "Unsafe" + a counterexample
1  if IsSat(I ∧ ¬P) then return "Unsafe"
2  U_0 := I, O_0 := ¬P, k := 1
3  begin := clock()
4  while true do
5      if SatAssume(I, ⋀_{m≤k−1} T^(m) ∧ ¬P^(k)) then return
        "Unsafe"
6      else
7          uc := GetUnsatAssumptions()
8          O_k := ¬uc
9          bmc_time := clock() − begin
10         if bmc_time ≥ Max_time then break
11     k++
12 CARcheck()
```

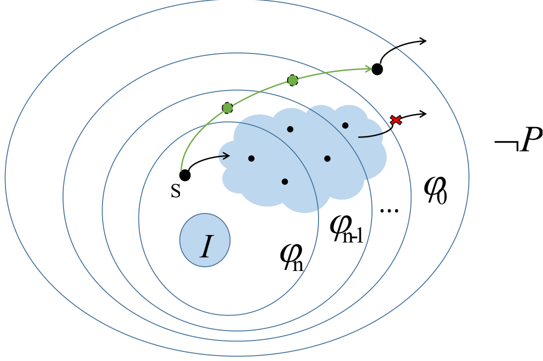

Figure 2: BAC: BMC-**aided** CAR. **Starting from** $s \in U$**, which is also in** φ_n**, the 'cloud' in the figure represents a set of states that** CAR **is struggling to get out of. The green path illustrates that invoking** BMC **at this stage can find a path to some state in** φ_0**, bypassing the search in the 'cloud'.**

3.2 BAC: BMC as an Aid for CAR

In Algorithm 1, after a state $s \in U$, which is also in φ_n, is selected at Line 5, CAR checks whether s can reach a successor in O_n by calling the SAT query $\textsc{IsSat}(t \wedge T \wedge O'_n)$, which essentially achieves the transition from φ_n to φ_{n-1}. If the query returns satisfiable, a new state $t \in \varphi_{n-1}$ ($\notin \varphi_n$) is obtained and then added to the U-sequence. Afterwards, CAR checks if t can transit to a certain state in φ_{n-2} by calling the SAT query $\textsc{IsSat}(t \wedge T \wedge O'_{n-1})$. In this procedure, CAR tries to find a path from s to $\neg P$ by searching for states such that they can transit out of the $\{\varphi\}$ frame to which they belong. In principle, a counterexample, if exists, has to contain at least one state that belongs to each φ_i for $0 \leq i \leq n$. Therefore, a path that cannot reach a certain φ_i, cannot become a counterexample. Continuing the search on such paths can only waste time in CAR.

CAR's search is based on backtracking upon failure while narrowing the O- and widening U- sequences. It is frequently the case that this backtracking process gets 'stuck', i.e., all the states among the path fall into the regions between φ_j and φ_i ($0 \leq i \leq j \leq |O|-1$), which is depicted as a 'cloud' in Fig. 2. Although states in the cloud region can be fully explored in principle, it can be very time consuming. A better search strategy may be to attempt to escape this region by unrolling the transition relation several times, like in BMC.

Algorithm 3: BAC: BMC-aided CAR

```
Input: A transition system Sys = (V, I, T) and a safety
       property P
Output: if terminates, "Unsafe" + a counterexample
1  if IsSat(I ∧ ¬P) then return "Unsafe"
2  U_0 := I, O_0 := ¬P
3  while true do
4      state := PickState(U)
5      O_tmp := ¬P
6      stack := ∅; stack.push(state, |O| − 1, 1)
7      count_states := 0
8      level := |O| − 1
9      while |stack| ≠ 0 do
10         (s, l, u) := stack.top()
11         level := min(l, level)
12         if count_states ≥ Max_n then
13             stack.clear()
14             unroll := |O| − level + 1
15             stack.push(state, level − 1, unroll)
16         else count_states++
17         if l < 0 then return "Unsafe"
18         if SatAssume(s^(0), ⋀_{0≤m≤u−1} T^(m) ∧ O_l^(u)) then
19             M := GetModel()
20             for i = 1 ... u do
21                 t^(i) := M|_{V^(i)}
22                 U_{j+i} := U_{j+i} ∪ t
23                 stack.push((t^(i), l + u − i − 1, 1))
24         else
25             stack.pop()
26             uc := GetUnsatAssumptions()
27             if (l + 1 < |O|) then O_{l+1} := O_{l+1} ∧ (¬uc)
28             else O_tmp := O_tmp ∧ (¬uc)
29             if l + 1 < |O| then stack.push(s, l + 1, u)
30     Add a new state-set to O and initialize it to O_tmp
```

Specifically, we can check if s can reach φ_{i-1} in k steps, while bypassing the states between them, by checking the satisfiability of

$$s^{(0)} \wedge \bigwedge_{0 \leq m \leq j-i} T^{(m)} \wedge O_i^{(j-i+1)} . \quad (7)$$

If the query is satisfiable, a list of new states can be extracted from the satisfiable assignment, corresponding to the path from s to O_i (see Line 20 in Algorithm 3). Otherwise, s can be blocked by adding the corresponding UC to the O-sequence and then CAR can switch to another state in φ_j.

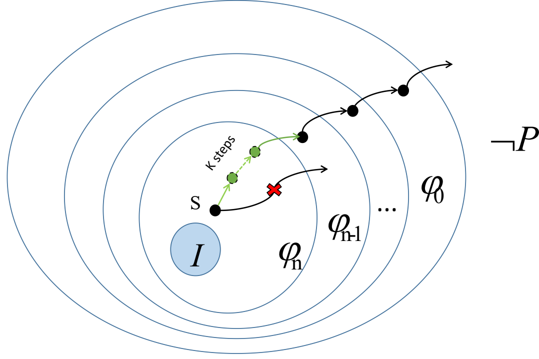

Figure 3: K-CAR: Computing the k-step successor inside CAR. Starting from $s \in U$, which is also in φ_n, once a one-step successor in φ_{n-1} cannot be found by CAR, BMC is then invoked to check whether a $k(k \geq 2)$-step successor can be detected in φ_{n-1} (represented by the green path).

As shown in Algorithm 3, BAC has several differences from CAR. The stack is used to store 3-tuples: in addition to the state and the O-sequence frame level as in CAR, it also stores the unrolling level. BAC uses two variables, $count_states$ and $level$, to respectively record the number of searched states (Line 7) and the smallest O-sequence frame level that has ever been reached (Line 8). When the maximal states number is reached, i.e., $count_states \geq Max_n$ (Line 12), BAC clears the stack and pushes a new triple $(state, level - 1, unroll)$ into the stack (Lines 13-15), hence searching for a successor in $unroll$ steps, where $unroll = |O| - level + 1$.

3.3 K-CAR: Computing the k-step Successor inside CAR

CAR invokes a SAT solver to check if there is a one-step successor of s in a certain O_n, i.e., the corresponding query is $\textsc{IsSat}(s \wedge T \wedge O'_n)$. If the SAT solver returns 'satisfiable', a new state $t \in O_n$ is extracted and then CAR recursively checks if t can reach some state in O_{n-1} in one step. However, if a certain SAT query is unsatisfiable, CAR backtracks to the previous state or picks another state from the U-sequence. For example, if $\textsc{IsSat}(t \wedge T \wedge O'_{n-1})$ is unsatisfiable, then the negation of uc, where $uc \subseteq t$, is added to O_n and CAR backtracks to check if $\textsc{IsSat}(s \wedge T \wedge O'_n)$ holds. Although $\textsc{IsSat}(t \wedge T \wedge O'_{n-1})$ is unsatisfiable, $\textsc{IsSat}(t^{(0)} \wedge T^{(0)} \wedge T^{(1)} \wedge O^{(2)}_{n-1})$ may be satisfiable, which means that t can reach O_{n-1} in 2 steps. In this case, the recursive procedure can proceed further.

Motivated by the example above, K-CAR computes k-step successors for the given state. The key difference is the action taken when an unsatisfiable result is returned from the SAT solver. Instead of directly selecting another state from the U-sequence and then going through the same procedure again, we use BMC to check if the state can reach the frame in k ($k > 1$) steps. We start with $unroll = 2$ and then increase it until a satisfiable result is returned or the given maximal unrolling level is reached, as demonstrated in Fig. 3. If the state $s \in \varphi_n$ cannot reach some state of φ_{n-1} in one step, i.e., $\textsc{IsSat}(s \wedge T \wedge O'_n)$ is unsatisfiable, we invoke BMC to check if s can reach φ_{n-1} in 2 steps (i.e., we solve $\textsc{IsSat}(s^{(0)} \wedge T^{(0)} \wedge T^{(1)} \wedge O^{(2)}_n)$). If the result is satisfiable, we obtain a new state in φ_{n-1} and recursively check if φ_{n-2} can be reached; Otherwise, the unrolling procedure continues until a satisfiable result is returned or the given maximal unrolling level is reached. The K-CAR procedure is described in Algorithm 4.

Algorithm 4: K-CAR: Computing the k-step Successor inside CAR

```
Input: A transition system Sys = (V, I, T) and a safety
       property P
Output: if terminates, "Unsafe" + a counterexample
1  if IsSat(I ∧ ¬P) then return "Unsafe"
2  U_0 := I, O_0 := ¬P
3  while true do
4      state := PickState(U)
5      O_tmp := ¬P
6      stack := ∅
7      stack.push(state, |O| − 1, 1)
8      while |stack| ≠ 0 do
9          (s, l, u) := stack.top()
10         if l < 0 then return "Unsafe"
11         if SatAssume(s^(0), ⋀_{0≤m≤u−1} T^(m) ∧ O_l^(u)) then
               M := GetModel()
12             for i = 1..u do
13                 t^(i) := M|_{V^(i)}
14                 U_{j+i} := U_{j+i} ∪ t
15                 stack.push(t, l + u − i − 1, 1)
16         else
17             stack.pop()
18             uc := GetUnsatAssumptions()
19             if l + u < |O| then O_{l+u} := O_{l+u} ∧ (¬uc) else
                   O_tmp := O_tmp ∧ (¬uc)
20             if u < MaxUnroll then stack.push(s, l, u + 1)
21             else if l + 1 < |O| then stack.push(s, l + 1, u)
22     Add a new state-set to O and initialize it to O_tmp
```

K-CAR's code is different than CAR in several places. First, like BAC, the stack stores triples with the form (s, l, u), where s is a state, l is a frame level of the O-sequence and u is the corresponding BMC unrolling level – see initialization in Line 6.

Second, as can be seen in Line 11, we check whether s can reach frame O_l in u steps, rather than a single step in CAR. If the result is satisfiable, we extract the u-long path from the model (Line 13), and add the corresponding states to the U-sequence (Line 14). By pushing, in line 15, a new triple $(t, l + u - i + 1, 1)$ to the stack we essentially check if $t^{(i)}$ can reach the corresponding frame of the O-sequence in one step.

Finally, if the SAT query is unsatisfiable and $u \leq MaxUnroll$, we invoke BMC by pushing a new triple $(s, l, u + 1)$ to the stack (Line 20), in order to find out whether s can reach O_l in $u + 1$ steps.

3.4 About Completeness

The induction condition of CAR (see Def. 1), namely that for $i \in 0..|O|$, $O_i \supseteq R^{-1}(O_{i-1})$, does not hold with K-CAR, because of the

UC that is removed from multiple O frames in line 19. For example, if O_3 cannot be reached from state s in 1 and 2 steps, then s (or a generalization thereof provided by the *UC*) is removed from both O_4 and O_5. Suppose that a state $t \in O_4$ is also a successor of s, i.e., $t \in R(s)$. Then by removing s from O_5, it is no longer the case that $O_5 \supseteq R^{-1}(O_4)$. This property is used in the completeness proof — see Prop. 1 in Sec. 2.3. A similar problem occurs with BICAR and BAC. We leave the challenge of making those algorithms complete for future work.

4 EXPERIMENTS

We implemented the three strategies described in the previous section in SimpleCAR [24], a simple but efficient implementation of the CAR algorithms. For this purpose we also implemented BMC inside SimpleCAR. We take BMC and CAR as the baseline for our experimental evaluation. We concentrate mainly on two aspects: whether those strategies can outperform BMC and CAR on finding bugs, and whether those three strategies can uniquely solve cases that cannot be solved by CAR and BMC. Our evaluation was based on 438 benchmarks in the Aiger [11] format from the single safety property track of the 2015 and 2017 Hardware-Model-Checking Competition[2]. All the counterexamples that our implementation found were successfully verified with the third-party tool *aigsim* that comes with the Aiger package[3].

We ran the experiments on a cluster running RedHat 4.8.5 with 240 nodes; each node is equipped with a 2.5Ghz Intel Xeon CPU with 96GB of RAM. The memory was limited to 8 GB and the time was limited to 1 hour.

In the rest of this section, we first give an overall experimental evaluation of the three strategies with selected parameters, and then an evaluation of each of those strategies in more depth, with a larger set of parameter values, to examine their effect. The artifacts (including all implementations and experimental results) are available at [1].

Since the three strategies depend on various parameters (e.g., the maximum BMC running time *Max_time* in BICAR, the maximum number of states *Max_n* in BAC and the maximum unrolling level *Max_unroll* in K-CAR), we experimented with several values to tune them. The results below represent the best selected values. In Fig. 4 the numerical elements in the strategies' names (e.g., 'BAC-1500') are the corresponding values of their parameters. As can be seen in the figure, BAC with *Max_n* = 1500 can solve 161 cases, compared to 145 and 148 cases solved by BMC and CAR, respectively. Moreover, it can uniquely solve 7 cases that cannot be solved by BMC and CAR.

BICAR with *Max_time* = 30 minutes (i.e., the time dedicated to the first phase of this algorithm) can solve 156 cases, 3 of which cannot be solved by BMC and CAR. As for K-CAR, when we set *Max_unroll* to 5, although the solved cases numbers are almost the same as that of CAR, it can uniquely solve 5 cases that BMC and CAR cannot. Fig. 4 shows also a *virtual best solver* (VBS), based on all columns in the figure. It can solve 172 cases. The 'overall' column adds up the results of BICAR, BAC and K-CAR in the figure. The

[2]There were 311 additional benchmarks in that set that we removed, since they were known to be safe and here we only focus on bug-finding.
[3]http://fmv.jku.at/aiger/

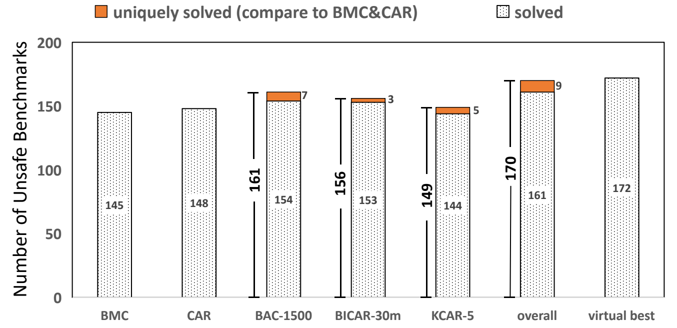

Figure 4: Number of benchmarks solved by different strategies. Uniquely solved benchmarks are those that cannot be solved by both BMC and CAR.

overall group solves 170 cases and uniquely solve 9 cases, and the performance is quite close to VBS, which shows that our strategies are competitive, and valuable in a parallel portfolio setting.

Detailed experimental results are shown in Tab. 1 and Fig. 5. Although BAC-1500 solves less cases than ABC-BMC in the preceding phase of the one hour experiment, BAC-1500 surpasses ABC-BMC in the end. And the average time for BAC-1500 and ABC-BMC are quite close. Tab. 2 shows a detailed comparison of the three proposed algorithms. BAC-1500 can independently solve 7 cases that cannot be solved by the rest two algorithms, with a smallest average time.

For reference, we also tried the best-known BMC implementation, namely the one in ABC [13] running with the 'bmc2' command (there are three different BMC implementations inside ABC, and 'bmc2' has the best performance based on [24]). It turns out that it can solve 159 cases, so we can only say that BAC outperforms it, slightly. However, all three algorithms solve some instances that it does not. In the future we intend to integrate ABC-BMC inside SimpleCAR. More details about the results appear below.

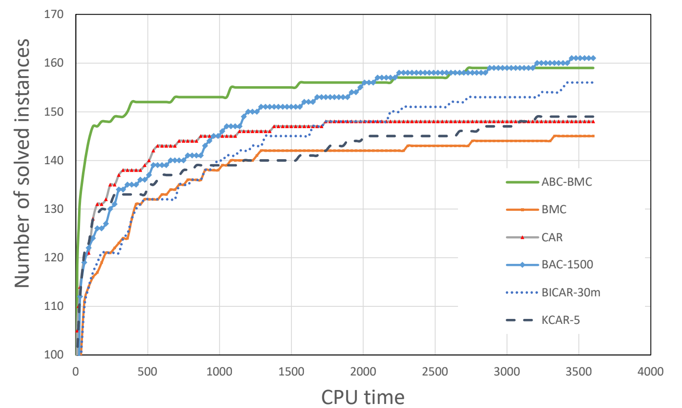

Figure 5: Cactus plots comparing different approaches.

BICAR:. As we have discussed in Sec. 3.1, BICAR performs BMC for a predefined time *Max_time* in the first phase, and then performs CAR in the second phase, starting from an O-sequence that is initialized by data gathered during the first phase (specifically, by

approach	# solved	time (sec)	average time (sec)
BMC	145	2133120.83	4870.14
ABC-BMC	159	2023570.81	4620.02
CAR	148	2100015.21	4794.56
BAC-1500	161	2035522.22	4647.31
BICAR-30m	156	2076062.05	4739.87
KCAR-5	149	2108864.44	4814.76

Table 1: Detailed experimental results of different approaches. Timeout instances are given twice the time.

approach	# solved	# independently solved (compared to the other two approaches)	average time (sec)
BAC-1500	161	7	4647.31
BICAR-30m	156	4	4739.87
KCAR-5	149	2	4814.76

Table 2: Detailed comparison of the three proposed algorithms, independently solved instances are compared to the other two approaches. Timeout instances are given twice the time.

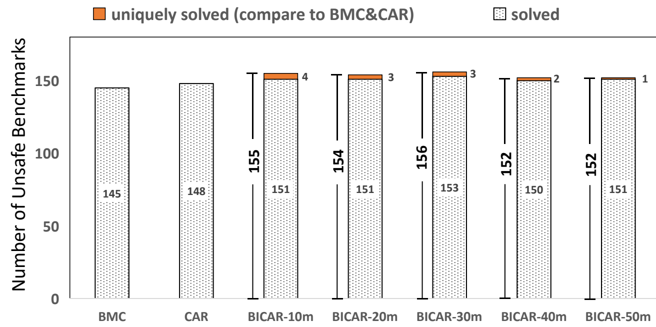

Figure 6: Number of benchmarks solved by BICAR**. Uniquely solved benchmarks are those that cannot be solved by** BMC **and** CAR**.**

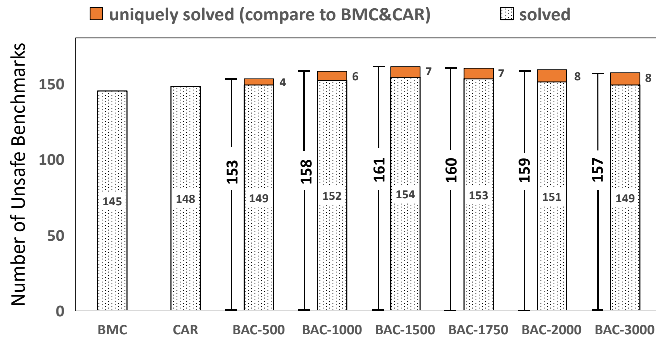

Figure 7: Number of benchmarks solved by BAC**. Uniquely solved benchmarks are those that cannot be solved by** BMC **and** CAR**.**

the negation of the unsatisfiable cores that were computed during the first phase). In Fig. 6, we see the impact of different values of *Max_time*.

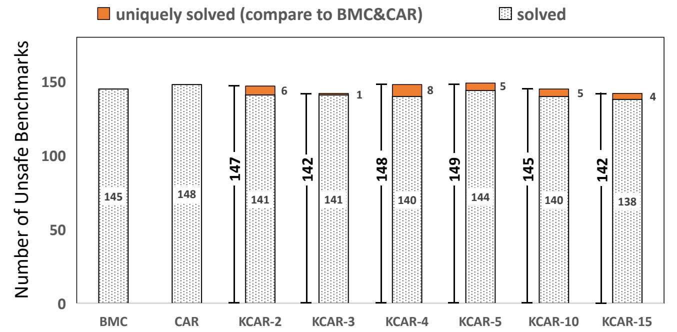

Figure 8: Number of benchmarks solved by K-CAR**. Uniquely solved benchmarks are those that cannot be solved by** BMC **and** CAR**.**

BAC:. Recall that in BAC, *Max_n* is the maximum number of states that are allowed before invoking BMC unrolling to attempt to escape from the current search location. Fig. 7 shows BAC's performance with *Max_n* assigned selected values in the range 500 to 3000.

As is shown in the figure, BAC can solve 161 cases and uniquely solve 7 cases when *Max_n* is set to 1500. Therefore BAC can surpass both BMC and CAR. Moreover BMC cannot solve 23 cases among the 161 cases solved by BAC, which has also proved BAC's value from the perspective of diversity.

K-CAR:. In K-CAR, we can adjust the value of *Max_unroll* to control the unrolling depth ($Max_unroll = 1$ is CAR itself). Fig. 8 shows the impact of this value.

More detailed results and graphs are available online in [2].

5 CONCLUSION

We presented three methods for integrating CAR and BMC, all of which achieve better results on average than either of those techniques on its own, and can solve instances that cannot be solved by either of them. There are several directions for future work. First, as mentioned above, we need to integrate SimpleCAR with a state-of-the-art BMC solver. Second, we need to find a way to replace the various constant parameters (e.g., *Max_n* in BAC, *Max_unroll* in K-CAR) with a heuristic that adapts their value at run time. For example, rather than assuming a constant value for *Max_unroll* in K-CAR (the unrolling depth), it is better to adapt its value at run time according to the actual difficulty of solving the resulting BMC instance, which varies significantly between different input models. Finally, it would be interesting to investigate if such integration techniques are also relevant to IC3/PDR, and if yes then check their effect on performance.

Acknowledgment. We thank anonymous reviewers for their helpful comments. This work is supported by National Key Research and Development Program (Grant #2020AAA0107800), and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software. Jianwen Li is supported by Shanghai Pujiang Talent Plan (Grant #19511103602) and National Natural Science Foundation of China (Grant #62002118 and #U21B2015).

REFERENCES

[1] Artifacts. https://drive.google.com/file/d/1sMD2qL9nmn6ktkNPERerA-eolLR-2Xdt/view?usp=sharing.
[2] Detailed graphs. https://drive.google.com/file/d/1XxIAhtwKdvqxvaJgk0Jxfn6fPo8G9E4L/view?usp=sharing.
[3] Minisat 2.2.0. https://github.com/niklasso/minisat.
[4] Safety model checking with complementary approximations. https://arxiv.org/pdf/1611.04946.pdf.
[5] C. Baier and J-P. Katoen. *Principles of model checking*. MIT Press, 2008.
[6] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.
[7] Dirk Beyer. Software verification. .https://sv-comp.sosy-lab.org/2021/index.php.
[8] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36st Design Automation Conf.*, pages 317–320. IEEE Computer Society, 1999.
[9] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.
[10] A. Biere and K Claessen. Hardware model checking competition. .http://fmv.jku.at/hwmcc15/.
[11] Armin Biere. AIGER Format. http://fmv.jku.at/aiger/FORMAT.
[12] A. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
[13] R. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification, CAV*, pages 24–40. Springer Berlin Heidelberg, 2010.
[14] R.E. Bryant. Graph-based algorithms for Boolean-function manipulation. *IEEE Transactions on Computing*, C-35(8):677–691, 1986.
[15] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, pages 428–439, 1990.
[16] R. Dureja, J. Li, G. Pu, M. Y. Vardi, and K. 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 - 11th International Conference, VSTTE 2019*, volume 12031 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2019.
[17] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134, 2011.
[18] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
[19] A. Griggio and M. Roveri. Comparing different variants of the IC3 algorithm for hardware model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):1026–1039, June 2016.
[20] A. Gurfinkel and A. Ivrii. Pushing to the top. In *Formal Methods in Computer-Aided Design.*, pages 65–72, 2015.
[21] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
[22] O. Kupferman and M.Y. Vardi. Model checking of safety properties. In *Proc. 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1999.
[23] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
[24] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y. Vardi. SimpleCAR: An Efficient Bug-Finding Tool Based on Approximate Reachability. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 37–44, Cham, 2018. Springer International Publishing.
[25] Jianwen Li, Shufang Zhu, Yueling Zhang, Gegang Pu, and Moshe Y. Vardi. Safety Model Checking with Complementary Approximations. In *ICCAD*, 2017.
[26] K. McMillan. Interpolation and SAT-based model checking. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.
[27] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
[28] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.
[29] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 367–385, Cham, 2019. Springer International Publishing.
[30] Y. Vizel and A. Gurfinkel. Interpolating property directed reachability. *Computer Aided Verification: 26th International Conference, CAV 2014*, pages 260–276, 2014.