Intersection and Rotation of Assumption Literals Boosts Bug-Finding

Rohit Dureja[1], Jianwen Li[1], Geguang Pu[2], Moshe Y. Vardi[3], and Kristin Y. Rozier[1]

[1] Iowa State University, Ames, IA, USA
[2] East China Normal University, Shanghai, China
[3] Rice University, Houston, TX, USA

Abstract. SAT-based techniques comprise the state-of-the-art in functional verification of safety-critical hardware and software, including IC3/PDR-based model checking and Bounded Model Checking (BMC). BMC is the incontrovertible best method for unsafety checking, aka *bug-finding*. Complementary Approximate Reachability (CAR) and IC3/PDR complement BMC for bug-finding by detecting different sets of bugs. To boost the efficiency of formal verification, we introduce heuristics involving intersection and rotation of the assumption literals used in the SAT encodings of these techniques. The heuristics generate smaller unsat cores and diverse satisfying assignments that help in faster convergence of these techniques, and have negligible runtime overhead. We detail these heuristics, incorporate them in CAR, and perform an extensive experimental evaluation of their performance, showing a 25% boost in bug-finding efficiency of CAR. We contribute a detailed analysis of the effectiveness of these heuristics: their influence on SAT-based bug-finding enables detection of different bugs from BMC-based checking. We find the new heuristics are applicable to IC3/PDR-based algorithms as well, and contribute a modified clause generalization procedure.

1 Introduction

Model checking techniques are widely used in proving functional correctness and have received unprecedented attention in the hardware and software design communities [6,18,22]. Given a system model M, and a property P representing a requirement, model checking proves whether or not P holds for M. A model checking algorithm exhaustively evaluates all possible behaviors (state-space exploration) of M, and returns a counterexample as evidence if any behavior violates the requirement. The counterexample gives the step-wise execution of the system that leads to property failure, i.e., a *bug*. Particularly, if P is a safety property, model checking reduces to reachability analysis, and the counterexample is of finite length. Popular reachability analysis algorithms include Bounded Model Checking (BMC) [9,8], Interpolation Model Checking (IMC) [25], Property Directed Reachability (IC3/PDR) [12,16], and Complementary Approximate Reachability (CAR). The common theme between these algorithms is that they are all SAT-based. BMC outperforms IMC on checking unsafe instances, i.e., *bug-finding*, while IC3/PDR and CAR can solve instances that BMC cannot [23]. It has been shown that better synergy between some of these algorithms and the SAT solver improves performance [14]. The continuous rapid advancement of SAT techniques also boosts the scalability of these algorithms.

Most SAT-based model checking algorithms use a CNF-based SAT solver as a *black-box*. The queries are expressed in CNF and the satisfiability result: SAT assignment, or unsat core, is used with or without modifications. Several solver management strategies: restart, clean-up, and (de)allocation, impact performance. In an ideal scenario, if the solver is aware of the verification problem then it may generate assignments or cores that help state-space exploration converge faster. However, achieving this is not trivial due to variability across different verification problems. There is a significant need to "guide" SAT search for model checking without modifying SAT solver internals, e.g. generating favorable unsat cores. This requires careful consideration of solver internals, and should have negligible overhead.

Complementary Approximate Reachability (CAR) [24,23] is a SAT-based model checking framework for reachability analysis. It can run in both forward and backward reachability modes; we focus on Backward-CAR as per previous work [23]. Contrary to reachability analysis via IC3/PDR, Backward-CAR maintains two sequences of over- and under- approximate reachable state-sets. The over-approximate sequence is used for safety checking, and the under-approximate for unsafety checking. We present clever and efficient heuristics to improve the performance of Backward-CAR on unsafety checking. The heuristics are inspired by assumption handling in modern CNF-based SAT solvers: assumptions literals are stored in a vector. The SAT solver propagates each assumption one-by-one, and therefore, the unsat core (UC) or satisfiable assignment can vary depending on the order in which literals are stored in the vector. Our heuristics, intersection and rotation aim to generate smaller UC and diverse states during search, respectively. In addition to the heuristics, we explore the effect of different state enumeration strategies in the under-approximate sequence of Backward-CAR. We argue that our heuristics are widely applicable and may improve the performance of other SAT-based model checking algorithms, like IC3/PDR. A thorough experimental evaluation on 748 single safety property benchmarks from HWMCC 2015 [2] and 2017 [3] reveals a 25% boost in the number of benchmarks that can be solved by Backward-CAR (155) compared to an earlier version in [23] (124). We also compare six implementations of Backward-CAR with varying heuristic combinations against reachability analysis algorithms ($5\times$BMC, $9\times$IC3/PDR) in state-of-the-art model checking tools (ABC, nuXmv, IIMC, IC3Ref).

Contributions. The contributions of our work are four-fold.

1. We propose heuristics that leverage assumption handling in SAT solvers for faster convergence and scalability of Backward-CAR (Section 3).
2. An extensive experimental analysis on real-world benchmarks supports our performance claims, and also gives a broad comparative overview of state-of-the-art algorithms for unsafety checking (Section 4).
3. We make all our tools, experiment data, and analysis publicly available.
4. A modified clause generalization procedure in IC3/PDR based on our heuristics, that may help improve scalability (Section 5).

2 Preliminaries

A *Boolean transition system* Sys is a tuple (V, I, T), where V is a set of Boolean variables, and every state s of the system is in 2^V, the set of truth assignments to variables in V. Let V' be the set of primed variables, then T is a Boolean formula over $V \cup V'$, denoting the transition relation of the system. We say that state s_2 is a *successor* of state s_1, denoted $(s_1, s_2) \in T$, iff $s_1 \cup s_2' \models T$. The variables and their negations are called *literals*. A conjunction of literals is called a *cube*. The negation of a cube is a *clause*. A cube and clause are sets of literals we conjunct and disjunct, respectively.

A *path* (of length k) in Sys is a finite state sequence $s_1, s_2, \ldots, s_k$, where each $(s_i, s_{i+1})(1 \leq i \leq k-1)$ is in T. A state t in Sys is reachable if there exists a path such that $s_k = t$. Given a Boolean transition system $Sys = (V, I, T)$ and a safety property P, which is a Boolean formula over V, the system is called *safe* if P holds in all reachable states of Sys, and otherwise it is called *unsafe*.

Let $X \subseteq 2^V$ be a set of states in Sys. We define $R(X) = \{s' \mid (s, s') \in T \textit{ where } s \in X\}$, i.e., $R(X)$ is the set of successors of states in X. Conversely, we define $R^{-1}(X) = \{s \mid (s, s') \in T \textit{ where } s' \in X\}$, i.e., $R^{-1}(X)$ is the set of predecessors of states in X. Recursively, we define $R^0(X) = X$ and $R^i(X) = R(R^{i-1}(X))$ for $i > 0$. The notations of $R^{-i}(X)$ is defined analogously.

Let v be a vector of literals indexed from 0. We use $v[i]$ to represent the i-th element of v, $v.size$ for the size of v and $v.index(l)$ for the index of a literal $l \in v$. The intersection of two vectors $v_1 \cap v_2$ is a new vector v such that (1) $l \in v \Leftrightarrow (l \in v_1 \wedge l \in v_2)$, and (2) $v.index(l_1) < v.index(l_2) \Leftrightarrow v_1.index(l_1) < v_1.index(l_2)$. We say v_2 is a *subvector* of v_1, if $size(v_2) \leq size(v_1)$ and $\exists n. v_1[n+i] = v_2[i]$ for $0 \leq i < v_2.size$. In particular, we say v_2 is the *head* (resp. *tail*) of v_1 if v_2 is a subvector of v_1 and $n = 0$ (resp. $n = v_1.size - v_2.size$). Lastly, we say that a set of states S is *diverse* if $\bigcap_{t \in S} t = \emptyset$.

2.1 SAT with Assumptions

In our formulation, we consider SAT queries of the form $\mathsf{SAT}(A, B)$, where B is a CNF formula, and A is a cube. A query with no assumptions is simply written as $SAT(\emptyset, B)$. Essentially, the query $\mathsf{SAT}(A, B)$ is equivalent to $\mathsf{SAT}(A \wedge B)$ but the implementation is typically more efficient. If $A \wedge B$ is

1. SAT, get_assignment() returns a satisfying assignment to literals in A and B.
2. UNSAT, get_unsat_core() returns a unsatisfiable core C of the literals in A, such that $C \subseteq A$, and $C \wedge B$ is UNSAT.

We abstract the implementation details of the underlying SAT solver, and assume interaction using the above functions.

2.2 Complementary Approximate Reachability

The CAR framework performs reachability analysis in both forward and backward directions. It maintains over- and under- approximate state sequences to perform safety and unsafety checking. CAR can be implemented in both forward (Forward-CAR) or

Table 1: Frame Sequences in Backward-CAR

	F-sequence (under)	**B-sequence (over)**
Init	$F_0 = I$	$B_0 = \neg P$
Constraint	$F_{i+1} \subseteq R(F_i)$	$B_{i+1} \supseteq R^{-1}(B_i)$
Safety Check	-	$\exists i \cdot B_{i+1} \subseteq \bigcup_{0 \leq j \leq i} B_j$
Unsafety Check	$\exists i \cdot F_i \cap \neg P \neq \emptyset$	-

backward (Backward-CAR) modes. We focus on Backward-CAR, hereby referred to as just CAR (refer [24] for details on Forward-CAR and correctness proofs). Given $Sys = (V, I, T)$, and a safety property P, the over-approximate state frame sequence, **B**-sequence, stores states that can reach the bad states $\neg P$, while the under-approximate frame sequence, **F**-sequence, stores states reachable from the initial state I. Frame B_i is the set of states that can reach the bad states, whereas, F_i is the set of states reachable from the initial states, in i time-steps. The states in B_i and F_i are represented as a conjunction of clauses (CNF) and disjunction of cubes (DNF), respectively. Table 1 summarizes the constraints and safety checking conditions of the two sequences.

3 Algorithm and Proposed Heuristics

The CAR algorithm incrementally builds the **B**-sequence and **F**-sequence by repeated calls to the SAT solver. The system is considered: *unsafe* when a state in the **F**-sequence intersects with the bad states, and *safe* when all states that can reach the bad states have been added to the **B**-sequence. We first describe the CAR algorithm and the motivation for our heuristics, followed by the description of the proposed heuristics.

3.1 Algorithm Description

The main CAR procedure is shown in lines 1–7 of Algorithm 1. It takes as input $Sys = (V, I, T)$ and a safety property P. The procedure first checks for any 0-length counterexample (line 1). The frame sequences are then initialized per Table.1. The main loop of CAR (lines 3–7) iteratively checks both unsafety and safety. For unsafety checking, CAR picks a state s from the **F**-sequence and checks if it can reach the bad states in UNSAFECHECK (lines 4–5). The PICKSTATE function enumerates states in the **F**-sequence (line 8). Subsequently, SAFECHECK evaluates if all states that can reach the bad states have been added to the **B**-sequence (line 6).

The UNSAFECHECK procedure of lines 8–17 takes as input a state s in the **F**-sequence, and the current frame i in the **B**-sequence, and the maximum depth k of the **B**-sequence. Let's assume $\hat{s} = s$ (line 9). The procedure checks if state s can reach states in B_i using the query $SAT(s, T \wedge B'_i)$ (line 10). If SAT, the assignment is a state t such that $(s, t) \in T$. If $i = 0$, then state t is a bad state (intersects B_0) and we have found a counterexample. Otherwise, t is added to the **F**-sequence, and the procedure recursively checks if t can reach a state in B_{i-1} (lines 13–14). If UNSAT, the negation of the unsat core $c \subseteq s$ is added to B_{i+1} (lines 15–16). Note that $\neg c$ represents the over-approximation of states that cannot reach the bad states and are blocked at B_{i+1}.

Alg. 1 Complementary Approximate Reachability (Backward)

```
1: if $SAT(I, \neg P)$ is satisfiable then return unsafe;
2: $F_0 := I$, $B_0 := \neg P$, $k := 0$;
3: while true do
4:     while ( Cube $s$ = PICKSTATE $(F)$) $\neq \emptyset$ do          ▷ state enumeration
5:         if UNSAFECHECK$(s, k, k)$ then return unsafe;
6:     if SAFECHECK $(k)$ then return safe;
7:     $k := k + 1$ and $B_k := \neg P$;                                ▷ extend B-sequence
8: procedure UNSAFECHECK$(s, i, k)$
9:     Cube $\hat{s}$ := REORDER$(s)$;               ▷ run proposed heuristics, by default $\hat{s} := s$
10:    while $SAT(\hat{s}, T \wedge B'_i)$ do
11:        if $i = 0$ then return true;                                 ▷ reached bad state
12:        Cube $t$ = get_assignment();                                 ▷ get SAT assignment
13:        $F_{j+1} := F_{j+1} \cup t$ supposing $s$ is in $F_j$ $(j \geq 0)$;   ▷ extend and add to F-sequence
14:        if UNSAFECHECK $(t, i-1, k)$ then return true;
15:    Cube $c$ :=get_unsat_core()                       ▷ $c \subseteq s$ is the UNSAT assumptions
16:    $B_{i+1} := B_{i+1} \cap \neg c$;                                 ▷ add to B-sequence
17:    return false;
18: procedure SAFECHECK$(k)$
19:    $i = 0$;
20:    while $i < k$ do                                        ▷ no new states can be added
21:        if not $SAT(\emptyset, \neg(B_{i+1} \Rightarrow (\bigvee_{0 \leq j \leq i} B_j)))$ then return true;
22:    return false;
```

The SAFECHECK procedure of lines 18–22 takes as input the maximum depth k of the **B**-sequence. By enumerating $0 \leq i \leq k$, the safety check in Table 1 for the **B**-sequence is reduced to SAT checking of a Boolean formula (line 21). If UNSAT, all states that reach the bad states have been added to B_{i+1}, and the design is safe. Otherwise, the procedure extends the **B**-sequence (line 7) and continues by picking a new state s from the **F**-sequence.

The successive blocking of states in B_{i+1} leads to faster convergence of CAR; fewer spurious states that don't reach a bad state. We want to find a minimal unsat core (MUC) c such that $\neg c$ blocks the maximum number of states in B_{i+1}, i.e., tighten the over approximation of states that lead to a bad state. However, computing MUC is expensive [24]. A straightforward solution is to drop the literals in c one-by-one and check whether the UNSAT result is preserved. However, this solution is inefficient [23]; most attempts to find a smaller UC are unsuccessful and add to the overall runtime. Our heuristics: intersection and rotation can find smaller UC with negligible runtime overhead. Our heuristics take advantage of how modern SAT solvers handle assumption literals. Minisat [17], for example, stores the assumption literals as a vector and applies *Unit Propagation* [15] starting from the first literal in the vector. Therefore, literals that appear to the front of the vector have a higher chance of being included in the unsat core, provided the SAT query is UNSAT, compared to literals towards the end. Consider the Boolean formula of line 10 and let $c_0 \subseteq s$ be the unsat core. Let $c_1 \subset c_0$ be a smaller cube. If we order the assumption literals in line 10 such that c_1 is the head of the new vector, there is a higher chance that the UC will contain literals from c_1. This assumption literal ordering technique is the primary motivation of our heuristics.

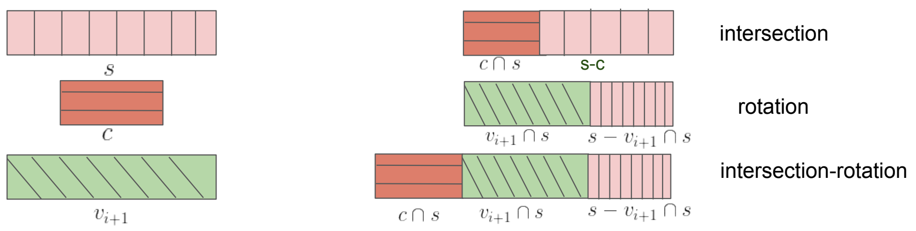


Fig. 1: Literal reordering in heuristics. We assume the SAT is $SAT(\hat{s}, T \wedge B'_i)$, where $\hat{s}$ is generated by the heuristics by reordering literals in enumerated state s, $\neg c$ is the last added clause in B_{i+1} (intersection), and v_{i+1} is the vector associated with B_{i+1} (rotation).

3.2 Intersection and Rotation Heuristics

Let $\neg c_0$ be the last clause added to B_{i+1}, i.e., c_0 is a cube and $c_0 \wedge T \wedge B'_i$ is UNSAT. If c_0 is minimal we cannot reduce it. Otherwise, we can make c_0 smaller by dropping existing literals to get c_1. The clause $\neg c_1$ is weaker than clause $\neg c_0$ and therefore blocks more states at B_{i+1}. The heuristics carefully reorder the assumption literals in s (line 10) to generate $\hat{s}$ as shown in Fig. 1.

Intersection Heuristic. Given a state s and the last added clause $\neg c$ in B_{i+1}, let $\hat{s}$ be a new assumption vector such that $c \cap s$ is the head, and $s - c$ is the tail of $\hat{s}$. We pick the last added clause in B_{i+1} to avoid overhead of selection among different clauses in B_{i+1}, however, other clauses can also be used. Therefore, in $SAT(\hat{s}, T \wedge B_i)$ literals from $c \cap s$ have a higher chance to be included in the unsat query if the query is UNSAT. It is important to note that $\hat{s}$ and s have the same literals but differ in their ordering in the assumption vector, thus preserving the satisfiability result.

Rotation Heuristic. Every call to UNSAFECHECK may generate a state t reachable from the input states. Ideally, we want these states to explore disjoint parts of the state space in the quest to find a path that reaches the bad states. The rotation heuristic helps in generating such diverse states. Each B_i $(i > 0)$ is associated with a vector v_i to store the assumptions literals for the most recent SAT query involving B_{i-1}. For example, v_i is equal to the enumerated state s_1 in the **F**-sequence that triggers the SAT query $SAT(s_1, T \wedge B'_{i-1})$. Subsequently, for B_{i-1} and a new enumerated state s_2, we generate $\hat{s}$ such that $v_i \cap s_2$ is the head and $s_2 - (v_i \cap s_2)$ is the tail of $\hat{v_i}$ as shown in Fig. 1. Note that $\hat{s}$ and s_2 have the same literals. Lastly, we update v_i to $\hat{s}$. The rotation heuristic generates diverse states as follows. For frame B_i in the **B**-sequence, let S be the set of generated states in the SAT queries and $C = \bigcap S$ be the set of common literals in states. Let c be the vector of literals in C. Assume x is an enumerated state in the **F**-sequence and the query $SAT(\hat{x}, T \wedge B'_{i-1})$ is UNSAT. Based on rotation literal reordering, c is the head of $\hat{x}$. The returned unsat core u is added to B_i. For a subsequent enumerated state y and frame B_i, a new state t is generated if the query $SAT(\hat{y}, T, B_i)$ is SAT. The new state satisfies $t \not\supseteq u$. Ideally, if $c \supseteq u$ is true, it is guaranteed that $t \cap c \neq c$. The state t is added to S, and therefore $\bigcap(S \cup \{t\}) \subset C$, i.e. the number of common literals is reduced. After several satisfiable SAT calls with different enumerated states and B_i, we will have $C = \emptyset$.

Combination of Intersection and Rotation Heuristics. The intersection and rotation heuristics have complementary strengths: intersection minimizes spurious state hits that do not reach bad states, while rotation generates diverse states reachable from the initial states. We use a combination to take advantage of their strength. For an enumerated states s we generate $\hat{s}$ as shown in Fig. 1. Let $\neg c$ be the last added clauses in B_{i+1} (intersection) and vector v_{i+1} is associated with B_{i+1} (rotation). Then $c \cap s$ is the head of $\hat{s}$, and $\hat{s_I}$ is the tail where $\hat{s_I}$ is generated from s by the rotation heuristic. Note that $\hat{s}$ may contain redundant literals but will preserve the satisfiability result of s.

4 Performance Evaluation

We incorporate the proposed heuristics in SimpleCAR [23]. Recall state enumeration in line 10 of Alg. 1. As the number of states stored in the **F**-sequence increase, the order of selection of state s becomes vital. We evaluate two simple strategies for state enumeration in the PICKSTATE procedure: begin selects states from the first element in F_0 to the last element in F_n, and end selects state from the last element in F_n to the first element in F_0. Therefore, we consider six different implementations of CAR with varying assumption ordering heuristics and state enumeration strategies.

4.1 Experiment Set-Up

We compare our additions to SimpleCAR with ABC 1.01 [13], IIMC[4], IC3Ref [4] and Simplic3 [19]. The evalauted checkers algorithms together with the respective running configurations are listed in Table 2. All checkers use the Minisat [5,17] solver. There are three implementations of BMC in ABC: `bmc` and `bmc2` for static and dynamic unrolling, respectively, and `bmc3` for dynamic unrolling plus the termination after exhausted state exploration. [5] The BMC in Simplic3 and IIMC has the same functionality as `bmc2` and `bmc3` in ABC, respectively.

We evaluate all tools against 748 benchmarks in the *aiger* format [10] from the SINGLE safety property track of the HWMCC in 2015 [2] and 2017 [3]. We primarily focus on unsafety checking in our analysis. We check correctness in two ways: (1) We use the `aigsim` [1] to check whether the counterexample generated for unsafe instances is a real counterexample by simulation, and (2) For inconsistent results (safe and unsafe for the same benchmark by at least two different tools) we attempt to simulate the unsafe counterexample, and if successful, report an error for the tool that returns safe. The experiments were performed on Rice University's DavinCI cluster[6], which comprises of 192 nodes running at 2.83GHz, 48GB of memory and running RedHat 6.0. We set the memory limit to 8GB with a wall-time limit of an hour for each benchmark. Each model checking run had exclusive access to a node. All artifacts for reproducibility and detailed experimental results for both safety and unsafety checking are available on the paper website at http://temporallogic.org/research/VSTTE19/.

[4] We use version 2.0 available at `https://ryanmb.bitbucket.io/truss/` [7]– similar to the version available at `https://github.com/mgudemann/iimc` with addition of Quip [21] and Backward IC3/PDR.

[5] From personal communication with Alan Mishchenko.

[6] https://oit.rice.edu/davinci

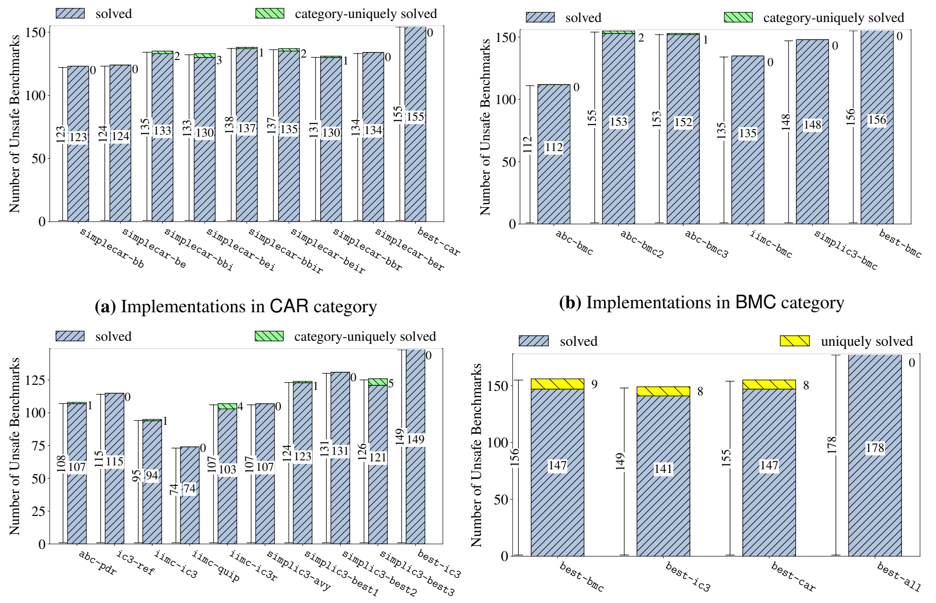
Table 2: Tools and algorithms (with category) evaluated in the experiments.

Tool	Algorithm	Configuration Flags
ABC	BMC (`abc-bmc`)	`-c 'bmc'`
	BMC (`abc-bmc2`)	`-c 'bmc2'`
	BMC (`abc-bmc3`)	`-c 'bmc3'`
	PDR (`abc-pdr`)	`-c 'pdr'`
IIMC	BMC (`iimc-bmc`)	`-t bmc --bmc_timeout 3600`
	IC3 (`iimc-ic3`)	`-t ic3`
	Quip [21] (`iimc-quip`)	`-t quip`
	Backward IC3 (`iimc-ic3r`)	`-t ic3r`
IC3Ref	IC3 (`ic3-ref`)	`-b`
Simplic3	BMC (`simplic3-bmc`)	`-a bmc`
	IC3 (`simplic3-best1`)	`-s minisat -m 1 -u 4 -I 0 -O 1 -c 1 -p 1 -d 2 -G 1 -P 1 -A 100`
	IC3 (`simplic3-best2`)	`-s minisat -m 1 -u 4 -I 1 -D 0 -g 1 -X 0 -O 1 -c 0 -p 1 -d 2 -G 1 -P 1 -A 100`
	IC3 (`simplic3-best3`)	`-s minisat -m 1 -u 4 -I 0 -O 1 -c 0 -p 1 -d 2 -G 1 -P 1 -A 100 -a aic3`
	Avy [27] (`simplic3-avy`)	`-a avy`
SimpleCAR	Backward CAR (`simpcar-bb`)	`-b -begin`
	Backward CAR (`simpcar-be`)	`-b -end`
	Backward CAR (`simpcar-bbi`)	`-b -begin -intersection`
	Backward CAR (`simpcar-bei`)	`-b -end -intersection`
	Backward CAR (`simpcar-bbr`)	`-b -begin -rotation`
	Backward CAR (`simpcar-ber`)	`-b -end -rotation`
	Backward CAR (`simpcar-bbir`)	`-b -begin -intersection -rotation`
	Backward CAR (`simpcar-beir`)	`-b -end -intersection -rotation`

4.2 Experimental Results

Performance of CAR. We compare the performance of six versions of CAR from this paper, and 2 from [23]: `simpcar-bb` and `simpcar-be`. The results are summarized in Fig. 2a. `simplecar-bbir` solves the most number of unsafe instances (138) than any other CAR implementation, while the virtual best CAR (`best-car`), which includes the six CAR implementations proposed in this paper, solves 155 instances. In contrast, the CAR implementations from [23] solves 124 instances; these instances are solved by all six CAR implementations from this paper, and on average take ∼**30% less time**. We also measure the average size of unsat cores generated by all CAR implementations at each frame in **B**-sequence. For the same benchmark, `best-car` generates on average ∼**14% smaller UC** compared to [23]. `simpcar-bbir` that uses a combination of intersection and rotation heuristics achieves the highest compression in UC size: on average ∼**20% smaller** UC. This supports our claim that smaller UC in the **B**-sequence lead to faster convergence, and validates the effectivess of our heuristics in generating smaller unsat cores and diverse states with negligible runtime overhead.

Performance of BMC. The performance of five different BMC implementations is summarized in Fig. 2b. `abc-bmc2` solves all instances that are solved by `abc-bmc`, `iimc-bmc` and `simplic3-bmc` for a total of 155. `abc-bmc3` is able to solve one



(a) Implementations in CAR category

(b) Implementations in BMC category

(c) Implementations in IC3/PDR category

(d) CAR vs. BMC vs. IC3/PDR

Fig. 2: Number of unsafe benchmarks solved. The "category-uniquely solved" benchmarks are not solved by any other implementation in the same category. The "uniquely solved" benchmarks are not solved by any other algorithm category.

instance not solved by `abc-bmc2`. The virtual best BMC (`best-bmc`, which includes all five BMC implementations, solves 156 instances.

Performance of IC3/PDR. We found an error in `simplic3-best3` on the instance "6s309b034", for which `abc-bmc3` returns unsafe (the counterexample passes the check of `aigsim`) but `simplic3-best3` returns safe. The performances of nine different IC3/PDR implementations is summarized in Fig. 2c. `simplic3-best2` solves the most number of unsafe instances (131) than any other IC3/PDR implementation, while the virtual best IC3/PDR (`best-ic3`), which includes all nine IC3/PDR implementations, solves 149 instances.

Comparison of CAR *and* BMC. The `best-bmc` and `best-car` implementations solve 156 and 155 instances, respectively. However, `best-bmc` solves 15 instances not solved by `best-car`, and `best-car` solves 14 instances not solved by `best-bmc`. The virtual best of BMC and CAR solves 170 instances.

Comparison of CAR *and* IC3/PDR. There are 20 instances solved by `best-car` that are not solved by `best-ic3`, whereas, `best-ic3` solves 14 instances not solved by `best-car`. Both Reverse-IC3/PDR (`iimc-ic3r`) and Backward-CAR perform reachability analysis in the reverse direction. `iimc-ic3r` solves four instances not solved by any other IC3/PDR implementation; all implementations of CAR solves these 4 instances. The virtual best of IC3/PDR and CAR solves 169 instances.

The three algorithm portfolios complement each other as summarized in Fig. 2d. BMC can solve 7 “6s” instances not solved by CAR and IC3/PDR, whereas, CAR can solve four “6s” and three “oski” instances not solved by IC3/PDR and BMC. Overall, the virtual best of BMC, IC3/PDR, and CAR solves 170 unsafe instances. Our heuristics have negligble runtime overhead and significantly boost the performance of CAR making it an integral part of any algorithm portfolio for unsafety checking.

5 Discussion and Future Work

Invariant checking algorithms, like IC3/PDR, maintain a frame sequence $F_0, F_1, \ldots, F_i$ to store over-approximate states reachable in up to i steps. The sequence is refined iteratively by tightening the over-approximation for every step, i.e., blocking unreachable states. IC3/PDR terminates when an inductive invariant is found. For more details on IC3/PDR, we refer the reader to [12,16,19]. Several techniques [11,20] for faster IC3/PDR convergence try to block more than one state, instead of directly using the UC from the SAT solver, by clause *generalization*.

The intersection heuristic can also benefit IC3/PDR by improving the efficiency of generalization. Alg. 2 describes a procedure to perform iterative generalization [13,19] using intersection. The literals in the clause to generalize are reorderd (line 5). The last added generalized clause $\neg c$ in F_i can be used for intersection to generate the head and tail for vector $\hat{s}$. Note that the reordering of s to $\hat{s}$ in repeated iterations of the loop (lines 2–10) mimics the same behavior as dropping literals from g, albeit, cleverly. The literal reordering may generate a smaller inductive clause $\neg b$ compared to $\neg c$ that tigthens the over-approximation F_i, hence, leading to faster convergence of IC3/PDR.

Alg. 2 IC3/PDR Generalization

```
    GENERALIZE-ITER(Clause $g, i$)
 1: done := False; Cube $s = \neg g$;
 2: for iter := 1 to max_iter do
 3:     if done then break
 4:     done := True;
 5:     Cube $\hat{s}$ = REORDER($s$);
 6:     if not $SAT(\emptyset, I \wedge \hat{s})$ and
                not $SAT(\hat{s}', F_i \wedge T \wedge \neg\hat{s})$ then
 7:         Cube $b$ := get_unsat_core()
 8:         while $SAT(\emptyset, b \wedge I)$ do
 9:             pick $l \in \hat{s} \setminus b$; set $b := b \cup \{l\}$;
10:         $s := b$; done = False; break
```

SAT solvers use the VSIDS [26] heuristic to score variables in a SAT query. The variables with high scores are preferred over variables with low scores for branching. Our heuristics implicitly perform variable scoring by picking literals from recently added clauses to the **B**-sequence, however, are external to the SAT solver. A better synergy between VSIDS in the SAT solver and our heuristics may generate even smaller UC. The state enumeration strategy also impacts performance. Quip [21] also suffers from this bottleneck: the algorithm discards states it cannot afford.[7] VSIDS can help generate diverse states by enumerating diverse satisfying assignments. CAR implementations with the intersection and rotation heuristics perform comparably, but disjointly; for example, `simpcar-beir` is unable to solve six instances solved by `simpcar-bei`. Evaluating model structure, and clausal learning as CAR progresses to better utilize the combination of the two heuristics is a promising research direction.

[7] Discussion with Alexander Ivrii

References

1. AIGER Tools. `http://fmv.jku.at/aiger/aiger-1.9.9.tar.gz`
2. HWMCC 2015. `http://fmv.jku.at/hwmcc15/`
3. HWMCC 2017. `http://fmv.jku.at/hwmcc17/`
4. IC3Ref. `https://github.com/arbrad/IC3ref`
5. Minisat 2.2.0. `https://github.com/niklasso/minisat`
6. Bernardini, A., Ecker, W., Schlichtmann, U.: Where Formal Verification Can Help in Functional Safety Analysis. In: ICCAD (2016)
7. Berryhill, R., Ivrii, A., Veira, N., Veneris, A.: Learning support sets in ic3 and quip: The good, the bad, and the ugly. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 140–147 (Oct 2017)
8. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic Model Checking Using SAT Procedures Instead of BDDs (1999)
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579. Springer (1999)
10. Biere, A.: AIGER Format. `http://fmv.jku.at/aiger/FORMAT`
11. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer Aided Design (FMCAD'07). pp. 173–180 (Nov 2007)
12. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI (2011)
13. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: CAV (2010)
14. Cabodi, G., Camurati, P.E., Mishchenko, A., Palena, M., Pasini, P.: Sat solver management strategies in ic3: an experimental approach. Formal Methods in System Design 50(1), 39–74 (Mar 2017), `https://doi.org/10.1007/s10703-017-0272-0`
15. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. Journal of Logic Programming 1(3), 267–284 (1984)
16. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: FMCAD (2011)
17. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT (2004)
18. Golnari, A., Vizel, Y., Malik, S.: Error-tolerant processors: Formal specification and verification. In: ICCAD (2015)
19. Griggio, A., Roveri, M.: Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. IEEE Trans. Comput-Aided Design Integr. Circuits Syst. 35(6), 1026–1039 (Jun 2016)
20. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in ic3. In: Formal Methods in Computer-Aided Design. pp. 157–164 (Oct 2013)
21. Ivrii, A., Gurfinkel, A.: Pushing to the Top. In: FMCAD (2015)
22. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4), 21:1–21:54 (Oct 2009), `http://doi.acm.org/10.1145/1592434.1592438`
23. 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)
24. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety Model Checking with Complementary Approximations. In: ICCAD (2017)
25. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV (2003)
26. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Proceedings of the 38th Design Automation Conference. pp. 530–535 (June 2001)
27. Vizel, Y., Gurfinkel, A.: Interpolating Property Directed Reachability. In: CAV (2014)