electronics

MDPI

Article

Finding More Property Violations in Model Checking via the Restart Policy

Mengtao Geng, Xiaoyu Zhang and Jianwen Li *

School of Software Engineering, East Normal China University, Shanghai 200062, China; 51194501040@stu.ecnu.edu.cn (M.G.); 51194501082@stu.ecnu.edu.cn (X.Z.)
* Correspondence: jwli@sei.ecnu.edu.cn

Abstract: Model checking is an efficient formal verification technique that has been applied to a wide spectrum of applications in software engineering. Popular model checking algorithms include Bounded Model Checking (BMC) and Incremental Construction of Inductive Clauses for Indubitable Correctness/Property Directed Reachability(IC3/PDR). The recently proposed Complementary Approximate Reachability (CAR) model checking algorithm has a performance close to BMC in bug-finding, while its depth-first strategy sometimes leads the algorithm to a trap, which will waste lots of computation. In this paper, we enhance the recently proposed Complementary Approximate Reachability (CAR) model checking algorithm by integrating the restart policy, which yields a restartable CAR model (abbreviated as r-CAR). The restart policy can help avoid the trap problem caused by the depth-first strategy and has played an important role in modern SAT-solving algorithms to search for a satisfactory solution. As the bug-finding in model checking is reducible to a similar search problem, the restart policy can be useful to enhance the bug-finding capability. We made an extensive experiment to evaluate the new algorithm. Our results show that out of the 749 industrial instances, r-CAR is able to find 13 instances that the state-of-the-art BMC technique cannot find and can solve more than 11 instances than the original CAR. The new algorithm successfully contributes to the current model-checking portfolio in practice.

Keywords: model checking; CAR; BMC; bug-finding; SAT

Citation: Geng, M.; Zhang, X.; Li, J. Finding More Property Violations in Model Checking via the Restart Policy. *Electronics* **2021**, *10*, 2957. https://doi.org/10.3390/electronics10232957

Academic Editor: Tiziana Margaria

Received: 28 October 2021
Accepted: 25 November 2021
Published: 27 November 2021

1. Introduction

Model checking [1] is an efficient technique for formal verification that has been applied into most stages of the life cycle in software development to ensure correctness. For example, model checking can be used to verify the software requirements [2–5], software design models [6–8] and even as testing and debugging [9,10]. Moreover, model checking can be adopted to verify a wide spectrum of applications such as the web [11–14], device drivers [15–17], GUI [18,19], distributed programs [20–22], embedded systems [23,24], databases [25], and malware [26]. These scenarios show that model checking has played an important role in software engineering [27].

Given a software design M as the model and the formal specification (property) P, which is often written by some temporal logic [28], model checking checks whether P holds for all behaviors of M. To achieve this goal, a model-checking algorithm explores the state space of M by starting from the initial states to all their reachable states in M. Moreover, model-checking techniques terminate the exploration as soon as (1) a counterexample as witness of the property violation is detected (In general, finding property violations refers to the same thing as finding bugs/counterexamples), or (2) the proof is accomplished that the initial states can never reach the states which violate the property P. If P is a safe property [29], the length of the counterexample becomes finite. As a result, the safety model checking can be reduced to the reachability analysis problem [30], and we focus on the safety model checking in this paper.

Although model checking has been widely used in software and hardware verification, the performance improvement is still eagerly on demand to help solve more industrial instances. It is well known that no model-checking technique is the best one to dominate all others, and different algorithms perform differently for different benchmarks [31]. Although invented nearly three decades ago, Bounded Model Checking (BMC) [32,33] is still considered as the most efficient technique for detecting property violations, or say, *bugs*. Meanwhile, Interpolation Model Checking (IMC) [34] and Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) [35], or Property Directed Reachability (PDR) [36], are shown to be more fit for proving correctness. Therefore, a portfolio of model checking techniques is often maintained by either academic or industrial model checkers to solve different problems.

Recently, a new model-checking algorithm named Complementary Approximate Reachability (CAR) [37], was proven to complement BMC on bug-findings, i.e., detecting property violations, and IC3/PDR on correctness proofs. That is, CAR is able to solve instances that BMC or IC3/PDR cannot solve within the given time and hardware sources. The achievement from CAR inspires us that, even though relevant techniques have been deeply investigated for decades, there are possibilities to improve the model-checking performance such that it can be more useful for the industry. In this paper, we focus on CAR and present an improved search strategy inside the algorithm to gain a better bug-finding performance.

CAR was inspired by IC3/PDR and the traditional reachability analysis [37], which maintains an over-approximate state sequence for correctness proof and an under-approximate state sequence for bug-finding. CAR utilizes the depth-first search strategy to find new states that meet the constraints, which are used to refine the under-approximate state sequence or collect the relevant information to refine the over-approximate state sequence if failed. The algorithm terminates as soon as either a *bad* state is in the under-approximate sequence, which indicates a counterexample has been detected, or an *invariant* has been computed based on the over-approximate sequence, which indicates the correctness proof has been asserted. For more details, see below. CAR can be performed in both forward and backward directions. Since evidences have shown that Backward-CAR is better than Forward-CAR at bug-finding [38], we follow the observation and focus on improving Backward-CAR. In the rest of the paper, all mentions of "CAR" represent Backward-CAR unless it is specifically clarified.

Although CAR has shown the advantage of detecting bugs for safety model checking and outperforms IC3/PDR in bug-finding, it cannot solve as many unsafe instances (those with bugs) as BMC in the current stage [39]. The depth-first strategy may lead the algorithm to a *trap* for those unsafe cases it is unable to solve. As a result, to keep searching for new states is almost impossible for the algorithm to locate the bad states and only wastes the computation sources. Such a similar phenomenon occurs on solving the satisfiability of Boolean formulas (SAT) [40], in which the search can also be in the trap if the order of variable assignments is not properly chosen. To tackle such issue, researchers propose a *restart* policy such that the current search path is discarded and a new one can be selected to get rid of the trap [41]. Their experiments show that such a simple strategy is very efficient to help speedup SAT solving, particularly for those satisfiable instances.

Inspired by the results achieved by applying the restart policy to modern SAT solvers [41], we leverage a similar idea to enhance the performance of CAR in bug-finding. The new algorithm is named r-CAR (restartable CAR). In our designation, the restart policy is invoked as soon as the size of the new elements of the over-approximate state sequence in a single search reaches the *frequency* $k \times t$, where k is the length of the over-approximate sequence and t is a given *threshold*, which can be dynamically updated based on a given *growth rate* gr during the search. That means, if the current threshold is t and the growth rate is gr, the threshold will be updated to be $(t \times gr)$ when the restart is invoked next time. Moreover, the search will be restarted the next time as soon as the size of the new elements of the over-approximate sequence reaches $(k \times t \times gr)$. As a result, the restart frequency depends on the threshold and the corresponding proportion to update it. Once the restart is triggered, CAR deletes all state

information collected in the current search and starts a new one immediately. Notably, the previous path information has been stored in the over-approximate sequence such that it is guaranteed the new search is able to find a different path with all founded before.

We conduct a comprehensive experimental evaluation on the 749 industrial instances from the Hardware Model Checking Competition in 2015 [42] and 2017 [43]. We implement our new algorithm based on the SimpleCAR model checker [38,44] and compare the bug-finding performance to the original CAR in SimpleCAR, as well as the BMC and IC3/PDR algorithms that are implemented in the state-of-the-art model checker ABC [45]. The results show that, given the same time and hardware sources, r-CAR can solve 13 new unsafe instances compared to BMC and find 11 more counterexamples compared to the original CAR by feeding different restart configurations. Moreover, r-CAR is able to outperform IC3/PDR on bug-finding (checking unsafe instances). The new algorithm helps increase the diversity to solve more instances. Therefore, we show that combining the restart policy with CAR is able to increase the power of the current model-checking portfolio in the industry.

In summary, this paper makes the following contributions:

- We propose r-CAR, an enhanced CAR-based model-checking algorithm, to detect more property violations, or, say, bugs/counterexamples.
- We implement r-CAR to produce a practical model checker called RestartCAR and conduct an extensive experimental evaluation to show that RestartCAR improves the capability of the SimpleCAR model checker to find more unsafe instances by feeding different restart configurations.
- We further identify the practical restart configurations for RestartCAR and study the effectiveness of r-CAR.

2. Related Work

Compared to Theorem Proving [46], another mainstream formal verification technique, the advantage of model checking is to avoid massive manual work and enable *automatic* verification, which is accomplished by performing the exhausted search on the graph constructed from the model together with the property. However, the main challenge in model checking is the exponential scaling of the model's state space, the so-called "state-explosion problem" [47].

Early approaches to model checking [48,49] were based on an explicit search of the model's transition graph, where nodes represent states and edges represent system transitions. Such explicit-state techniques typically do not scale well beyond models with a few million states [50]. A major breakthrough, in the early 1990s, was the introduction of symbolic techniques, which replaced explicit search with Boolean reasoning techniques. The development of Binary Decision Diagrams (BDDs) [51] led to the development of BDD-based symbolic model checking, which enabled the verification of systems with 1020 states [52]. Yet BDD-based techniques rarely scale to models with more than 1000 Boolean state variables, which limits their applicability to the verification in industry [53].

In the late 1990s, SAT solving emerged as a highly effective Boolean reasoning technique [54]. The first application of SAT solving to model checking was in the context of bounded model checking (BMC), in which the search over model behavior is subject to a depth bound [32]. This approach, where model checking is reduced to a sequence of SAT-solving calls, one for each depth bound, has been shown to be highly effective in practice, particularly for detecting property violations (bugs) [49]. Yet BMC is incomplete, as it can only reveal the presence of counterexample behavior, but not prove their absence, which led to a quest to develop SAT-based complete model-checking techniques. This is still a very much active area, as no single approach has proven to be superior to all other approaches, cf. [40]. While some approaches have tried to find ways to extend BMC to make it complete, e.g., [55], others have tried to follow the approach of BDD-based model checking.

There are two ideas in BDD-based model checking [52]: (1) a set of states can be represented by a Boolean formula (a BDD is a special case), and (2) a key operation in searching the state space is the image/pre-image operation, in which we symbolically compute the set of successor/predecessor states of a given set S of states. Much of the research in BDD-based symbolic model checking has focused on the efficient implementation of the image operation, cf. [56]. One direction of research on SAT-based complete model-checking techniques has been on a SAT-based implementation of the image operation. While standard SAT solving returns a single satisfying assignment when the formula under test is satisfiable, there is a variant, called All-SAT, that returns a representation of all satisfying assignments, cf. [57].

All-SAT-based symbolic model checking did not, however, prove to provide a highly scalable approach. Two other SAT-based approaches emerged in the following years. Interpolation-based model checking [34] combines the use of Craig Interpolation as an abstraction technique with the use of BMC as a search technique. IC3/PDR starts with an over-approximation and is gradually refined to be more and more precise [35,36]. Both approaches have proven to be highly scalable and are today parts of the algorithmic portfolio of modern model checkers, such as ABC [45]. Normally, users prefer to using BMC for bug-finding (checking unsafety) and using IC3/PDR and IMC for correctness proving (checking safety). Recently, a new model checking algorithm, CAR, was presented, and the preliminary results showed that it succeeds in complementing BMC, IC3/PDR, and IMC by solving instances that cannot be solved by those three algorithms [37,44].

Although BMC, IC3/PDR, IMC, and CAR utilize the SAT technique to achieve the search task, they follow different strategies. Explicitly, BMC and IMC use the breadth-first strategy, while IC3/PDR and CAR use the depth-first search strategy. Since IMC is developed upon BMC, its bug-finding performance is completely dominated by that of BMC, according to previous literature [44]. In addition, IC3/PDR pays more effort to generate the so-called *minimal inductive clauses* for proving correctness such that its overall performance on bug-finding is not as good as that of CAR. Therefore, BMC and CAR are the best two options for bug-finding. On one hand, BMC has been proposed for decades, and is now very difficult to improve the performance. On the other hand, CAR is a new algorithm that leaves many potential slots for improvement. As a result, this paper presents one candidate solution to improve the bug-finding performance on CAR. Compared to the original CAR [44], the new algorithm r-CAR enhances it by introducing the restart policy such that the depth-first search inside CAR can be restarted if the current path is determined as a non-promising one to find the solution. Moreover, the import of restart significantly improves CAR's performance in bug-finding while preserving the performance in proving correctness, as shown in the experimental section.

In general, among all modern model checking algorithms mentioned above, BMC can find far more counterexamples than IC3/PDR and IMC within the same time and memory limit, but BMC does not have the ability to prove correctness. The CAR algorithm has the ability to prove correctness and has a performance close to BMC in bugs-finding. IC3/PDR and IMC focus more on proving correctness and therefore has a better performance in proving correctness and a relative poor performance in bugs-findings compared to CAR and BMC. After we introduce the restart policy into the CAR algorithm, we obtain the r-CAR algorithm, which retains CAR's ability of proving correctness and enhances the ability to find counterexamples. Moreover, r-CAR can find more counterexamples by running different parameter combinations in parallel, which is not available to other algorithms. Please check Table 1.

Table 1. Modern model checking algorithms.

Name	Strategy	Bug-Finding	Prove Correctness
BMC	breadth-first	++++	
IC3/PDR	depth-first	++	+++
IMC	breadth-first	+	+++
CAR	depth-first	+++	++
r-CAR	restartable + depth-first	++++	++

All the aforementioned model checking algorithms are originally bit-level techniques that can only handle Boolean transition systems. Recently, several efforts have been made to immigrate such bit-level algorithms to the so-called *word-level* model checking, using the SMT engine instead of the SAT one due to the increasing interests in the SMT domain [58–61]. Normally speaking, the bit-level model checking techniques are used mainly in hardware verification, while the work-level model checking techniques focus on software verification.

3. Preliminaries

3.1. Boolean Transition System

Modern SAT-based model checking techniques consider the *Boolean transition system* as the model. A Boolean transition system *Sys* is defined as a tuple (V, I, T), where V is a set of Boolean variables and each state s of the system is in 2^V, the set of truth assignments to variables in V. I is the set of initial states. If we mark the copy of V as $V^{'}$ to represent the set of primed variables, T is the transition relation of the system over $V \cup V^{'}$. We say that state s_2 is a *successor* of state s_1, if $s_1 \cup s_2^{'} \models T$, denoted as $(s_1, s_2) \in T$.

A finite state sequence $s_0, s_1, ..., s_k$ is called a path of length k, if each (s_i, s_{i+1}) for $(0 \leq i \leq k-1)$ is in T. We say the state t is reachable from state p in (resp. within) k steps, if there exists a finite path with length k (resp. smaller than k) such that $s_0 = p$ and $s_k = t$ are true. All states that are reachable from the initial states I are called the *reachable states* of *Sys*. Given a safety property P (represented as a Boolean formula) and Boolean transition system $Sys = (V, I, T)$, we say the system is *safe* for P if every reachable state s of *Sys* satisfies P, i.e., $s \models P$. Otherwise, the system is *unsafe*. We call the state violating P a *bad* state and use $\neg P$ to denote the set of all bad states. A path from an initial state in I to one of the bad states in $\neg P$ is called a *counterexample*.

Let $X \subseteq 2^V$ be a set of states in *Sys*. We define the relation $R(X)$ to be the set of successors of the states in X, i.e., $R(X) = \{s' | (s, s') \in T \text{ for } s \in X\}$. We define $R^i(X) = R(R^{i-1}(X))$ for $i > 1$. Similarly, we define $R^{-1}(X)$ as the set of predecessors of states in X and $R^{-i}(X)$ analogously for $i > 1$.

We call a Boolean variable a or its negation $\neg a$ as a *literal*. Let L be a set of literals. A *cube* is a Boolean formula with the form of $\bigwedge l$ where $l \in L$. Analogously, a *clause* is a Boolean formula with the form of $\bigvee l$, where $l \in L$. It is not trivial to see that a state of *Sys* is a cube. In the rest of the paper, we will mix-use the terms *state* and *cube* for convenient description.

3.2. The High-Level Description of CAR

Derived from the traditional reachability analysis, CAR can perform in both the forward and backward directions. As Backward CAR has been shown better than Forward CAR [38], in the rest of the paper, we focus on Backward CAR and all mentions of "CAR" represent Backward CAR. The CAR algorithm maintains a finite *under-approximate* state sequence $F = F_0, F_1, ..., F_k (k \geq 0)$ starting from I (the set of initial states), i.e., $F_0 = I$, and each F_i is a subset of states reachable from I in i steps. Such an under-approximate sequence is called the *F-sequence*. In addition, CAR maintains an *over-approximate* sequence $B = B_0, B_1, ..., B_k (k \geq 0)$ starting from the bad states, i.e., $B_0 = \neg P$, and a state is included in $B_i (i \geq 0)$ if it can reach $\neg P$ in i steps. The sequence is called the *B-sequence*. In addition,

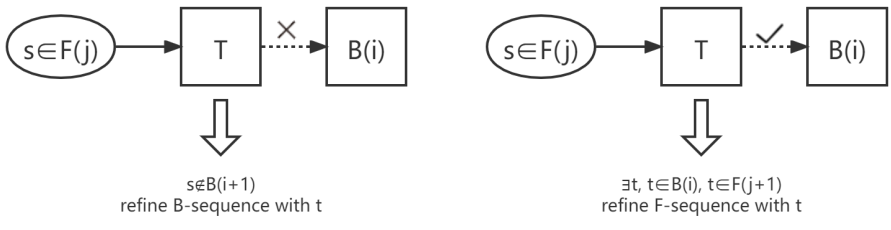
each element $B[i]$ of the B-sequence is named a *frame* and i is the *frame level*. States in F-sequence are represented as a disjunction of cubes, while the states in B-sequence are represented as a conjunction of clauses.

A summary of both F- and B-sequences including the initialization, constraints, and safety/unsafety checking conditions are listed in Table 2. The F-sequence is defined recursively that (1) $F_0 = I$, i.e., the first element of the sequence is the set of all initial states, and (2) $F_{i+1} \subseteq R(F_i)$ for $i \geq 0$, i.e., the element of the sequence at position $i+1$ is a subset of states which represent the successors of those at position i. Since each F_i represents only a part of real states at position i, the F-sequence is under-approximate. Because the F-sequence does not include all state information, we can only use it to check unsafety. That is, if a state in some F_i is also a bad state in $\neg P$, a counterexample is found and the unsafety result can be reported. Analogously, the B-sequence is defined recursively that (1) $B_0 = \neg P$, i.e., the first element of the sequence is the set of all bad states (represented by $\neg P$), and (2) $B_{i+1} \supseteq R^{-1}(B_i)$ for $i \geq 0$, i.e., the element of the sequence at position $i+1$ is a superset of states which represent the predecessors of those at position i. Since each B_i includes the information of all real states at position i, the B-sequence is over-approximate. Since the B-sequence includes more state information than the real ones, we can only use it to check safety. If every state in some B_{i+1} is included in some B_j for $0 \leq j \leq i$, the correctness is proved and the safety result is reported.

Table 2. The summary of key structures in CAR.

	F-Sequence (under)	**B-Sequence (over)**
Initial	$F_0 = I$	$B_0 = \neg P$
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	$\exists i \cdot F_i \cap \neg P \neq \emptyset$	-

Figure 1 shows the schema on how to refine elements in both sequences. The crux is a Boolean formula $\phi = s \wedge T \wedge B(i)'$, in which s is a state in the F_j, T is the transition relation formula, and B(i) is the i-th element of the B-sequence ($B(i)'$ is the prime version). Informally speaking, the formula ϕ queries whether one of the successors of state s can be in $B(i)$. The query can be sent to a SAT solver, and if a satisfying assignment is returned, the F-sequence can be updated based on the information from the assignment (see Figure 1b). Otherwise, the B-sequence can be refined according to the unsatisfiable cores from the SAT solver, which is a subset of s (Figure 1a). As the length of the B-sequence being increased, we enumerate the elements in F- and B-sequences to feed the above formula ϕ and therefore update all information of the sequences.



(**a**) (**b**)

Figure 1. The schema to refine elements in the B- and F-sequences. This figure is better viewed online. (**a**) The schema to update elements in the B-sequence. (**b**) The schema to refine elements in the F-sequence.

3.3. SAT Calls with Assumptions and Unsatisfiable Cores

As introduced above, the CAR algorithm frequently invokes the SAT calls whose inputs have the form of $A \wedge B$, where $B = T \wedge B(i)'$ is a CNF formula, a Boolean formula

with the form of $\bigwedge c_i$, where each c_i is a clause, and A ($= s$) is a cube. We use the notation $SAT(A, B)$ to represent such SAT queries and take A as the *assumptions* of the SAT solver. Although the result of $SAT(\emptyset, A \bigwedge B)$ is equivalent to that of $SAT(A, B)$, using the latter one is typically more flexible for incremental SAT solving, which is a very efficient mechanism to frequently invoke SAT solvers in practice. There are two different outcomes from an SAT solver when handling the query $SAT(A, B)$. If the result is *satisfiable*, an assignment of the formula $A \wedge B$ is provided by the SAT solver. Otherwise, $A \wedge B$ is unsatisfiable and an Unsatisfiable Core (UC) $uc \subseteq A$, which is a subset of the assumptions A, and can be returned by the SAT solver. In CAR, the assignments are used to extract the new *explicit* states of the system that are added to the under-approximate sequence, while the unsatisfiable cores are collected to refine the over-approximate sequence.

4. Algorithm Design

4.1. Algorithmic Description of CAR

The pseudo-codes for the main procedures of CAR are shown in Algorithm 1. The entry procedure takes a system $Sys = (V, I, T)$ and a safety property P as the inputs, and outputs are *safe* if an invariant is detected (Line 14), which indicates the correctness is proven, or *unsafe* if a counterexample is found (Line 7), which means a property violation exists. The texts in red are introduced to implement the restart policy, which will be explained in the next section.

The main framework of CAR is shown from Line 1 to Line 15 of Algorithm 1. The first SAT call at Line 1 is used to check whether there is a counterexample with the length of 0, which means that some initial state in I is also a bad state in $\neg P$. If the SAT query returns unsatisfiable, CAR initializes the B-sequence and F-sequence at Line 3, according to the rules in Table 2. The whole loop from Line 5 to Line 15 increases the length of the B-sequence gradually (see Line 13) and first calls the *UNSAFECHECK* procedure to search new states and returns unsafe if a counterexample is found. Notably, inside the procedure, the length of the F-sequence can be increased while that of the B-sequence cannot. Meanwhile, the F- and B-sequence can be updated during the search inside the procedure. If *UNSAFECHECK* proceeds but no counterexamples are detected, the *SAFECHECK* procedure is then used to check whether an invariant can be found based on the information of the F-sequence. The whole loop terminates as soon as one of the above two procedures returns, as discussed in [37]. A summary of procedures in CAR is listed below:

Algorithm 1 Main Procedures of r-CAR and CAR (without texts in red)

Input: $Sys = (V, I, T)$ and Safety Property P;
Output: return safe or unsafe.

1: **if** $SAT(I, \neg P)$ is satisfiable **then return** unsafe;
2: **end if**
3: $F_0 := I$, $B_0 := \neg P$, $k := 0$;
4: *restart* := *false*, *count* := 0, *threshold* := $threshold_0$;
5: **while** true **do**
6: **while** (Cube $s =$ PICKSTATE $(F)) \neq \emptyset$ **do**
7: **if** UNSAFECHECK(s, k, k) **then return** unsafe;
8: **end if**
9: **if** *restart* **then** break;
10: **end if**
11: **end while**
12: **if not** *restart*
13: $k := k + 1$;
14: **if** SAFECHECK (k) **return** safe;
15: **else** RESTART()
16: **end while**

17: **procedure** UNSAFECHECK(s, i, k)
18: **if** RESTARTPOINT$(k, threshold, count)$
19: *restart* := *true*;
20: **return** false
21: Cube $\hat{s} :=$ REORDER(s);
22: **while** $SAT(\hat{s}, T \wedge B_i^{'})$ **do**
23: **if** $i = 0$ **then return** true;
24: **end if**
25: Cube $t :=$ get_assignment();
26: $F_{j+1} := F_{j+1} \cup t$ supposing s is in F_j $(j \geq 0)$;
27: **if** UNSAFECHECK $(t, i-1, k)$ **then return** true;
28: **end if**
29: **end while**
30: Cube $c :=$get_unsat_core()
31: *count* := *count* + 1;
32: $B_{i+1} := B_{i+1} \cap \neg c$;
33: **return** false;
34: **end procedure**

35: **procedure** SAFECHECK(k)
36: $i := 0$;
37: **while** $i < k$ **do**
38: **if not** $SAT(\emptyset, \neg(B_{i+1} \Rightarrow (\bigvee_{0 \leq j \leq i} B_j)))$
39: **return** true;
40: **end while**
41: **return** false;
42: **end procedure**

- **PICKSTATE** at Line 6 takes the F-sequence as the input and uses certain decision strategies to enumerate and select a state from the sequence. For example, we may enumerate the states from the beginning (resp. end) to the end (resp. beginning) of the sequence, which can be implemented in a trivial way. The procedure returns an empty set $\emptyset$ if all states in the sequence are considered but no more available states can be chosen.
- **REORDER** at Line 21 takes a state as the input. Inspired by the concept of assumptions in modern SAT solvers, this procedure maintains two non-conflict policies named *intersection* and *rotation*, which are designed to generate smaller unsatisfiable cores so

as to boost the efficiency of the algorithm. The procedure reorders the literals in the state s to generate its new copy $\hat{s}$ (Cube $\hat{s}$ at Line 21), which is then transferred to the SAT solver as assumptions. For example, given a state $s = (l_1, l_2, l_3, l_4)$, the returned state $\hat{s}$ may be (l_3, l_4, l_1, l_2) according to the reorder policy inside the procedure. Although the SAT query result remains the same, the latter assumptions may lead to a smaller unsatisfiable core (UC) and the literature [39] has shown the efficiency of such reorder heuristics.

- **get_assignment** at Line 25 returns a satisfying assignment of the input formula if the SAT query result is satisfiable. A new state t, which is a successor of s, can be extracted from the assignment. Details are referred to in [37].
- **get_unsat_core** at Line 30 generates an unsatisfiable core uc, which is a subset of the assumptions $\hat{s}$ in the current SAT call, if the query result is unsatisfiable. It is trialed to see that uc is also a subset of s. Essentially, the unsatisfiable core uc represents a set of states (including s) that does not meet the query. Using uc instead of s to update the over-approximate sequence is proven to be more effective.
- **UNSAFECHECK** from Lines 17 to 34 takes a state s, an integer i representing the current frame level of the B-sequence, and an integer k representing the length of the B-sequence as inputs. The procedure first reorders the input state s to $\hat{s}$ through the REORDER procedure, and then invokes an SAT call $SAT(\hat{s}, T \wedge B_i{}')$ to check whether state s can directly reach states in B_i. If the result is unsatisfiable, it calls get_unsat_core() to obtain an unsatisfiable core $c \subseteq s$. Considering that $\neg c$ represents the over-approximate set of states which should not be in B_{i+1} because they cannot reach states in B_i, the unsatisfiable core c is added to B_{i+1}. On the other hand, if the SAT query is satisfiable and the given integer i is 0 (Line 23), that is, state s can reach the bad states in B_i. As the state s is selected from the F-sequence, which stores states reachable from I, a counterexample is found and the procedure returns true. If the SAT call is satisfiable with $i > 0$ (Lines 25–27), we invoke get_assignment() to obtain a new state t, which is added into the F-sequence as the successor of s, and recursively invoke UNSAFECHECK$(t, i-1, k)$.
- **SAFECHECK** at Lines 35–42, takes an integer k and the length of the B-sequence as the inputs. By enumerating i from 0 to k, the procedure checks whether all the states in B_{i+1} have been contained in the union set of $B_0, B_1, ..., B_i$. If this is the case, We can assert that all the states that can reach the bad states $B_0 = \neg P$ have been included in the B-sequence. Because the initial states I are not in the B-sequence, the system Sys is safe for the property P. This procedure exactly implements the safety check condition shown in Table 2.
- **RestartPoint** at Line 18 returns true if CAR is ready to restart the state search according to the restart policies introduced below.

4.2. Restart Policy

The *restart* mechanism has been widely implemented in modern SAT solvers to improve their performance. The motivation comes from the observation that the search inside the solver may become trapped due to an improper order of the assignments to the variables in the Boolean formula. Under such scenarios, to keep searching is almost impossible to find the final result but only wastes the computation sources. Therefore, it is reasonable to abandon the current search path and restart it again with different variable assignments. Studies have shown that such a simple strategy turns out to be very efficient to help solve more satisfiable instances [41]. It is surprising to see that CAR also suffers from a similar problem during the state search, and the idea of applying the restart policy to CAR comes out straightforward.

As shown in Algorithm 1, the texts in red are pseudo-codes added to integrate the start policy in CAR and therefore produce r-CAR. We use a counter variable *count* to record the number of unsatisfiable cores generated in the current search. The *count* increases every time a new unsatisfiable core is computed (Line 31) and will be set to 0 after each

restart (Line 8). The insight is that too many unsatisfiable cores are computed in a single search probably leads to a trap. In addition, a *threshold* that can be dynamically updated is provided, and the restart policy is triggered as soon as the condition $count > frequency$ becomes true (Line 3). Notably, we use a flag *restart* to control whether the restart policy should be triggered (Line 12), whose value is updated based on the return value of the *RestartPoint* procedure (Line 18).

Once the restart policy is triggered, CAR abandons the current search and starts over again. However, the restart frequency is a key reason that affects the final performance. If the frequency is set too high, CAR may lose the instances that can be solved when no restart is applied to the algorithm. On the other hand, if the frequency is set too low, it may not be helpful to solve more instances that cannot be solved when no restart is applied to the algorithm. In the implementation, we control the restart frequency according to a *threshold*, whose value is initialized at the beginning ($threshold_0$ at Line 4 of Algorithm 1) and then can be updated based on a *growth rate gr*. The value of $threshold_0$ determines the initial frequency of the restart policy through the equation $frequency = threshold_0 * (k + 1)$, where $(k + 1)$ represents the length of the current B-sequence. How the restart frequency dynamically updates depends on *gr*. After each time the CAR algorithm triggers the restart policy, the *threshold* is multiplied by *gr*, leading to the increment in the restart frequency.

In the *UNSAFECHECK* procedure, *RESTARTPOINT* is invoked to judge whether it is ready to restart. The procedure takes an integer *k* representing the length of B-sequence and an integer *count*, which counts the number of new unsatisfiable cores (Line 31) generated in the current search, as the inputs. Since the B-sequence is over-approximate, generating new unsatisfiable cores exactly makes the B-sequence more precise, which may prevent the algorithm from searching the same path. Therefore, we take the length of the B-sequence into consideration, and the restart frequency is the product of the length of B-sequence and *threshold*. As soon as *count* is larger than *frequency*, *RESTARTPOINT* returns true and the *restart* flag becomes true, which makes the procedure *UNSAFECHECK* terminate with the output *false* (Lines 18–20). Once the restart point is reached, all recursive calls in *UNSAFECHECK* are returned as false, leading to the termination of the loop at Lines 6–9 and the entry to the procedure RESTART at Line 15.

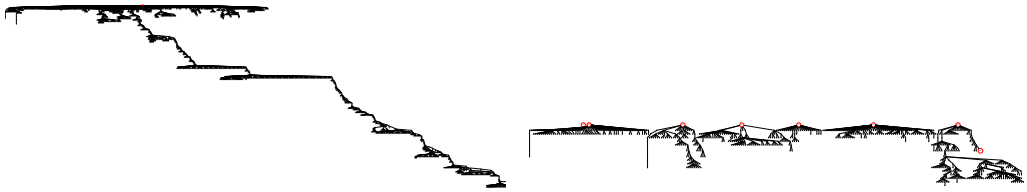
The *RESTART* procedure at Line 7 of Algorithm 2 resets the unsatisfiable core counter *count* (Line 8) and enlarges the *threshold* with growth rate *gr* (Line 9). Compared to the previous search from initial states I, we have updated B_i with a certain number of unsatisfiable cores, which probably generates a different search path from the previous ones. The procedure *BACKTRACK* contains the process of returning to initial states I, eliminating the F-sequence to release the memory, and some auxiliary work such as the reconstruction of the SAT solver.

Algorithm 2 The restart policy

```
1:  procedure RestartPoint(k, threshold, count)
2:      Let frequency := (k + 1) * threshold;
3:      if count > frequency then return true;
4:      end if
5:      else return false;
6:  end procedure
7:  procedure Restart
8:      count := 0;
9:      threshold := threshold * gr;
10:     BACKTRACK();
11: end procedure
```

To clearly show the scenario before and after introducing the restart policy, we show the difference between the searching diagrams of CAR and r-CAR (CAR + restart policy) in Figure 2. Although the figures are greatly reduced, zooming in on the pictures will not

cause distortion. From the figure, we found that CAR searches very deeply, consuming lots of CPU time but returning with no counterexample. Meanwhile, Figure 2b shows the searching path of r-CAR. It can be observed that r-CAR does not spend too much time on a certain path and restarts the search several times. Finally, a counterexample with a length of seven is found. You can find the trace of the counterexample on the top-right corner of Figure 2b. A counterexample is a path from the initial state to a final state. The initial state and final state are surrounded by red circles.



(**a**) (**b**)

Figure 2. The search paths of CAR and r-CAR (CAR + restart policy) algorithms on solving the instance "oski15a08b09s". Zooming in on the pictures will not cause distortion. (**a**) The search path of CAR algorithm. (**b**) The search path of r-CAR algorithm.

5. Experiments

5.1. Experimental Setup

We implement r-CAR based on the SimpleCAR model checker [44]. As mentioned before, the restart frequency has a significant influence on the effectiveness of the restart policy. In our conjecture, the frequent restarts in r-CAR may not preserve the advantages already achieved in original CAR, while a low frequency cannot help solve new instances. In our proposed algorithm, two parameters *threshold* and gr are introduced to determine the restart frequency in a dynamic way. We evaluate different combinations of these two parameters. We assign a relatively small value to *threshold* and a value equal to or greater than 1 to gr, e.g., $threshold = 128, gr = 1.2$, aiming to avoid the disadvantage of frequent restarts by gradually increasing the threshold after each restart.

We compare our r-CAR implementation to SimpleCAR, which implements the original CAR, and ABC [45], a prestigious model checker in the community which implements BMC and IC3/PDR and won the hardware model checking competition several times. Notably, there are different kinds of BMC implementations in ABC, and we select the one invoked by the *bmc2* command in the tool, which has the best performance based on previous evaluations [38]. Both SimpleCAR and ABC use the Minisat SAT solver [62,63] as the computation engine for model checking.

All the experiments are performed on a cluster which consists of 2304 processor cores in 192 nodes, and each node runs RedHat 6.0 with a 2.83 GHz CPU and 48 GB of memory (RAM). In the experiments, for each algorithm, the time and memory limits of each instance are set to be 1 h and 8 GB as default.

We evaluate all algorithms against 749 industrial benchmarks from the single safety property track (SINGLE) of the HWMCC in 2015 [42] and 2017 [43], whose categories are listed in Table 3. Each instance in the benchmark is an aiger model [64], which formalizes the And-Invertor Graph of a circuit together with the safety property to be verified. Latches are an important part of an aiger model. Sequential circuits have latches as state elements. In a model, the number of latches reflects the complexity of the model to a certain extent. Specifically, the state space is 2^l, where l is the number of latches. In Table 3, we grouped all benchmarks according to their source. For example, there are 180 cases, whose names are started with "6s" and provided by IBM. In these 180 cases, the smallest case contains no latches, while the largest contains 467,369 latches, with an average of 16,674 latches per case.

Table 3. The categories of benchmarks.

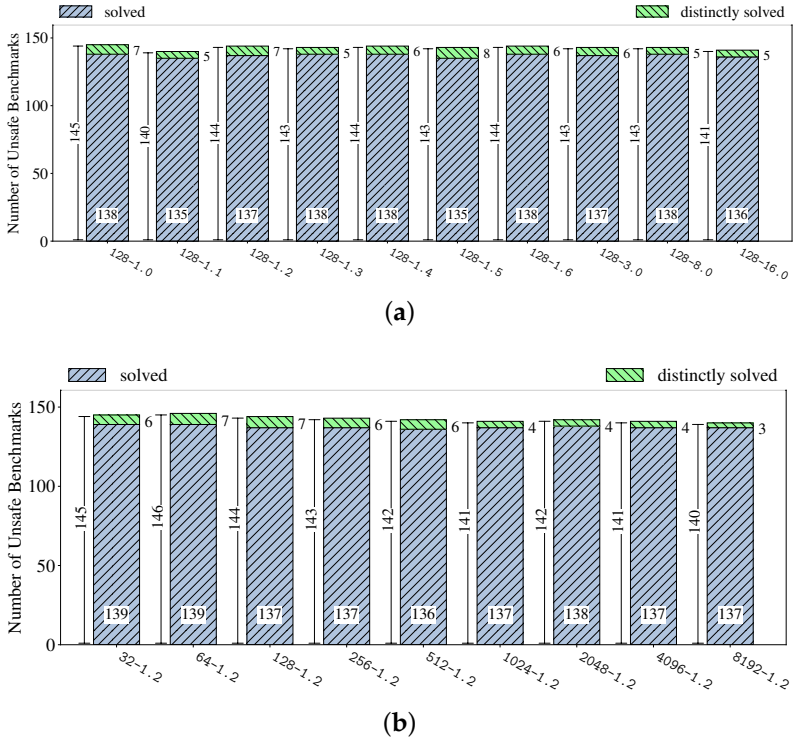
Groups	Source	Min Lacthes	Max Latches	Avg Latches	Case Numbers
6s	IBM	0	467,369	16,674	180
Intel	Intel	36	17,843	3285	43
Oski	Oski Tech	2915	25,715	15,977	171
Others	-	3	26,148	609	355
Total	-	0	467,369	8132	749

This paper focuses on unsafety checking, under which a counterexample can be provided to help identify the property violation. We use the *aigsim* tool from the Aiger package [65] to check whether the produced counterexamples are correct. We report that the counterexamples generated from all checkers pass the tests successfully.

5.2. Results

RQ1: What is the appropriate configuration for the restart policy? In the experiments, the original CAR (without restart policy) is able to solve 145 unsafe instances by providing counterexamples. To evaluate the performance of the restart policy on r-CAR, we first fix the initial *threshold* to be 128 and make the growth rate *gr* vary from 1.0 to 16.0. The number of solved and *distinctively solved* (for the meaning, see the figure) instances with the corresponding parameters are shown in Figure 3a. From the figure, the restart policy effectively expands the algorithm's diversity to find considerable new counterexamples with different configurations. In particular, the restart strategy has better results when the value of *gr* is in the range of one to two, which acquires the most amount of new instances (seven or eight). When *gr* is set to be larger than 16, it can always surprisingly find 5 "distinctly solved" instances. The reason is that these five distinctly solved instances (6s218b2950, oski15a01b03s, oski15a01b43s, oski15a10b11s, oski15a10b14s) only need one single turn of restart when the *threshold* is set to be 128, which means that *gr* plays no effect on these five cases. It is more appropriate to set *gr* in the range of 1.0 to 3.0 as there are some differences in their "distinctly solved" instances, which means we can obtain more counterexamples in total through operating the algorithm with different parameters in parallel.

We then vary the value of *threshold* from 64 to 8192 by fixing $gr = 1.2$ (as the representative), and the corresponding results are shown in Figure 3b. The restart policy performs better when the value of *threshold* is smaller than 256, under which not only more "distinctly solved" but also several unique instances are detected. For example, "oski15a08b15s" can only be found by "64–1.2", "6s351rb15" and "oc8051topo08"can only be solved by "128–1.2". In our conjecture, certain instances are sensitive to the particular combinations of the parameters that determine the frequency of the restart policy. Setting the initial *threshold* to be larger than 1024 is too large for a one-hour execution to make the restart strategy work. In short, the *threshold* is recommended to be set in the range of 32 to 256, while *gr* is recommended to be set in the range of 1.0 to 3.0 for benchmarks from HWMCC. Unfortunately, there is currently no good way to obtain the parameters suitable for specific cases. More precisely, the methods to extract the characteristics of a model are currently lacking. Larger models do not necessarily adapt to more frequent restarts. We think how to define and extract the characteristics of a model is a meaningful follow-up work.



(**a**)

(**b**)

Figure 3. Number of unsafe benchmarks solved from the experiments. The category "distinctly solved" benchmarks are solved by CAR with the corresponding restart policy but not by the original CAR. The "solved" benchmarks solved by CAR with and without the restart policy. *X*-axis 128–1.0 means the initial $threshold = 128$, $gr = 1.0$, and the same applies to others. (**a**) Results of the restart policy with the initial $threshold = 128$. (**b**) Results of the restart policy with $gr = 1.2$.

It should be highlighted that, although IC3/PDR can also perform differently by varying the parameters to generate the inductive clauses [31], it helps more significantly to prove safe instances. Meanwhile, applying the restart policy to CAR results in a better performance on solving unsafe instances, which cannot be achieved by varying different parameters inside IC3/PDR.

Due to the fact that the state space of a model grows exponentially with the number of variables, we achieve little marginal benefit when allocating more linear time to the computational task. For example, out of a total of 749 instances, the original CAR can find 145 counterexamples in 1 h and can find 147 counterexamples in 5 h, with only 2 more counterexamples in an extra 4 h. Similarly, it takes BMC 1 h to find 153 counterexamples and 5 h to find 159 counterexamples, with only 6 more counterexamples in an extra 4 h. Considering that r-CAR finds many different counterexamples when using different parameters, it is better to separately allocate computing resources to run r-CAR with different parameter combinations in parallel.

RQ2: Is restarting the policy effective? We focus on the number of counterexamples found by different algorithms, shown in Figure 4.

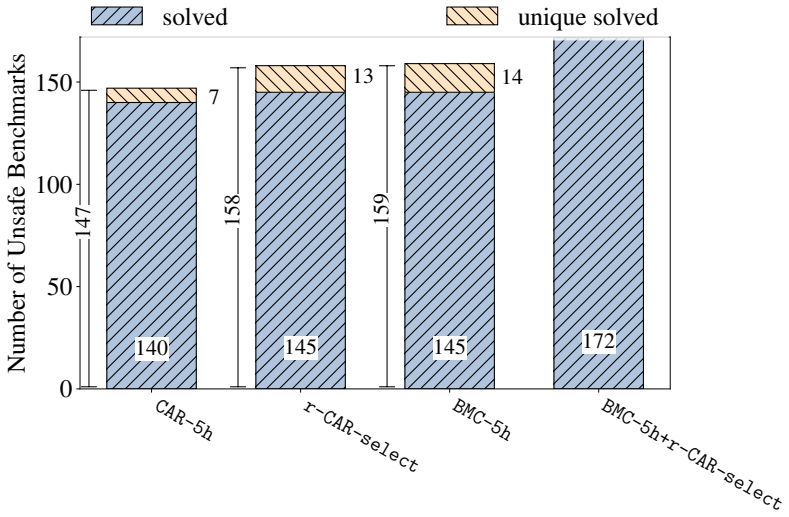
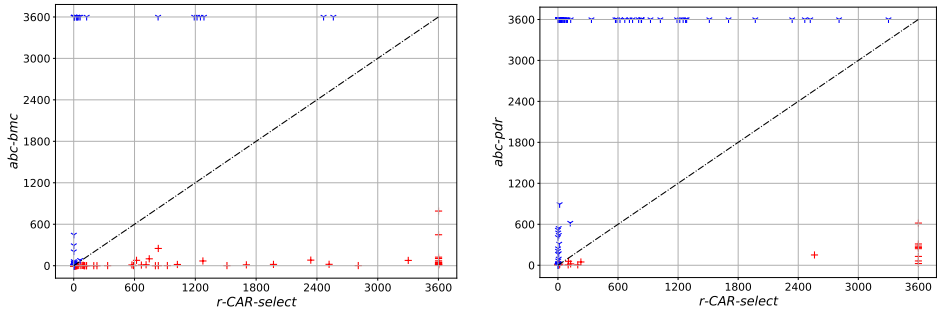


Figure 4. Comparison of the number of counterexamples.

We combine the results from the five configurations ("64–1.2", "128–1.2", "128–1.5", "128–3.0", and "256–1.2") of r-CAR with each configuration running for one hour, to represent running these five different parameter combinations in parallel, noted as *r-CAR-select*. Correspondingly, both CAR and BMC run for five hours. The BMC implementation in ABC finds 159 counterexamples in 5 h, CAR can find 147 counterexamples, and r-CAR-select can find 158 counterexamples. We can conclude that the restart policy is effective as it helps CAR find 11 more counterexamples than before (from 147 to 158). In addition, the performance of r-CAR-select is roughly equal to BMC (158 and 159), which is considered to be the most effective and widely used algorithm in bug-finding. To better compare the experiment result of r-CAR and BMC, we mark the number of counterexamples that cannot be found by the other side as "unique solved". Similarly, the "unique solved" of "CAR-5h" in Figure 4 represent the number of counterexamples that cannot be found by BMC. As we can see, r-CAR-select finds 13 counterexamples that cannot be found by BMC, much more than before (CAR finds 7), which affirms our claim that the restart policy plays a non-negligible role as a part of the portfolio to check property violation or bug-finding.

To evaluate the performance of r-CAR from another angle, we compare the time spent in solving each case between r-CAR vs. BMC and r-CAR vs. IC3/PDR, the results of which are shown in Figure 5a,b, respectively. The *X*-axis of these two figures is the time spent for the best result from r-CAR-select, while the *Y*-axis represents the time spent for BMC (resp. IC3/PDR) to solve the problem. Obviously, each point above the diagonal represents a single case in which BMC (or IC3/PDR) spends more time to find a counterexample than r-CAR-select and vice versa. It should be noted that points with the abscissa or ordinate of 3600 represent the instances that the corresponding method cannot find this counterexample in 3600 s. In Figure 5a, we can find that for those instances that can be solved by both algorithms, either BMC or r-CAR can solve most of them in a short time (less than 600 s). Even though BMC performs better in the instances that can be solved by both algorithms, r-CAR uniquely solved several instances that BMC could not solve. These two methods complement each other very well in bug-finding. Meanwhile, from the results shown in Figure 5b, r-CAR-select succeeds in solving many more instances than IC3/PDR within 3600 s and also solves them much faster. Obviously, r-CAR outperforms IC3/PDR in bug-finding.



(**a**) (**b**)

Figure 5. Comparison of time spent by r-CAR-select with other algorithms. (**a**) Comparison on time spent by r-CAR-select and BMC. (**b**) Comparison on time spent by r-CAR-select and IC3/PDR.

We then show the comparison of the overall performance among different approaches in Figure 6. We can clearly see that r-CAR-select, CAR, and BMC dominate PDR in bug-finding. Compared to BMC, CAR and r-CAR-select solve fewer instances in the early stage. The reason for this is that BMC is based on the strategy of breadth-first search, which normally operates fast at the beginning but can become slower as the depth increases. We can find that BMC solves fewer new instances after 10 min. r-CAR follows the depth-first search strategy, and the restart is triggered as soon as the depth of the searching path reaches the threshold. Therefore, the restart policy gives r-CAR more opportunities to detect the property violation as time increases, and the restart policy gradually closes the gap between r-CAR and BMC. Considering that r-CAR-select represents the result of running five parameter combinations in parallel, even if we multiply the time cost of r-CAR-select by five times, we know that after a total of about five hours, r-CAR can catch up with BMC, based on the result of Figure 4. As mentioned before, bug-finding is not the strong point of IC3/PDR, and the results of IC3/PDR shown in the figure support this claim firmly.

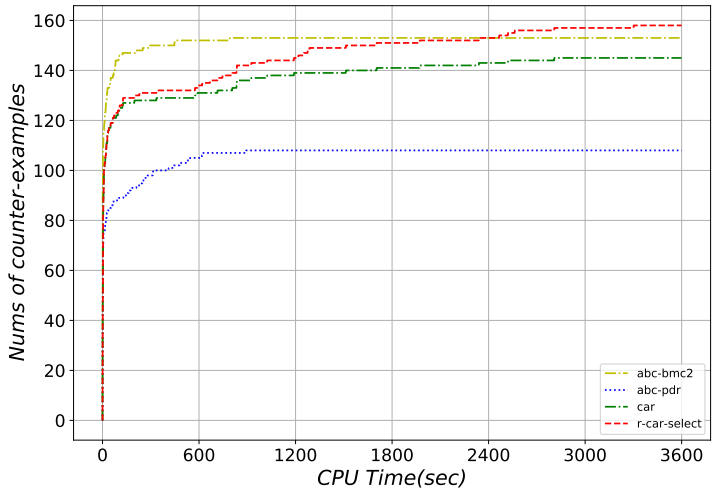


Figure 6. The overall performance of CAR, r-CAR, BMC, and PDR.

It should be clarified that r-CAR with a single configuration cannot outperform BMC. We argue that comparing r-CAR-select, which consists of five different restart configurations, to BMC is still fair because we give BMC 5 h. The BMC implementation (abc-bmc2) we select is the best one as far as we know, and abc-bmc2 can solve all instances that other BMC implementations can solve, according to our preliminary experiments. As a result, testing more BMC implementations cannot affect the conclusions made in this paper. The ability to perform differently with different configurations is the advantage of r-CAR, which cannot be achieved by either BMC or IC3/PDR.

6. Conclusions

To summarize, we apply the restart policy to CAR, aiming to get rid of the trap which occurs during the search and make the algorithm not terminate in a reasonable time. The results of the experiments show that the restart policy increases the diversity of the CAR algorithm, though there is no single configuration that can improve the overall performance significantly. The new finding of 11 unsafe instances indicates the efficiency of the restart policy in the domain. Moreover, the CAR algorithm with the restart policy can now find 13 unsafe instances with counterexamples that BMC cannot find, which enhances the ability of the current model checking portfolio.

This is the first work to understand how the restart policy performs on model checking, and our experiments result have proven the effectiveness of the restart policy. We expect that our research can be helpful to understand the performance characteristics of the restart policy. It is possible to run different parameter combinations which control the restart frequency in parallel to solve previously unsolvable cases. In future work, we plan to design more elaborate and sophisticated restart mechanisms to improve the overall performance of CAR such that it is able to outperform BMC in bug-finding with a single restart configuration. Due to the fact that different problems are sensitive to different restart frequencies, it is also interesting to introduce the learning techniques to learn the best solution for different instances.

Author Contributions: Conceptualization, M.G., X.Z. and J.L.; methodology M.G., X.Z. and J.L.; software M.G. and J.L.; investigation, M.G. and J.L.; resources, J.L.; data curation, J.L.; writing—original draft preparation, M.G. and X.Z.; writing review and editing, M.G. and J.L.; supervision, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by National Key Research and Development Program (2020AAA0107800), the Science and Technology Commitment of Shanghai, China (20PJ1403500) and National Science Foundation of China (62002118 and U21B2015).

Data Availability Statement: Publicly available dataset for HWMCC 2015 and 2017 is analyzed in this study. These data can be found here: (http://fmv.jku.at/hwmcc15/ and http://fmv.jku.at/hwmcc17/) (accessed on 24 November 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Clarke, E.; Grumberg, O.; Peled, D. *Model Checking*; MIT Press: Cambridge, MA, USA, 1999.
2. Alrajeh, D.; Kramer, J.; Russo, A.; Uchitel, S. Elaborating requirements using model checking and inductive learning. *IEEE Trans. Softw. Eng.* **2013**, *39*, 361–383. [CrossRef]
3. Heitmeyer, C.; Kirby, J.; Labaw, B.; Archer, M.; Bharadwaj, R. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng.* **1998**, *24*, 927–948. [CrossRef]
4. Ammann, P.E.; Black, P.E.; Majurski, W. Using model checking to generate tests from specifications. In Proceedings of the Second International Conference on Formal Engineering Methods, Brisbane, Australia, 9–11 December 1998; pp. 46–54.
5. Fuxman, A.; Pistore,M.; Mylopoulos, J.; Traverso, P. Model checking early requirements specifications in tropos. In Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, Toronto, ON, Canada, 27–31 August 2001; pp.174–181.
6. Visser, W.; Havelund, K.; Brat, G.; Park, S.; Lerda, F. Model checking programs. In Proceedings of the ASE 2000, Fifteenth IEEE International Conference on Automated Software Engineering, La Jolla, CA, USA, 16–19 July 2000; pp. 3–11.
7. Xie, F.; Levin, V.; Browne, J.C. Model checking for an executable subset of uml. In Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 26–29 November 2001; pp. 333–336.
8. Xie, F.; Levin, V.; Browne, J.C. Objectcheck: A model checking tool for executable object-oriented software system designs. In *Fundamental Approaches to Software Engineering*; Kutsche, R., Weber, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 331–335.
9. Beyer, D.; Keremoglu, M.E. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*; Gopalakrishnan, G., Qadeer, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 184–190.
10. Merz, S. *Model Checking: A Tutorial Overview*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 3–38.
11. Fu, X.; Bultan, T.; Su, J. Model checking xml manipulating software. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Boston, MA, USA, 12–14 July 2004; Association for Computing Machinery: New York, NY, USA, 2004; pp. 252–262.

12. Hall´e, S.; Ettema, T.; Bunch, C.; Bultan, T. *Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines*; Association for Computing Machinery: New York, NY, USA, 2010; pp. 235–244.
13. Artzi, S.; Kiezun,A.; Dolby, J.; Tip, F.; Dig, D.; Paradkar, A.; Ernst, M.D. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.* **2010**, *36*, 474–494. [CrossRef]
14. Gao, H.; Miao, H.; Chen, S.; Mei, J. Applying bounded model checking to verifying web navigation model. In *Computer and Information Science 2011*; Lee, R., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 1–15.
15. Witkowski, T.; Blanc, N.; Kroening, D.; Weissenbacher, G. Model checking concurrent linux device drivers. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, Atlanta, GA, USA, 5–9 November 2007; Association for Computing Machinery: New York, NY, USA, 2007; pp. 501–504.
16. Kim, M.; Kim, Y.; Kim, H. Unit testing of flash memory device driver through a sat-based model checker. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 15–19 September 2008; pp. 198–207.
17. Kim, M.; Kim, Y.; Kim, H. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Trans. Softw. Eng.* **2011**, *37*, 146–160. [CrossRef]
18. Dwyer, M.B.; Carr, V.; Hines, L. Model checking graphical user interfaces using abstractions. *SIGSOFT Softw. Eng. Notes* **1997**, *22*, 244–261. [CrossRef]
19. Dwyer, M.B.; Robby; Tkachuk, O.; Visser, W. Analyzing interaction orderings with model checking. In Proceedings of the 19th International Conference on Automated Software Engineering, Linz, Austria, 24 September 2004; pp. 154–163.
20. Haydar, M.; Boroday, S.; Petrenko, A.; Sahraoui, H. Properties and scopes in web model checking. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 7–11 November 2005; Association for Computing Machinery: New York, NY, USA, 2005; pp. 400–404.
21. Artho, C.; Garoche, P. Accurate centralization for applying model checking on networked applications. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan, 18–22 September 2006; pp. 177–188.
22. Artho, C.; Leungwattanakit, W.; Hagiya, M.; Tanabe, Y.; Yamamoto, M. Cache-based model checking of networked applications: From linear to branching time. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, 16–20 November 2009; pp. 447–458.
23. ortler, T.V.; R¨ulke, S.; Hofstedt, P. Bounded model checking of contiki applications. In Proceedings of the 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), Tallinn, Estonia, 18–20 April 2012; pp. 258–261.
24. Eisler, S.; Scheidler, C.; Josko, B.; Sandmann, G.; Stroop, J. Preliminary results of a case study: Model checking for advanced automotive applications. In *International Symposium on Formal Methods*; Fitzgerald, J., Hayes, I.J., Tarlecki, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 533–536.
25. Gligoric, M.; Majumdar, R. Model checking database applications. In *Tools and Algorithms for the Construction and Analysis of Systems*; Piterman, N., Smolka, S.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 549–564.
26. Song, F.;Touili, T. Pushdown model checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*; Flanagan, C., König, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 110–125.
27. Karna, A.K.; Chen, Y.; Yu, H.; Zhong, H.; Zhao, J. The role of model checking in software engineering. *Front. Comput. Sci.* **2016**, *12*, 642–668. [CrossRef]
28. Pnueli, A. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), Providence, RI, USA, 31 October–2 November 1977; pp. 46–57.
29. Kupferman, O.; Vardi, M. Model checking of safety properties. In *International Conference on Computer Aided Verification*; Series Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1633, pp. 172–183.
30. McMillan, K. *Symbolic Model Checking*; Kluwer Academic Publishers: New York, NY, USA 1993.
31. Griggio, A.; Roveri, M. Comparing different variants of the IC3 algorithm for hardware model checking. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2016**, *35*, 1026–1039. [CrossRef]
32. Biere, A.; Cimatti, A.; Clarke, E.; Fujita, E.; Zhu, Y. Symbolic model checking using SAT procedures instead of BDDs. In Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, New Orleans, LA, USA, 1 June 1999; IEEE Computer Society: Washington, DC, USA, 1999; pp. 317–320.
33. Biere, A.; Cimatti, A.; Clarke, E.; Zhu, Y. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Series Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1579.
34. McMillan, K. Interpolation and SAT-based model checking. In *Computer Aided Verification*; Hunt, J., Warren, A., Somenzi, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2725, pp. 1–13.
35. Bradley, A. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*; LNCS Series; Jhala, R., Schmidt, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6538, pp. 70–87.
36. Een, N.; Mishchenko, A.; Brayton, R. Efficient implementation of property directed reachability. In Proceedings of the FMCAD, Austin, TX, USA, 30 October–2 November 2011; pp. 125–134.
37. Li, J.; Zhu, S.; Zhang, Y.; Pu, G.; Vardi, M.Y. Safety Model Checking with Complementary Approximations. In Proceedings of the ICCAD, Irvine, CA, USA, 13–17 November 2017.

38. Li, J.; Dureja, R.; Pu, G.; Rozier, K.Y.; Vardi, M.Y. SimpleCAR: An Efficient Bug-Finding Tool Based on Approximate Reachability. In *Computer Aided Verification*; Chockler, H., Weissenbacher, G., Eds.; Springer: Cham, Switzerland, 2018; pp. 37–44.
39. Dureja, R.; Li, L.; Pu, G.; Vardi, M.Y.; Rozier, K.Y. Intersection and rotation of assumption literals boosts bug-finding. In *Verified Software. Theories, Tools, and Experiments*; Chakraborty, S., Navas, J.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 12031, pp. 180–192.
40. Vizel, Y.; Weissenbacher, G.; Malik, S. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE* **2015**, *103*, 2021–2035. [CrossRef]
41. Biere, A. Adaptive restart strategies for conflict driven sat solvers. In *Theory and Applications of Satisfiability Testing—SAT 2008*; Büning, H.K., Zhao, X., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 28–33.
42. HWMCC 2015. 2015. Available online: http://fmv.jku.at/hwmcc15/ (accessed on 24 November 2021).
43. HWMCC 2017. 2017. Available online: http://fmv.jku.at/hwmcc17/ (accessed on 24 November 2021).
44. SimpleCAR. 2021. Available online: https://github.com/lijwen2748/simplecar/releases/tag/v0.1 (accessed on 24 November 2021).
45. Brayton, R.; Mishchenko, A. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 24–40.
46. Green, C. Application of theorem proving to problem solving. In Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, WA, USA, 7–9 May 1969; pp. 219–240.
47. Clarke, E.; Emerson, E.; Sifakis, K. Model checking: Algorithmic verification and debugging. *Commun. ACM* **2009**, *52*, 74–84. [CrossRef]
48. Clarke, E.; Emerson, E.; Sistla, A. Automatic verification of finitestate concurrent systems using temporal logic specifications. *Acm Trans. Program. Languagues Syst.* **1986**, *8*, 244–263. [CrossRef]
49. Copty, F.; Fix, L.; Fraer, R.; Giunchiglia, E.; Kamhi, G.; Tacchella, A.; Vardi, M. Benefits of bounded model checking at an industrial setting. In Proceedings of the 13th International Conference on Computer Aided Verification, Paris, France, 18–22 July 2001; Series Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2102, pp. 436–453.
50. Holzmann, G. The model checker SPIN. *IEEE Trans. Softw. Eng.* **1997**, *23*, 279–295. [CrossRef]
51. Bryant, R. Graph-based algorithms for Boolean-function manipulation. *IEEE Trans. Comput.* **1986**, *100*, 677–691. [CrossRef]
52. Burch, J.; Clarke, E.; McMillan, K.; Dill, D.; Hwang, L. Symbolic model checking: 1020 states and beyond. In Proceedings of the 5th IEEE Symposium on Logic in Computer Science, Philadelphia, PA, USA, 4 June 1990; pp. 428–439.
53. Xu, J.; Williams, M.; Mony, H.; Baumgartner, J. Scalable reachability analysis via automated dynamic netlist-based hint generation. *Form. Methods Syst. Des.* **2014**, *45*, 144–164. [CrossRef]
54. Malik, S.; Zhang, L. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* **2009**, *52*, 76–82. [CrossRef]
55. Sheeran, M.; Singh, S.; Stalmarck, G. Check safety properties using induction and a SAT-solver. In Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, Austin, TX, USA, 15–17 November 2000; Series Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1954, pp. 108–125.
56. Burch, J.R.; Clarke, E.M.; Long, D.G. Symbolic model checking with partitioned transition relations. In Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, Edinburgh, UK, 20–22 August 1991; pp. 49–58.
57. Yu, Y.; Subramanyan, P.; Tsiskaridze, N.; Malik, S. All-SAT using minimal blocking clauses. In Proceedings of the 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, 5–9 January 2014; pp. 86–91.
58. Mann, M. Pono: A Flexible and Extensible SMT-Based Model Checker. In *Computer Aided Verification, CAV*; Silva, A., Leino, K.R.M., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; Volume 12760.
59. Goel, A.; Sakallah, K. AVR: Abstractly Verifying Reachability. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*; Biere, A., Parker, D., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2020; Volume 12078.
60. Lange, T.; Neuhäußer, M.R.; Noll, T. IC3 software model checking. *Int. J. Softw. Tools Technol. Transfer.* **2020**, *22*, 135–161. [CrossRef]
61. Winterer, F.; Seufert, T.; Scheibler, K.; Teige, T.; Scholl, C.; Becker, B. ICP and IC3 with Stronger Generalization. In Proceedings of the MBMV 2021; 24th Workshop, München, Germany, 18–19 March 2021; VDE: Berlin, Germany, 2021; pp. 1–12.
62. Minisat 2.2.0. Available online: https://github.com/niklasso/minisat (accessed on 24 November 2021).
63. Eén, N.; Sörensson, N. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*; Giunchiglia, E., Tacchella, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 502–518.
64. AIGER Format. 2007. Available online: http://fmv.jku.at/aiger/FORMAT (accessed on 24 November 2021).
65. AIGER Tools. Available online: http://fmv.jku.at/aiger/aiger-1.9.9.tar.gz (accessed on 24 November 2021).