Applying the Restart Policy to Hardware Model Checking Boosts Bug-Finding

***Abstract*—Model checking is an efficient technique for verification of hardware designs. However, its performance, particularly on bug-finding, needs to be improved to meet the industrial requirement. In this paper, we focus on improving CAR, a recently proposed model checking technique, by applying simple restart policies to the algorithm. Our results show that, out of the 749 industrial instances, CAR with different restart policies is able to find 17 instances that the state-of-the-art technique BMC can not find, and solve more than 12 instances than its previous version. The new algorithm succeeds to contribute to the current model-checking portfolio in practice.**

***Index Terms*—Model Checking, CAR, BMC, SAT, Bug-finding**

I. Introduction

In the hardware design community, model checking [] has emerged as a popular technique for both bug-finding and correctness proof of hardware systems (circuits). Given a hardware design M as the model and the formal specification (property) P, model checking checks whether P holds for all behaviours 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 to represent the property violation is detected, 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, the length of the counterexample becomes finite. As a result, the safety model checking can be reduced to the reachability analysis problem [], and we focus on the safety model checking in this paper.

Although model checking has been widely used in 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 []. Although invented in nearly three decades ago, Bounded Model Checking (BMC) [1] [8] is still considered to be the most efficient technique for detecting bugs. Meanwhile, Interpolation Model Checking (IMC) [] and IC3 [], or Property Directed Reachability (PDR) [], are shown more powerful to prove correctness. Therefore, a portfolio of model checking techniques is often maintained by the industry companies to solve different problems.

Recently, a new model-checking algorithm named Complementary Approximate Reachability (CAR) [8], was proven to complement BMC on bug-findings 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 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 [], 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 if failed, collect the relevant information to refine the over-approximate state sequence. 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 evidence has shown Backward-CAR is better than Forward-CAR [], we follow the observation and focus on improving Backward-CAR. In the rest of the paper, all mentions of "CAR" represent Backward-CAR.

Although CAR has been shown the advantage of detecting bugs for safety model checking and outperforms IC3/PDR on bug-finding, it cannot solve as many unsafe instances (those with bugs) as BMC in the current stage []. Looking into the state paths generated by CAR during the search, the observation comes up that, the depth-first strategy may lead the algorithm to get trapped for those unsafe cases unable to solve. As a result, it is almost impossible for the algorithm to locate the bad state but only waste the computation sources. Such similar phenomenon occurs on solving the satisfiability of Boolean formulas (SAT) [], in which the search can also be trapped 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 []. Their experiments show that such simple strategy is very efficient to help speedup SAT solving computation, particularly for those satisfiable instances.

Inspired from the result achieved by applying the restart policy to modern SAT solvers, we leverage the similar idea to enhance the performance of CAR on bug-finding. In our designation, the restart policy is invoked as soon as the size of new elements of the over-approximate state sequence in a single search, reaches $k * t$ 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 proportion p during the search. That means, if the current threshold is t and the proportion is p, the threshold will be updated to be $(t * p)$ when the restart is invoked next time. Moreover, the search will be restarted next time as soon as the size of new elements of the over-approximate sequence reaches $(k*t*p)$. 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 states information collected in the current search and start a new one immediately.

We conduct a comprehensive experimental evaluation on the 749 industrial instances from Hardware Model Checking Competition 2015 [] and 2017 []. We implement our new algorithm based on the SimpleCAR model checker [], and compare the bug-finding performance to the original CAR in SimpleCAR as well as the BMC and PDR algorithms that are implemented in the ABC model checker []. The results show that, given the same time and hardware sources, our new algorithm can solve 17 more unsafe instances than BMC+PDR and 12 more than the original CAR, by feeding the restart policy with different frequencies. Although the new algorithm with a single configuration does not have a better performance than the original CAR, it does help increases the diversity to solve more instances, and therefore, increases the power of the current model-checking portfolio in the industry.

II. PRELIMINARIES

A. Boolean transition system

Modern SAT-based model checking techniques consider the *Boolean transition system* as the model. A *Boolean transition system* Sys is 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, iff $s_1 \cup s_2^{'} \models T$, denoted $(s_1, s_2) \in T$.

A finite state squence $s_0, s_1, ..., s_k$ 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. The set of all states that are reachable from the initial states I is 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* if P holds for all the reachable states of Sys. Otherwise, the system is *unsafe*. We call the state violating P is 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., $R(X) = \{s^{'} | (s, s^{'}) \in T \textit{ for } s \in X\}$. 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 $i > 1$.

TABLE I
THE SUMMARY OF KEY STRUCTURES IN 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	-	$\exists i, B_{i+1} \subseteq \bigcup_{0<j<i} B_j$
UnSafety Check	$\exists i, F_i \cap \neg P \neq \emptyset$	-

We call a Boolean variable a or its negation $\neg a$ as a *literal*. Let L be a set of literals. A *cube* is Boolean formula with the form of $\bigwedge l$ where $l \in L$. Analogously, a *clause* is 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.

B. SAT calls with assumptions and unsatisfiable core

The CAR algorithm frequently invokes the SAT calls whose inputs have the form of $A \wedge B$, where B is a CNF formula[1] and A is a cube. We use the notation $SAT(A, B)$ to represent such SAT query 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 calling, 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, can be returned by the SAT solver.

C. An Overview of Complementary Approximate Reachability

Derived from the traditional reachability analysis, CAR can also perform in both the forward and backward directions. As mentioned before, Backward CAR has been shown better than Forward CAR []. 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 the set of states reachable from I in i steps. Such under-approximate sequence is called the *F-sequence*. Also, 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 if it can reach $\neg P$ in i steps. 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 the constraints and safety/unsafety checking conditions are shown in TABLE I, and more details are referred to the next section.

Alg. 1 Main Procedures of CAR

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

12: procedure UNSAFECHECK$(s, i, k)$
13:     if RESTARTPOINT$(k, threshold, count)$
14:         $restart := true$;
15:         return false
16:     Cube $\hat{s}$ := REORDER$(s)$;
17:     while $SAT(\hat{s}, T \wedge B_i^{'})$ do
18:         if $i = 0$ then return true;
19:         Cube $t$ = get_assignment();
20:         $F_{j+1} := F_{j+1} \cup t$ supposing $s$ is in $F_j$ $(j \geq 0)$;
21:         if UNSAFECHECK $(t, i-1, k)$ then return true;
22:     Cube $c$ :=get_unsat_core()
23:     $count := count + 1$;
24:     $B_{i+1} := B_{i+1} \cap \neg c$;
25:     return false;

26: procedure SAFECHECK$(k)$
27:     $i = 0$;
28:     while $i < k$ do
29:         if not $SAT(\emptyset, \neg(B_{i+1} \Rightarrow (\bigvee_{0 \leq j \leq i} B_j)))$
30:             return true;
31:     return false;
```

III. Algorithm Design

A. Algorithmic Description of CAR

The pseudo-codes of CAR's main procedures are shown in Alg. 1. The algorithm takes a system $Sys = (V, I, T)$ and a safety property P as the inputs, and outputs *safe* if an invariant is detected (Line 10) or *unsafe* if a counterexample is found (Line 6). 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 11 of Alg. 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 2, according to the rules in TABLE I. The while loop from Line 4 to Line 11, first calls the *UNSAFECHECK* procedure to search new states and returns unsafe if a bad state is found. Meanwhile, the F- and B-sequence can be updated during the search inside the procedure. If *UNSAFECHECK* proceeds but no bad states 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 while loop terminates as soon as one of the above two procedures returns. A summary of procedures used in CAR is listed below.

- **PICKSTATE** at Line 5 takes the F-sequence as input and uses some decision strategies to enumerate and select a state from the sequence. The procedure returns an empty set $\emptyset$ if there is no available state in the sequence.
- **REORDER** at Line 16 takes a state as the input. Inspired from 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 s to generate its new copy $\hat{s}$ (Cube $\hat{s}$ at Line 16), which is then transferred to the SAT solver as assumptions. For example, given a state $s = l_1 \cap l_2 \cap l_3 \cap l_4$, the returned state $\hat{s}$ may be $l_3 \cap l_4 \cap l_1 \cap l_2$ 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).
- **get_assignment** at Line 19 returns a satisfying assignment of the input formula if the SAT query result is satisfiable. A new state t can be extracted from the assignment.
- **get_unsat_core** at Line 22 generates an unsatisfiable core, which is a subset of the assumptions in the current SAT call, if the query result is unsatisfiable.
- **UNSAFECHECK** from Line 12 to Line 25 takes a state s, an integer i representing the current working level of the B-sequence, and an integer k representing the maximum depth of the B-sequence as inputs. The procedure first reorders the input state s to $\hat{s}$ through REORDER(s), and then invokes a SAT call $SAT(\hat{s}, T \cap B_i^{'})$ to check whether state s can directly reach states in B_i. If the result is unsatisfiable, it calls get_unsat_core() to get 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 can not reach 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 18), that is, state s can reach the bad states in B_i. As the state s was selected from the F-sequence, a counterexample is found and the procedure returns true. If the SAT call is satisfiable with $i > 0$ (Line 19-21), we invoke get_assignment() to get another state t, which is added into the F-sequence as the successor of s, and recursively invoke UNSAFECHECK$(t, i-1, k)$.
- **SAFECHECK** at Line 26-31 takes an integer k, the depth of the B-sequence, as the input. By enumerating i from 0

[1]The Boolean formula with the form of $\bigwedge c_i$ where each c_i is a clause.

to k, the procedure checks whether all the states in B_{i+1} have been contained in the union set of $B_0, B_1, ..., Bi$. If this is the case, We can assert that all the states that can reach to bad state $B_0 = \neg P$ 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 I.

- **RestartPoint** at Line 13 returns true if CAR is ready to restart the state search according to the restart policies introduced in the next section.

B. 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 get trapped due to the improper assignments to the variables of the Boolean formula. Under such scenarios, it is almost impossible to find the final result but only wastes the computation sources. As a result, it is reasonable to abandon the current search path and restart it again with different variable assignments. Studies have shown that such simple strategy turns out to be very efficient to help solve more satisfiable instances []. It is surprising to see that CAR also suffers from a similar problem to get trapped during state search, and the idea comes out straightforward that applying the restart policy to CAR.

As shown in Alg. 1, the texts in red are pseudo-codes added to integrate the start policy in CAR. We choose the number of unsatisfiable cores generated during the search as the criteria to invoke the restart policy. The insight is that too many UCs are computed in a single search probably leads it to be trapped. To implement that, a counter variable *count* is introduced to record such number in the current search, and its value increases every time a new UC is computed (Line 23). Also, a threshold that can be dynamically updated is provided and the restart policy is triggered as soon as the condition $count > threshold$ becomes true. Notably in the main loop of CAR, we use a flag $restart$ to control whether the restart policy should be triggered (Line 8), whose value is updated based on the return value of the *RestartPoint* procedure (Line 13).

Once the restart policy is triggered, CAR needs to abandon the current search and start over again. However, the frequency to restart is a key reason to affect 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 solve 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 3) and then can be updated based on a *growth rate* gr. Parameter $threshold_0$ determines the initial frequency of the restart policy through the equation $bound = threshold_0 * (k+1)$, where $(k+1)$ representing the length of current B-sequence.

Alg. 2 The Restart Policy

```
1: procedure RESTARTPOINT($k, threshold, count$)
2:     Let $bound := (k+1) * threshold$;
3:     if $count > bound$ then return true;
4:     else return false;

5: procedure RESTART
6:     $count := 0$;
7:     $threshold = threshold * gr$;
8:     BACKTRACK();
```

How the restart frequency dynamically updates depends on gr. After each time the CAR algorithm triggers the restart policy, $threshold$ is multiplied by gr, leading to the increment of the restart's boundary.

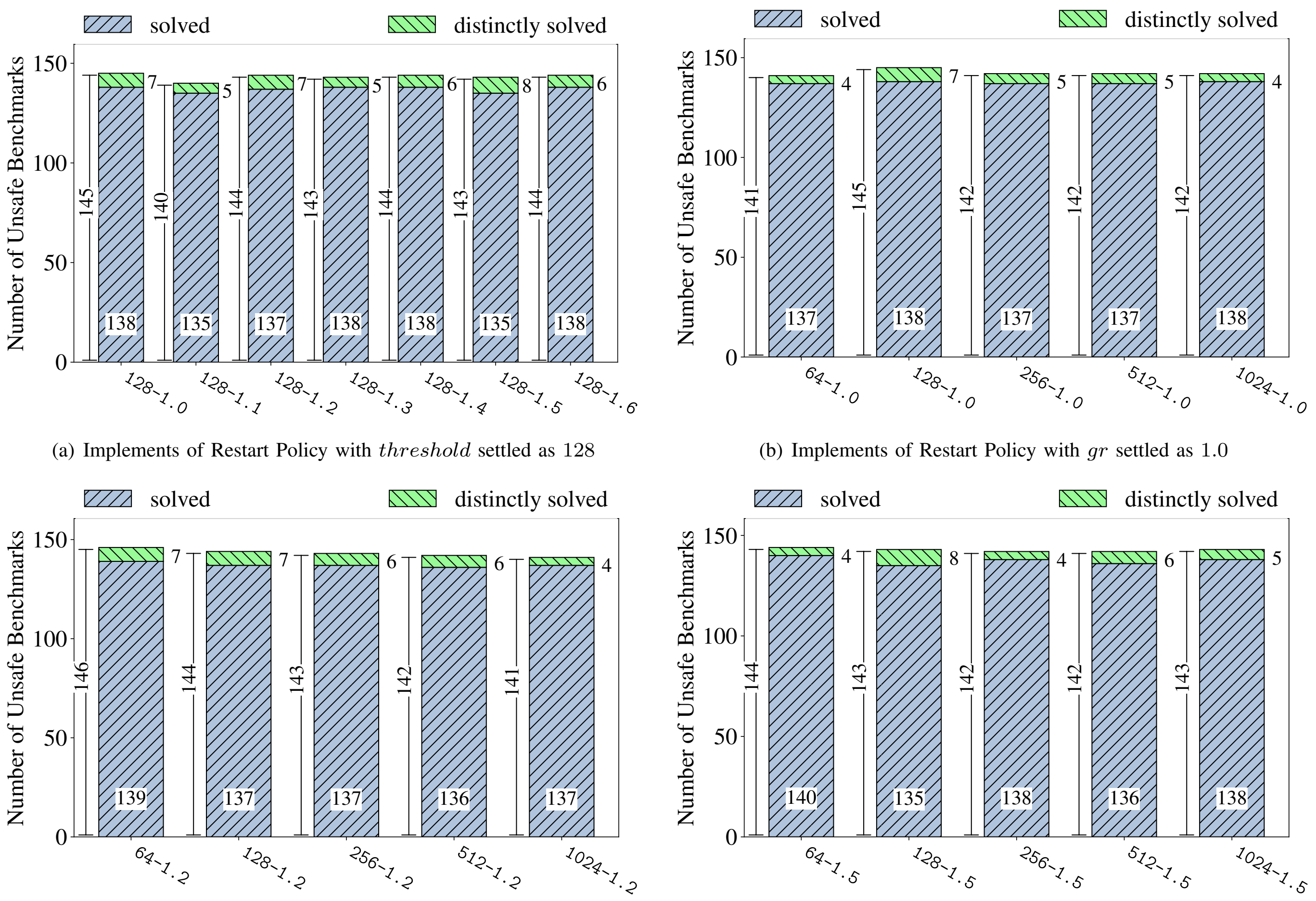
In the UNSAFECHECK procedure, RESTARTPOINT is set to judge whether it is time to restart. The procedure takes an integer k representing the length of B-sequence and an integer $count$ which counts the number of new unsatisfiable cores (line 23) generated in current search. Considering the B-sequence is an over-approximate sequence, 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 B-sequence into consideration, and the restart boundary is the product of the length of B-sequence and $threshold$. As soon as $count$ is larger than $bound$, RESTARTPOINT returns true and the $restart$ flag becomes true, which makes the procedure UNSAFECHECK terminate with output "false" (line 13-15). After the restart point is reached, all of current recursive calls of UNSAFECHECK are returned as false, leading to the termination of loop from Line 5-7 as well as the entry to the procedure RESTART at Line 11.

The RESTART procedure at Line 5 of Alg. 2 resets the unsatisfiable core counter $count$ (line 6 in Alg. 2) and enlarge the $threshold$ with growth rate gr (Line 7 in Alg. 2). Compared to the previous search from initial states I, we have updated B_i with a certain number of unsatisfiable cores, which probably results in a different search path. The procedure BACKTRACK contains the process of returning to initial states I, eliminating the F-sequence to release the memory, and some auxiliary work like the reconstruction of the SAT solver due to the memory consideration. Details are omitted here.

IV. Experiments

A. Experimental Setup

We implement the restart policy to the SimpleCAR model checker []. As mentioned before, the restart frequency has a significant influence on the effectiveness of the restart policy. In our conjecture, the frequent restarts in CAR may not preserve the advantages already achieved, while a low frequency of the algorithm cannot help solve new instances. In our proposed algorithm, two parameters $threshold$ and gr are introduced to determine the restart frequency in a



(a) Implements of Restart Policy with $threshold$ settled as 128

(b) Implements of Restart Policy with gr settled as 1.0

(c) Implements of Restart Policy with gr settled as 1.2

(d) Implements of Restart Policy with gr settled as 1.5

Fig. 1. Number of unsafe benchmarks solved by CAR after applying the restart policy. 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 $threshold = 128$, $gr = 1.0$ and the same applies to others.

dynamic way. We evaluate different combinations of these two parameters. We assign a relatively small value to $threshold$ and assign gr a value equal to or greater than 1 to gr, i.e., $threshold = 128, gr = 1.2$, aiming to avoid the disadvantage of frequent restarts by gradually increasing the threshold after each restart.

We compare our results to those from the original CAR implemented in SimpleCAR, as well as from BMC and PDR that are integrated in ABC tool [], which is a prestigious model checker in the community and won the hardware model checking competition many times. Notably, there are kinds of BMC implementations in ABC, and we select the *bmc2* which has the better performance based on previous evaluations []. Both SimpleCAR and ABC use the Minisat SAT solver [] as the computation engine for model checking.

All the experiments are performed on a cluster consisting of 2304 processor cores in 192 nodes running REDHat 6.0 with a 2.83GHz CPU and 48GB of memory (RAM). In the experiments, the time limit is set to be one hour and the memory-use is 8 GB, for each instance. We evaluate all algorithms against 749 industrial benchmarks from the single safety property track (SINGLE) of the HWMCC in 2015 [] and 2017 []. Each instance in the benchmark is an aiger model, which formalizes the And-Invertor Graph [] of a circuit together with the safety property to be verified.

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

B. Results

1) Comparison to original CAR: In the experiments, the original CAR is able to solve 145 unsafe instances without counterexamples. To evaluate the performance of the restart policy on CAR from this paper, we first fix $threshold$ to be 128 and make the growth rate gr vary from 1.0 to 1.6. The number of solved and *distinctively solved* (The meaning see the figure.) instances with the corresponding parameters

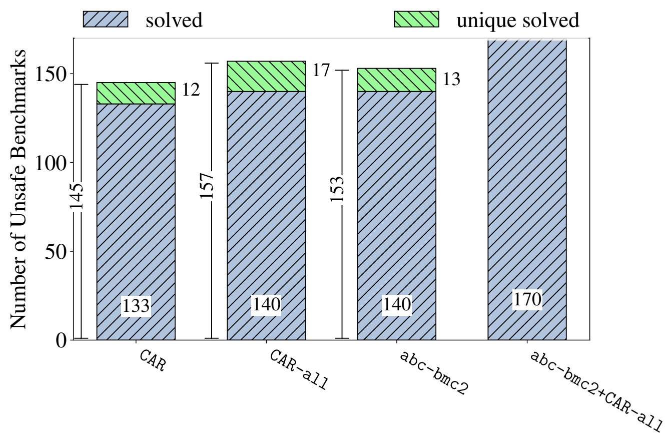


Fig. 2. Number of unsafe benchmarks solved. The "uniquely solved" benchmarks are not solved by any other algorithm category, except the comparison of CAR and CAR-all.

are shown in Fig. 1(a). From the figure, the restart policy effectively expands the algorithm's diversity by finding considerable new counterexamples with different configurations. In particular, the restart strategy has better results when gr are set to be 1.0, 1.2 or 1.5, which are able to gain the most new instances (7 or 8).

We then vary $threshold$ from 64 to 1024 by fixing gr to be 1.0, 1.2 and 1.5 respectively. The corresponding results are shown in Fig. 1(b)-1(d). Combining the results from Fig. 1(a)-1(d), the restart policy performs the best when the value of $threshold$ is around 128, 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" can only be solved by "128-1.2" and "oc8051topo08" can only be solved by "128-1.5". Setting $threshold$ to the value 1024 seems to be too large for a one-hour experiment to make the restart strategy work.

It should be highlighted that, although IC3/PDR can also perform differently by varying the paremeters to generate the inductive clauses [], it help more to prove safe instances. Meanwhile, applying the restart policy to CAR results in a better performance on solving unsafe instances that can produce the counterexamples.

2) Comparison to original CAR and BMC : As shown in Fig. 2, the BMC implementation in ABC solve 153 unsafe instances. CAR-all includes the results from CAR with and without the restart policy by choosing different parameters ($threshold$ and gr). In summary, the restart policy helps CAR find 12 more counterexamples (from 145 to 157) and 4 more "unique solved" that can only solved by CAR. Also, CAR-all solves 157 instances in total, which outperforms BMC (the amount is 153) and gains 15 instances that can not be found by BMC. The virtual combination of CAR-all and BMC solves 170 instances, which affirms that the restart policy plays a non-negligible role as a part of the portfolio for hardware model checking.

V. Concluding Remarks

Applying the restart policy on Hardware Model Checking Boosts intends to avoid waiting too long in the hope of there will be a future run will succeed finally, while the top-level decision is not so appropriate. The results of the experiments have not met our expectations due to the fact that we have not found many new counterexamples in a single run. But the new finding of 12 bugs indicates that the restart policy thus works effectively in the domain of hardware model checking. With the help of restart policy, the CAR algorithm can now find 17 instances with counterexample out of 749 industrial cases that the BMC can not find, which also increases the ability of the current hardware model checking portfolio. We view our work as a beginning step to a better understanding of the role restart policy can play in hardware model-checking algorithms.

In future work, after confirmed whether the restart policy works in the domain of hardware model checking, we plan to design more elaborate and sophisticated mechanisms to enhance the model checking algorithm's performance. Due to the fact that different cases are sensitive to various levels of the frequency of the restart policy, the work of parameter learning is also meaningful.

References

[1] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

[2] Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Proceedings of the 36th International Conference on Computer-Aided Design, 2017, pp. 95–100

[3] Li, Jianwen, et al. "Simplecar: An efficient bug-finding tool based on approximate reachability." International Conference on Computer Aided Verification. Springer, Cham, 2018.

[4] Huang, Jinbo. "The Effect of Restarts on the Efficiency of Clause Learning." IJCAI. Vol. 7. 2007.

[5] Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in ComputerAided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, TX (2011), http://dl.acm.org/citation.cfm?id=2157654.2157675

[6] Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[7] McMillan, K.L.: Interpolation and sat-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[8] Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using sat procedures instead of bdds (1999), http://doi.acm.org/10.1145/309847.309942

[9] Audemard, Gilles, and Laurent Simon. "Refining restarts strategies for SAT and UNSAT." International Conference on Principles and Practice of Constraint Programming. Springer, Berlin, Heidelberg, 2012.

[10] HWMCC 2015. http://fmv.jku.at/hwmcc15/

[11] HWMCC 2017. http://fmv.jku.at/hwmcc17/