Revisiting Assumptions Ordering in CAR-Based Model Checking

Yibo Dong[1], Yu Chen[2], Jianwen Li[1], Geguang Pu[1], Ofer Strichman[3]
[1]East China Normal University, China, prodongf@gmail.com, {jwli, ggpu}@sei.ecnu.edu.cn
[2]Chuzhou University, China, chenyu@chzu.edu.cn
[3]Technion, Isreal, ofers@technion.ac.il

***Abstract*—Model checking is an automatic formal verification technique that is widely used in hardware verification. The state-of-the-art complete model-checking techniques, based on IC3/PDR and its general variant CAR, are based on computing symbolically sets of under- and over-approximating state sets (called 'frames') with multiple calls to a SAT solver. The performance of those techniques is sensitive to the order of the assumptions with which the SAT solver is invoked, because it affects the *unsatisfiable cores* that it emits if the formula is unsatisfiable — which the solver emits when the formula is unsatisfiable — that crucially affect the search process. This observation was previously published in [10], where two partial assumption ordering strategies, *intersection* and *rotation* were suggested (partial in the sense that they determine the order of only a subset of the literals). In this paper we extend and improve these strategies based on an analysis of the reason for their effectiveness. We prove that intersection is effective because of what we call *locality* of the cores, and our improved strategy is based on this observation. We conclude our paper with an extensive empirical evaluation of the various ordering techniques. One of our strategies, Hybrid-CAR, which switches between strategies at runtime, not only outperforms other, fixed ordering strategies, but also outperforms other state-of-the-art bug-finding algorithms such as** ABC-BMC**.**

***Index Terms*—Hardware Verification, Model Checking, Complementary Approximate Reachability**

I. Introduction

Model checking is a widely used formal verification technique in hardware design, where it ensures that a system model satisfies given safety properties. For safety properties, the model checking problem reduces to reachability analysis [7], and various techniques have been developed to address this, including Bounded Model Checking (BMC) [1], Property Directed Reachability (PDR) [3], and Complementary Approximate Reachability (CAR) [15]. Among these methods, CAR has shown great potential by solving instances that BMC and PDR cannot, but its performance remains sensitive to the way assumption literals are ordered when passed to the SAT solver. This sensitivity arises because the order in which literals are processed affects the generation of unsatisfiable cores (UCs), which are essential for driving the state-space exploration. This paper focuses on improving the quality of those UCs, which accelerates the narrowing process of the O-frames[1].

To improve UC quality in CAR, it is essential to understand how SAT solvers generate UCs. In each SAT call in CAR, a formula of the form $\bigwedge_{l\in\mathcal{A}} l \wedge \phi$ is processed, where $\mathcal{A}$ is a sequence of *assumption* literals and ϕ is a Boolean formula in Conjunctive Normal Form (CNF). Modern CDCL-based SAT solvers handle $\mathcal{A}$ by positioning its literals in order as the initial decisions, and perform as usual Boolean Constraint Propagation (BCP). A contradiction emerging from those decisions indicate unsatisfiability. In that case the solver outputs a UC, which is a subset of $\mathcal{A}$ that is sufficient to render ϕ unsatisfiable.

The order of literals in $\mathcal{A}$ impacts the resulting UC and hence the overall performance of CAR. Previous work developed assumption-ordering strategies that improve the quality of UCs. For example, IC3REF [13] sorts literals by their frequency, while SIMPLECAR [14], a CAR implementation, uses two strategies: *Intersection*, which prioritizes literals in both the current state and the previous UC, and *Rotation*, which prioritizes literals found in previously failed states [10]. These strategies empirically enhance bug-finding performance. Indeed, SIMPLECAR is one of the baseline approaches against which we evaluate our improvements. Additionally, we compare our contributions to the leading BMC implementations and recent optimizations of CAR [19].

Our contributions are as follows:

1) We revisit one of the two heuristics proposed in [10], called *Intersection*, and suggest an explanation for its effectiveness. Briefly, we show that it leads to finding proofs of unsatisfiability faster because of what we call the *locality* of the cores. Based on this observation, we propose an extension of this technique that improves locality, and decides on the order of more literals comparing to the original version of *Intersection*. We also show how it affects a combination of *locality* with another strategy from [10] called *Rotation*.
2) We observe that the last added literal to a UC, which we call the *conflict literal*, is necessarily part of a minimal core. We empirically show that prioritizing this literal reduces the time to find proofs.
3) We introduce Hybrid-CAR, a dynamic strategy that alternates between literal-ordering strategies at runtime, achieving superior performance over static approaches and surpassing existing bug-finding methods, including the SOTA BMC implementation ABC-BMC [4].
4) We provide an extensive empirical study of the influence of assumption ordering on the UC, thereby empirically

Jianwen Li and Geguang Pu are the corresponding authors. This work is supported by NSFC Grant #U21B2015 and #62372178.

[1]CAR with the improvements reported here has recently won the 3rd place in the bit-level model-checking competition [12].

demonstrating the significance of literal ordering in CAR-based model checking.

Owing to space considerations, we cannot include here detailed preliminaries about the PDR and CAR model-checking algorithms, and have to assume that the reader is familiar with them. These, as well as additional analysis of the experiments, appear in the long version of this article [9].

II. Literal Reordering: prior work and insights

We start with a high-level description of the CAR algorithm, the foundation of our work. CAR maintains two sequences of state sets: an over-approximating sequence (O) and an under-approximating sequence (U). The sequences start with $O_0 = \neg P$ (the negation of the safety property) and $U_0 = I$ (the initial states). CAR updates these sequences by checking the satisfiability of transitions for each state in U. If a transition is satisfiable, then the target state is added to U to widen it ; Otherwise, the UC is added to O, which narrows it. The process terminates when a state in U intersects with $\neg P$ (returning 'Unsafe') or the union of O subsumes the next frame (returning 'Safe').

As mentioned in the introduction, modern CDCL-based SAT solvers take as input, in addition to the formula ϕ, a vector of literals $\mathcal{A}$, called the assumptions, and checks whether $\bigwedge_{l\in\mathcal{A}} l \wedge \phi$ is satisfiable. The SAT solver chooses the assumption literals to be the first decisions in the order that they are given. As usual, after each such decision, it invokes BCP. Suppose there is already a conflict in the first $|A|$ ($A \subseteq \mathcal{A}$) decision levels (recall that this can happen after learning and backtracking to those levels). In that case, the search is terminated – the formula is declared unsatisfiable under A (with further analysis the solver can detect a subset of A that is responsible for the conflict, but this is immaterial to the discussion). This implies that assumption literals after $|A|$ in the predefined order cannot be part of the generated UC. As a result, *prior assumption literals have a higher probability of appearing in the UC*. That is why literal ordering matters.

The *Intersection* strategy and *locality* of cores: The *Intersection* strategy [10] places the intersection with the last UC in the beginning of the assumptions sequence. The literals from this UC are placed first in the order, which makes them more likely to appear in the new core, hence make consecutive cores *similar*. This is what we call 'the *locality* of the cores'.

The term *locality* is used, among other places, in describing decision heuristics in SAT solving. All CDCL solvers use decision heuristics that prioritize variables that participated in recent conflicts, hence they focus the search. Although this is not directly related to the current paper, our hypothesis is that this decision strategy is effective because it generates proofs faster: similar clauses are necessary for constructing a *resolution proof* (for satisfiable cases, learning has little effect to begin with [16]). And if there is a small proof, it is better to focus the search and hopefully find it rather than generating unrelated clauses.

Our argument is that finding cores in CAR that are similar should have a similar effect: it makes proofs involving the

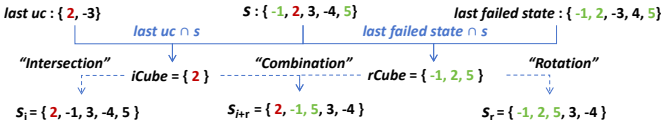


Fig. 1. An illustrating example of the reordering process, where s is the incoming state, 'last uc' and 'last failed state' are both from the last UNSAT query. s_i, s_r and s_{i+r} each represents the reordered state $\hat{s}$ using only *Intersection*, only *Rotation* and their combination, respectively.

O frames easier and hence faster. In other words, every time that we check whether a state can reach an O frame, if that frame contains apriori many of the clauses that are necessary for the proof that the state is not reachable, the proof will converge faster. We tested this hypothesis empirically based on benchmarks from the HWMCC 2015 and 2017 [11] competitions[2], and the results appear in the following table. It confirms that locality improves runtime for UNSAT cases and speeds up overall state reachability proofs.

Strategy	Natural	Intersection
Average time of UNSAT calls(s)	0.0132	0.0105
Average time of finding proofs(s)	0.9541	0.6287

The first row shows the effect of locality on the average run time of UNSAT cases, and the second is the average overall time for proving that a state cannot reach an O frame. Both show an acceleration.

The *Rotation* strategy: The *Rotation* technique prioritizes literals common in recent failed states. The rationale is that it helps the solver avoid getting trapped for long in various parts of the search space, by generating UCs that are mostly based on those common literals. It maintains a vector of literals $common$ with a fixed size equal to the size of a state.

Our results, summarized in the following table, show a reduction in the number of SAT calls with this strategy. The basis of the evaluation is the same as before.

Strategy	Natural	Rotation
Average #SAT calls to find proofs	207.13	190.25
Average time to find proofs (s)	0.9541	0.7277

Combining *Intersection* and *Rotation*: When it comes to combining the two strategies, it should be noted that the latest UC selected in *Intersection* is derived from the last failed state. This observation leads to the conclusion that the cube generated via *Intersection*, called $iCube$, is a subset of the cube generated via *Rotation*, which is called $rCube$. As shown in Fig.1, to integrate the two algorithms is merely to position the literals produced by *Intersection* ahead of those generated by *Rotation* while eliminating duplicate literals in the latter.

The results in the following table show that indeed the combination finds proofs faster on average:

Strategy	Natural	Combination (I+R)
Average time of UNSAT calls(s)	0.0132	0.0137
Average #(SAT Query) to find proof	207.13	173.57
Average time of finding proofs(s)	0.9541	0.5990

III. Literal Ordering Strategies: new approaches

As discussed in the previous section, *Intersection* and *Rotation*, either separately or combined, determine the position

[2]The experiment setup in this section is the same as that in Sec. IV.

of only a limited portion of the whole literal set, whereas the position of other literals is determined by what we called the 'natural' order, i.e., arbitrary.

In this section we will show a way to increase the portion of literals that their position is determined, utilizing more historical information on the cores, and consequently improving the overall runtime.

A. Literal reordering with CoreLocality

Stemming from the intuition that incorporating recent UCs beyond the most recent one could help by improving locality, we propose a new literal ordering strategy *CoreLocality*, which is outlined in Alg. 1. By expanding the scope of considered UCs, intersecting with each and organizing the results chronologically (with the intersection with newer UCs placed earlier), *CoreLocality* facilitates sorting a greater number of literals, thereby refining the guidance of the search.

As shown in the algorithm, in addition to the state s and frame level l, a new parameter $iLimit$ is introduced to denote the limit on the amount of UCs to utilize. The *for* block at Line 2-8 computes the intersection according to the corresponding UC, and pushes them into $\hat{s}$ in order. For the *if* block at Line 11-13, it is similar to *Rotation*.

Algorithm 1: Reordering algorithm: *CoreLocality*

Input: A state s, frame level l, configuration $iLimit$
Output: $\hat{s}$: The reordered s

1 $\hat{s} := \emptyset$
2 **for** $k : 0 \rightarrow iLimit$ **do**
3 Let $ref_k = getTheLast\_kth\_UC(l)$
4 **if** $ref_k \neq \emptyset$ **then**
5 **for** *each* $lit \in ref_k$ **do**
6 **if** $lit \in s \wedge lit \notin \hat{s}$ **then**
7 $\hat{s}.pushBack(lit)$
8 ▷ Literals added here form the $iCubes$
9 $cVec := getCommonVector(l)$
10 **for** *each* $lit \in cVec$ **do**
11 **if** $lit \in s \wedge lit \notin \hat{s}$ **then**
12 $\hat{s}.pushBack(lit)$
13 ▷ Literals added here form the $rCube$
14 **for** *each* $l \in s \wedge l \notin \hat{s}$ **do**
15 $\hat{s}.pushBack(l)$
16 **return** $\hat{s}$

Example III.1. *Fig. 2 illustrates a computational process for the* CoreLocality *strategy with several different $iLimit$ values. The left dashed box shows the last 3 UCs in chronological order (1st being the most recent one), along with the last failed state, and the current state s. Next, $iCubes$ and $rCube$ are computed based on this data, similar to the calculation in Fig. 1, as shown in the middle dashed box. Finally, in the right dashed box, s is reordered by $iCubes$ and $rCube$ based on different choices of $iLimit$. As is shown, by incorporating the 2nd UC, the literal '-4' is pushed forward.* □

The distinction between *CoreLocality* and *Intersection* is demonstrated in Fig. 3. It prioritizes literals that would otherwise be relegated to the rear of $rCube$, or even after $rCube$.

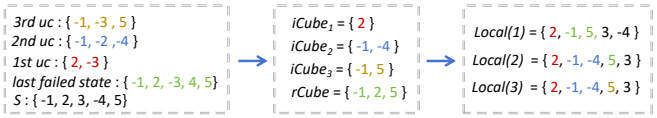


Fig. 2. An example of the *CoreLocality* strategy. UCs are in chronological order, where '1st UC' is from the most recent UNSAT query. $Local(k)$ denotes a reordered state, utilizing k UCs, i.e., $iLimit = k$.

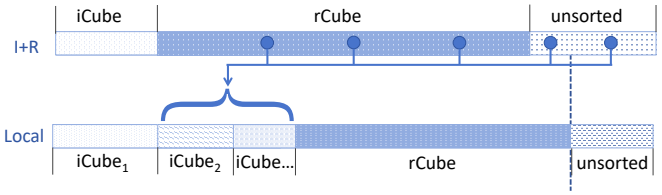


Fig. 3. Comparing *CoreLocality* and a combination of *Intersection* and *Rotation*. Some literals (blue dots) from $rCube$ and $unsorted$ are prioritized.

Tuning *CoreLocality*: In the *CoreLocality* strategy, the parameter $iLimit$, which denotes the maximum number of utilised UCs, serves as a metric of the 'local' scope, defining the range within which a UC is considered 'recent'. In other words, given that the relevance of a UC to the current query diminishes as it becomes more distant, setting a limit excludes prior outdated UCs from current consideration. While increasing the value of $iLimit$ allows for the inclusion of additional information, it also diminishes the impact of *Rotation* due to the precedence of $iCubes$ over the $rCube$. Furthermore, while it is feasible to set the $iLimit$ large enough to order all the literals, this approach is observed to be highly inefficient because getting one more literal reordered (a literal that appears in a subsequent UC, but not in any previous one) often necessitates thousands or even tens of thousands of UCs.

The optimal value of $iLimit$ depends, of course, on the specific problem context and constitutes a trade-off between literal coverage and the computational cost to achieve it. Indeed, the results in the table below (the left value of the 'Time' column) show that the speed to find proofs of *CoreLocality* depends on $iLimit$, but, as expected, it is not monotonic. They also show that with these low values of $iLimit$ we are able to find proofs faster than the previous methods. In practice the best $iLimit$ value can be found based on experiments, but there is also an option to change it during run time, as we will explain later.

Strategy	Time (s)	Strategy	Time (s)
Natural	0.94 / 0.95	Local(3)	0.64 / 0.55
Combination(I+R)	0.77 / 0.59	Local(4)	0.57 / 0.48
Local(2)	0.61 / 0.58	Local(5)	0.64 / 0.50

B. Moving forward the conflict literals

Not all literals in a UC are equal. Specifically, the last literal added to the core, by definition, was *necessary* for that proof (in other words, it is part of a *minimal* core). We call it the *conflict* literal. During the process of getting UCs, we give such conflict literals a higher priority by placing them in the front of the core. For example, suppose that $iCube = (1, 2, 3)$ and literal 3 is the conflict literal of this clause. Then we reorder $iCube$ to $(3, 1, 2)$, before we proceed with building the assumption literal order as described earlier (Alg. 1). As shown in the table above (the right value of the 'Time' column),

this small optimization accelerates, on average, the process of finding proofs. From here on, when we say *CoreLocality*, we mean *CoreLocality* together with this optimization.

C. Hybrid-CAR: Combining different orderings

Empirically, the best configuration of *CoreLocality* varies according to the specific problem and is hard to predict. It is often observed that a model checking problem that can be solved easily with one literal ordering strategy will time-out with another. This encouraged us to research a dynamic strategy, by which the configuration is periodically switched.

We coupled this direction with a new restart mechanism for CAR. Our experiments show that the U-sequence in CAR frequently expands quickly, resulting in an increasingly longer time to extend a new O frame. Perhaps, then, resetting the U frames to I and the O sequence to $O[0]$, and progressing with a different literal ordering can converge faster. Based on this hypothesis we implemented Hybrid-CAR, a version of CAR in which a timer is kept in the SAT solver. Once it exceeds a predefined value, it triggers a restart procedure: (a) clear all the U states other than the states in the lowest level, and (b) increase the value of $iLimit$ by one. Finally, to preserve completeness, we give the option to increase the time limit, each time restart is called.

IV. Experiments

Our experiments focused on bug-finding only, and accordingly we implemented our suggested algorithms on top of SimpleCAR [14], which is an implementation of the CAR algorithm, in its best version for bug-finding [10]. We compared ourselves to the best public BMC implementation (the one in ABC-BMC [4]), and the best combination of CAR and BMC in [19]. Our evaluation was based on 438 benchmarks[3] in the Aiger [6] format from the single safety property track of the 2015 and 2017 [11] Hardware Model Checking Competition (HWMCC)[4], which is consistent with the benchmark set of [19]. All the counterexamples found were successfully verified with the third-party tool *aigsim* that comes with the Aiger package [2]. All the artifacts are available in Github [8].

We ran the experiments on a cluster of Linux servers, each equipped with an Intel Xeon Gold 6132 14-core processor at 2.6 GHz and 96 GB RAM. The version of the operating system is Red Hat 4.8.5-16. For each running instance, the memory was limited to 8 GB and the time was limited to 1 hour.

Q1. How does *CoreLocality* perform when compared to the present best reordering strategy in CAR, i.e., *Intersection + Rotation*? The previous literal-ordering strategy for CAR, namely the combination of *Intersection* and *Rotation* as published in [10], is very close to *CoreLocality* when $iLimit$ is set to one ('Local-1'), except that *CoreLocality* introduces a reordering inside the UCs (see Sec. III-A). Recall that in Sec. II we presented empirical evidence that confirms independently of [10] that these two strategies improve the empirical results.

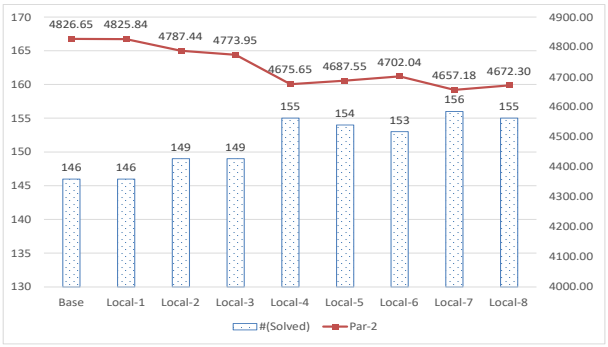
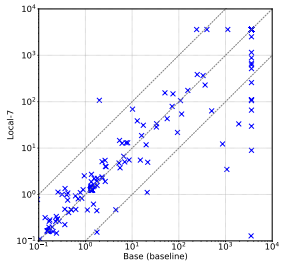


Fig. 4. Results on different reordering strategies, in terms of the total solved instances and PAR-2 score. 'Base' refers to the combination of *Intersection* and *Rotation*, 'Local-i' ($1 \leq i \leq 8$) represents the *CoreLocality* strategy with $iLimit = i$.

As is shown in Figure 4, the performance of *CoreLocality* with $1 < iLimit \leq 8$ outperforms that of the base strategy on both the number and time of solved cases, peaking at $iLimit = 7$. Increasing the number of solved cases, even by a few instances, is important in light of the decades of research and development of model checkers.

The Par-2 score calculates average time consumption while doubling the time for instances that timed out. *CoreLocality* with $iLimit > 1$ consistently consumes less time than the Base strategy. It is also apparent that there is a correlation between the number of solved instances and the total time consumption. A detailed pairwise comparison between the peak and Base is shown below on the right.



Overall, *CoreLocality* outperforms the Base strategy on both the number and time of solved cases. Notably, the performance of *CoreLocality* with different configurations is not correlated to the value of $iLimit$. This is consistent with our discussion in Section III-A, that increasing the limit does not have a monotonic effect.

Q2: How does Hybrid-CAR perform when compared to the state-of-the-art bug-finding algorithms? We compared Hybrid-CAR to the original SimpleCAR, BAC-1500 (the best solution shown in [19]) and ABC-BMC on bug finding[5].

It turns out that Hybrid-CAR performs better than the competitors. With a 1-hour timeout, the best version of Hybrid-CAR, in which the restart is invoked every 6 minutes, solves 166 cases in total, which is seven more than that solved by ABC-BMC, and 20 more than that solved by the original SimpleCAR. With a 6-hour timeout (shown in the long version [9] of this article), this version of Hybrid-CAR solves 169 cases, which is 20 more than the original SimpleCAR, and outperforms the competition. Furthermore, Hybrid-CAR

[3]Results of these benchmarks are either unknown or known to be unsafe.

[4]These are the last two years of HWMCC using the AIGER format. Since 2019, the official format switched to a word-level format BTOR [5].

[5]Other tools such as ABC-PDR, nuXmv-IC3, AVY [17] do not perform well on unsafe cases [18] and are hence not included here.

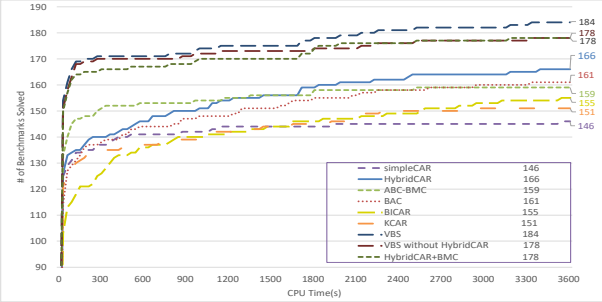


Fig. 5. Comparison of run-time performance among different model checkers. VBS represents the virtual best, i.e., parallel running all and taking the best.

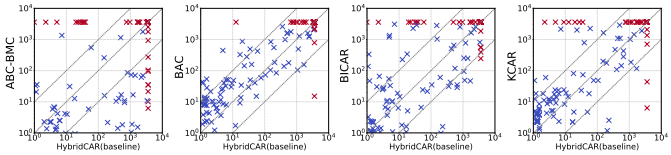


Fig. 6. Pairwise Comparison of Hybrid-CAR and competitors. Timeout instances in either are marked in red.

solves more instances in just one hour (166) than that solved by ABC-BMC in 6 hours (165).

Figure 5 includes the comparison among the best version of Hybrid-CAR and ABC-BMC, as well as the combinations of BMC and CAR presented in [19], i.e., BAC, BICAR and KCAR. Hybrid-CAR can solve more cases (166) than the others, and six *unique* cases, i.e., benchmarks that cannot be solved by all the other methods.

The following table shows the uniquely solved instances of each technique (i.e. that no other tool can solve), and, in parenthesis, in comparison to ABC-BMC and BAC, e.g., Hybrid-CAR solves 19 and 11 cases that cannot be solved by these two tools, respectively. Moreover, we note that a portfolio of only Hybrid-CAR and ABC-BMC can solve 178 instances, almost reaching the virtual best results (184) that a portfolio of all these algorithms can solve. A detailed pairwise comparison is shown in Fig. 6.

simpleCAR	0 (13 / 4)	ABC-BMC	5 (0 / **15**)
Hybrid-CAR	6 (**19** / 11)	BICAR	0 (5 / 10)
BAC	4 (17 / 0)	KCAR	1 (10 / 6)

V. Conclusion

In this paper, we revisited the assumption literal ordering strategies presented in [10]. We hypothesized that *Intersection* works because of what we call *core locality*, which means that similar cores help the SAT solver find proofs faster. Our empirical data, as we showed, supports this claim. Both *Intersection* and *Rotation* determine only a part of the literal order, hence the order of most of the assumptions is left arbitrary. Our improved strategy, CoreLocality (Sec. III-A), generalizes *Intersection* and orders a larger part of the assumptions sequence, while improving the core locality. Together with prioritizing *conflict literals* (Sec. III-B) they shorten rather significantly the time it takes the SAT solver to find proofs. We also presented a hybrid approach called Hybrid-CAR (Sec. III-C), which switches between different configurations of CoreLocality during run time, while resetting the U sequences. Our results show that these strategies perform better on average than the reordering strategies of [10] and also better than the various integrations of CAR with BMC [19]. In particular, Hybrid-CAR is able to outperform all bug-finding model-checking algorithms off-the-shelf.

References

[1] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC)*, pages 317–320, 1999.
[2] Armin Biere. AIGER Format. http://fmv.jku.at/aiger/FORMAT.
[3] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
[4] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[5] Robert Brummayer, Armin Biere, and Florian Lonsing. Btor: bit-precise modelling of word-level problems for model checking. In *Proceedings of the joint workshops of the 6th international workshop on satisfiability modulo theories and 1st international workshop on bit-precise reasoning*, pages 33–38, 2008.
[6] Robert Brummayer, Alessandro Cimatti, Koen Claessen, Niklas Een, Marc Herbstritt, Hyondeuk Kim, Toni Jussila, Ken McMillan, Alan Mishchenko, Fabio Somenzi, et al. The AIGER and-inverter graph (aig) format version 20070427. In *The AIGER And-Inverter Graph (AIG) Format Version 20070427*, 2007.
[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
[8] Artifacts. https://github.com/AnonymousAccO-O-O/HybridCAR.
[9] Yibo Dong, Yu Chen, Jianwen Li, Geguang Pu, and Ofer Strichman. Revisiting assumptions ordering in CAR-based model checking (long version). https://doi.org/10.48550/arXiv.2411.00026.
[10] Rohit Dureja, Jianwen Li, Geguang Pu, Moshe Y. Vardi, and Kristin Y. Rozier. Intersection and rotation of assumption literals boosts bug-finding. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 180–192, Cham, 2020. Springer International Publishing.
[11] HWMCC . http://fmv.jku.at/hwmcc??/ (replace ?? with the year).
[12] HWMCC 2024. https://hwmcc.github.io/2024/.
[13] IC3Ref. https://github.com/arbrad/IC3ref.
[14] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y. Vardi. SimpleCAR: An efficient bug-finding tool based on approximate reachability. In *Proc. Int. Conf. Computer Aided Verification (CAV)*, pages 37–44, 2018.
[15] Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, and Moshe Y. Vardi. Safety model checking with complementary approximations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, pages 95–100. IEEE Press, 2017.
[16] Chanseok Oh. Between SAT and UNSAT: The fundamental difference in CDCL SAT. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 307–323, Cham, 2015. Springer International Publishing.
[17] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 367–385, Cham, 2019. Springer International Publishing.
[18] Yechuan Xia, Anna Becchi, Alessandro Cimatti, Alberto Griggio, Jianwen Li, and Geguang Pu. Searching for i-good lemmas to accelerate safety model checking. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 288–308, Cham, 2023. Springer Nature Switzerland.
[19] X. Zhang, S. Xiao, J. Li, G. Pu, and O. Strichman. Combining bmc and complementary approximate reachability to accelerate bug-finding. In *Proc. 41st IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 2022.