SimpleCAR: An Efficient Bug-Finding Tool Based On Approximate Reachability

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

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

Abstract. We present a new safety hardware model checker SimpleCAR that serves as a reference implementation for evaluating Complementary Approximate Reachability (CAR), a new SAT-based model checking framework inspired by classical reachability analysis. The tool gives a "bottom-line" performance measure for comparing future extensions to the framework. We demonstrate the performance of SimpleCAR on challenging benchmarks from the Hardware Model Checking Competition. Our experiments indicate that SimpleCAR is particularly suited for unsafety checking, or *bug-finding*; it is able to solve 7 unsafe instances within 1 hour that are not solvable by any other state-of-the-art techniques, including BMC and IC3/PDR, within 8 hours. We also identify a bug (reports safe instead of unsafe) and 48 counterexample generation errors in the tools compared in our analysis.

1 Introduction

Model checking techniques are widely used in proving design correctness, and have received unprecedented attention in the hardware design community [9, 16]. Given a system model M and a property P, model checking proves whether or not P holds for M. A model checking algorithm exhaustively checks all behaviors of M, and returns a counterexample as evidence if any behavior violates the property P. The counterexample gives the execution of the system that leads to property failure, i.e., a *bug*. Particularly, if P is a safety property, model checking reduces to reachability analysis, and the provided counterexample has a finite length. Popular safety checking techniques include Bounded Model Checking (BMC) [10], Interpolation Model Checking (IMC) [21], and IC3/PDR [12, 14]. It is well known that there is no "universal" algorithm in model checking; different algorithms perform differently on different problem instances [7]. BMC outperforms IMC on checking unsafe instances, while IC3/PDR can solve instances that BMC cannot and vice-versa. [19]. Therefore, BMC and IC3/PDR are the most popular algorithms in the portfolio for unsafety checking, or *bug-finding*.

Complementary Approximate Reachability (CAR) [19] is a SAT-based model checking framework for reachability analysis. Contrary to reachability analysis via IC3/PDR, CAR maintains two sequences of over- and under- approximate reachable state-sets. The over-approximate sequence is used for safety checking, and the under-approximate sequence for unsafety checking. CAR does not

require the over-approximate sequence to be monotone, unlike IC3/PDR. Both forward (Forward-CAR) and backward (Backward-CAR) reachability analysis are permissible in the CAR framework. Preliminary results show that Forward-CAR complements IC3/PDR on safe instances [19].

We present, SimpleCAR, a tool specifically developed for evaluating and extending the CAR framework. The new tool is a complete rewrite of CARChecker [19] with several improvements and added capabilities. SimpleCAR has a lighter and cleaner implementation than CARChecker. Several heuristics that aid Forward-CAR to complement IC3/PDR are integrated in CARChecker. Although useful, these heuristics make it difficult to understand and extend the core functionalities of CAR. Like IC3/PDR, the performance of CAR varies significantly by using heuristics [17]. Therefore, it is necessary to provide a basic implementation of CAR (without code-bloating heuristics) that serves as a "bottom-line" performance measure for all extensions in the future. To that end, SimpleCAR differs from CARChecker in the following aspects:

- Eliminates all heuristics integrated in CARChecker except a configuration option to enable a IC3/PDR-like clause "propagation" heuristic.
- Uses UNSAT cores from the SAT solver directly instead of the expensive minimal UNSAT core (MUC) computation in CARChecker.
- Poses incremental queries to the SAT solver using assumptions;
- While CARChecker contributes to safety checking [19], SimpleCAR shows a clear advantage on unsafety checking.

We apply SimpleCAR to 748 benchmarks from the Hardware Model Checking Competition (HWMCC) 2015 [2] and 2017 [3], and compare its performance to reachability analysis algorithms (BMC, IMC, 4×IC3/PDR, Avy [22], Quip [18]) in state-of-the-art model checking tools (ABC, nuXmv, IIMC, IC3Ref). Our extensive experiments reveal that Backward-CAR is particularly suited for unsafety checking: it can solve 8 instances within a 1-hour time limit, and 7 instances within a 8-hour time limit not solvable by BMC and IC3/PDR. We conclude that, along with BMC and IC3/PDR, CAR is an important candidate in the portfolio of unsafety checking algorithms, and SimpleCAR provides an easy and efficient way to evaluate, experiment with, and add enhancements to the CAR framework. We identify 1 major bug and 48 errors in counterexample generation in our evaluated tool set; all have been reported to the tool developers.

2 Algorithms and Implementation

We present a very high-level overview of the CAR framework (refer [19] for details). CAR is a SAT-based framework for reachability analysis. It maintains two over- and under- approximate reachable state sequences for safety and unsafety checking, respectively. CAR can be symmetrically implemented either in the forward (Forward-CAR) or backward (Backward-CAR) mode. In the forward mode, the F-sequence $(F_0, F_1, \ldots, F_i)$ is the over-approximated sequence, while the B-sequence $(B_0, B_1, \ldots, B_i)$ is under-approximated. The roles of the F- and

B- sequence are reversed in the backward mode. We focus here on the backward mode of CAR, or Backward-CAR (refer [19] for Forward-CAR)

2.1 High-level Description of Backward-CAR

A frame F_i in the F-sequence denotes the set of states that are reachable from the initial states (I) in i steps. Similarly, a frame B_i in the B-sequence denotes the set of states that can reach the bad states ($\neg P$) in i steps. Let $R(F_i)$ represent the set of successor states of F_i, and $R^{-1}(B_i)$ represent the set of predecessor states of B_i. Table 1 shows the constraints on the sequences and their usage in Backward-CAR for safety and unsafety checking.

Table 1: Sequences in Backward-CAR.

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

Let $S(F) = \bigcup F_i$ and $S(B) = \bigcup B_i$. Alg. 1 gives a description of Backward-CAR. The B-sequence is extended exactly once in every iteration of the loop in lines 2–8, but the F-sequence may be extended multiple times in each loop iteration in lines 3–5. As a result, CAR normally returns counterexamples with longer depth compared to the length of the B-sequence. Due to this inherent feature of the framework, CAR is able to complement BMC and IC3/PDR on unsafety checking.

Alg. 1 High-level description of Backward CAR

```
1: F_0 = I, B_0 = ¬P, k = 0;
2: while true do
3:     while S(B) ∧ R(S(F)) ≠ ∅ do
4:         update F- and B- sequences.
5:         if ∃i · F_i ∩ ¬P ≠ ∅ then return unsafe;
6:     perform propagation on B-sequence (optional);
7:     if ∃i · B_{i+1} ⊆ ⋃_{0≤j≤i} B_j then return safe;
8:     k = k + 1 and B_k = ¬P;
```

2.2 Tool Implementation

SimpleCAR is publicly available [5, 6] under the GNU GPLv3 license. The tool implementation is as follows:

- **Language:** C++11 compilable under `gcc` 4.4.7 or above.
- **Input:** Hardware circuit models expressed as and-inverter graphs in the *aiger* 1.9 format [11] containing a single safety property.
- **Output:** "1" (unsafe) to report the system violates the property, or "0" (safe) to confirm that the system satisfies the property. A counterexample in the *aiger* format is generated if run with the `-e` configuration flag.
- **Algorithms:** Forward-CAR and Backward-CAR with and without the propagation heuristic (enabled using the `-p` configuration flag).
- **External Tools:** Glucose 3.0 [8] (based on MiniSAT [15]) is used as the underlying SAT solver. *Aiger* tools [1] are used for parsing the input aiger files to extract the model and property information, and error checking.

– **Differences with CARChecker [19]:** The Minimal Unsat Core (MUC) and Partial Assignment (PA) techniques are not utilized in SimpleCAR, which allows the implementation to harness the power of incremental SAT solving.

3 Experimental Analysis

3.1 Strategies

Tools. We consider five model checking tools in our evaluation: ABC 1.01 [13], IIMC 2.0[4], Simplic3 [17] (IC3 algorithms used by nuXmv for finite-state systems[5]), IC3Ref [4], CARChecker [19], and SimpleCAR. For ABC, we evaluate BMC (`bmc2`), IMC (`int`), and PDR (`pdr`). There are three different versions of BMC in ABC: `bmc`, `bmc2`, and `bmc3`. We choose `bmc2` based on our preliminary analysis since it outperforms other versions. Simplic3 proposes different configuration options for IC3. We use the three *best candidate* configurations for IC3 reported in [17], and the Avy algorithm [22] in Simplic3. We consider CARChecker as the original implementation of the CAR framework and use it as a reference implementation for SimpleCAR. A summary of the tools and their arguments used for experiments is shown in Table 2. Overall, we consider four categories of algorithms implemented in the tools: BMC, IMC, IC3/PDR, and CAR.

Benchmarks. We evaluate all tools against 748 benchmarks in the *aiger* format [11] from the SINGLE safety property track of the HWMCC in 2015 and 2017.

Error Checking. We check correctness of results from the tools in two ways:

1. We use the `aigsim` [1] tool to check whether the counterexample generated for unsafe instances is a real counterexample by simulation.
2. For inconsistent results (safe and unsafe for the same benchmark by at least two different tools) we attempt to simulate the unsafe counterexample, and if successful, report an error for the tool that returns safe (surprisingly, we do not encounter cases when the simulation check fails).

Platform. Experiments were performed on Rice University's DavinCI cluster, which comprises of 192 nodes running at 2.83GHz, 48GB of memory and running RedHat 6.0. We set the memory limit to 8GB with a wall-time limit of an hour. Each model checking run has exclusive access to a node. A time penalty of one hour is set for benchmarks that cannot be solved within the time/memory limits.

3.2 Results

Error Report. We identify one bug in `simplic3-best3`: reports safe instead of unsafe, and 48 errors with respect to counterexample generation in `iimc-quip` algorithm (26) and all algorithms in the Simplic3 tool (22). At the time of writing, the bug report sent to the developers of Simplic3 has been confirmed. In our analysis, we assume the results from these tools to be correct.

[4] We use version 2.0 available at https://ryanmb.bitbucket.io/truss/ – similar to the version available at https://github.com/mgudemann/iimc with addition of Quip [18].

[5] Personal communication with Alberto Griggio.

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

	Tool	Algorithm	Configuration Flags
IC3/PDR	ABC	BMC (`abc-bmc`)	`-c 'bmc2'`
		IMC (`abc-int`)	`-c 'int'`
		PDR (`abc-pdr`)	`-c 'pdr'`
	IIMC	IC3 (`iimc-ic3`)	`-t ic3 --ic3_stats --print_cex --cex_aiger`
		Quip [18] (`iimc-quip`)	`-t quip --quip_stats --print_cex --cex_aiger`
	IC3Ref	IC3 (`ic3-ref`)	`-b`
	Simplic3	IC3 (`simplic3-best1`)	`-s minisat -m 1 -u 4 -I 0 -O 1 -c 1 -p 1 -d 2 -G 1 -P 1 -A 100`
		IC3 (`simplic3-best2`)	`-s minisat -m 1 -u 4 -I 1 -D 0 -g 1 -X 0 -O 1 -c 0 -p 1 -d 2 -G 1 -P 1 -A 100`
		IC3 (`simplic3-best3`)	`-s minisat -m 1 -u 4 -I 0 -O 1 -c 0 -p 1 -d 2 -G 1 -P 1 -A 100 -a aic3`
		Avy [22] (`simplic3-avy`)	`-a avy`
CAR	CARChecker	Forward CAR* (`carchk-f`)	`-f`
		Backward CAR* (`carchk-b`)	`-b`
	SimpleCAR	Forward CAR† (`simpcar-f`)	`-f -e`
		Backward CAR† (`simpcar-b`)	`-b -e`
		Forward CAR‡ (`simpcar-fp`)	`-f -p -e`
		Backward CAR‡ (`simpcar-bp`)	`-b -p -e`

* with heuristics for *minimal unsat core* (MUC) [20], partial assignment [23], and propagation.
† no heuristics
‡ with heuristic for PDR-like clause propagation

Coarse Analysis. We focus our analysis to unsafety checking. Fig. 1 shows the total number of unsafe benchmarks solved by each category (assuming portfolio-run of all algorithms in a category). CAR **complements** BMC **and** IC3/PDR **by solving 128 benchmarks of which 8 are not solved by any other category.** Although CAR solves the least amount of total benchmarks, the count of the uniquely solved benchmarks is comparable to other categories. When the wall-time limit (memory limit does not change) is increased to 8 hours, BMC and IC3/PDR can only solve one of the 8 uniquely solved benchmarks by CAR. The analysis supports our claim that CAR complements BMC/IC3/PDR on unsafety checking.

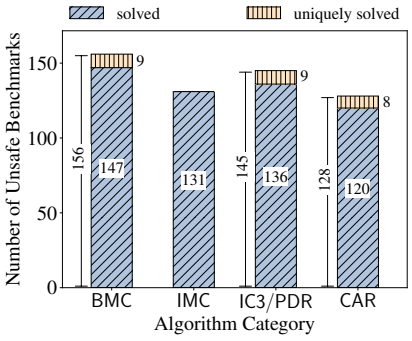
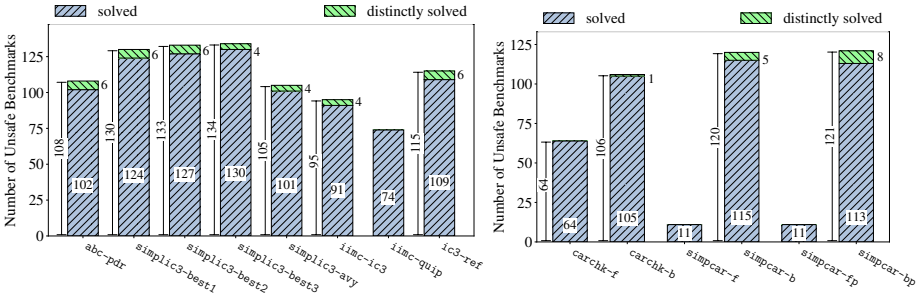


Fig. 1: Number of benchmarks solved by each algorithm category (run as a portfolio). Uniquely solved benchmarks are not solved by any other category.

Granular Analysis. Fig. 2 shows how each algorithm in the IC3/PDR (Fig. 2a) and CAR (Fig. 2b) categories performs on the benchmarks. `simpcar-bp` **distinctly solves all 8 benchmarks uniquely solved by the** CAR **category (Fig. 1), while no single** IC3/PDR **algorithm distinctly solves all uniquely solved benchmarks in the** IC3/PDR **category.** In fact, a portfo-



(a) Algorithms in IC3/PDR category

(b) Algorithms in CAR category

Fig. 2: Number of benchmarks solved by every algorithm in a category. Distinctly solved benchmarks by an algorithm are not solved by any algorithm in other categories. The set union of distinctly solved benchmarks for all algorithms in a category equals the count of uniquely solved for that category in Fig. 1.

lio including at least `abc-pdr`, `simplic3-best1`, and `simplic3-best2` solves all 8 instances uniquely solved by the IC3/PDR category. It is important to note that SimpleCAR is a very basic implementation of the CAR framework compared to the highly optimized implementations of IC3/PDR in other tools. Even then `simpcar-b` **outperforms four** IC3/PDR **implementations.** Our results show that Backward-CAR is a favorable algorithm for unsafety checking.

Analysis Conclusions. Backward-CAR presents a more promising research direction than Forward-CAR for unsafety checking. We conjecture that the performance of Forward- and Backward- CAR varies with the structure of the *a*iger model. Heuristics and performance-gain present a trade-off. `simpcar-bp` has a better performance compared to the heuristic-heavy `carchk-b`. On the other hand, `simpcar-bp` solves the most unsafe benchmarks in the CAR category, however, adding the "propagation" heuristic effects its performance: there are several benchmarks solved by `simpcar-b` but not by `simpcar-bp`.

4 Summary

We present SimpleCAR, a safety model checker based on the CAR framework for reachability analysis. Our tool is a lightweight and extensible implementation of CAR with comparable performance to other state-of-the-art tool implementations of highly-optimized unsafety checking algorithms, and complements existing algorithm portfolios. Our empirical evaluation reveals that adding heuristics does not always improve performance. We conclude that Backward-CAR is a more promising research direction than Forward-CAR for unsafety checking, and our tool serves as the "bottom-line" for all future extensions to the CAR framework.

Acknowledgments. This work is supported by NSF CAREER Award CNS-1552934, NASA ECF NNX16AR57G, NSF CCF-1319459, and NSFC 61572197 and 61632005 grants. Geguang Pu is also partially supported by MOST NKTSP Project 2015BAG19B02 and STCSM Project No.16DZ1100600.

References

1. AIGER Tools. http://fmv.jku.at/aiger/aiger-1.9.9.tar.gz
2. HWMCC 2015. http://fmv.jku.at/hwmcc15/
3. HWMCC 2017. http://fmv.jku.at/hwmcc17/
4. IC3Ref. https://github.com/arbrad/IC3ref
5. SimpleCAR Source. https://github.com/lijwen2748/simplecar/releases/tag/v0.1
6. SimpleCAR Website. http://temporallogic.org/research/CAV18/
7. Amla, N., Du, X., Kuehlmann, A., Kurshan, R.P., McMillan, K.L.: An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In: CHARME (2005)
8. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: IJCAI (2009)
9. Bernardini, A., Ecker, W., Schlichtmann, U.: Where Formal Verification Can Help in Functional Safety Analysis. In: ICCAD (2016)
10. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic Model Checking Using SAT Procedures Instead of BDDs (1999)
11. Biere, A.: AIGER Format. http://fmv.jku.at/aiger/FORMAT
12. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI (2011)
13. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: CAV (2010)
14. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: FMCAD (2011)
15. Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT (2004)
16. Golnari, A., Vizel, Y., Malik, S.: Error-tolerant processors: Formal specification and verification. In: ICCAD (2015)
17. Griggio, A., Roveri, M.: Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. IEEE Trans. Comput-Aided Design Integr. Circuits Syst. 35(6), 1026–1039 (Jun 2016)
18. Ivrii, A., Gurfinkel, A.: Pushing to the Top. In: FMCAD (2015)
19. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety Model Checking with Complementary Approximations. In: ICCAD (2017)
20. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K., Simon, L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2011. Lecture Notes in Computer Science, vol. 6695, pp. 159–173. Springer Berlin Heidelberg (2011)
21. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: CAV (2003)
22. Vizel, Y., Gurfinkel, A.: Interpolating Property Directed Reachability. In: CAV (2014)
23. Yu, Y., Subramanyan, P., Tsiskaridze, N., Malik, S.: All-SAT Using Minimal Blocking Clauses. In: VLSID (2014)