Empirical Evaluation of IC3-Based Model Checking Techniques on Verilog RTL Designs

Aman Goel
University of Michigan, Ann Arbor
amangoel@umich.edu

Karem Sakallah
University of Michigan, Ann Arbor
karem@umich.edu

***Abstract*—IC3-based algorithms have emerged as effective scalable approaches for hardware model checking. In this paper we evaluate six implementations of IC3-based model checkers on a diverse set of publicly-available and proprietary industrial Verilog RTL designs. Four of the six verifiers we examined operate at the bit level and two employ abstraction to take advantage of word-level RTL semantics. Overall, the word-level verifier employing data abstraction outperformed the others, especially on the large industrial designs. The analysis helped us identify several key insights on the techniques underlying these tools, their strengths and weaknesses, differences and commonalities, and opportunities for improvement.**

I. Introduction

IC3 [1] (also referred to as PDR [2]) has emerged as an effective model checking (MC) algorithm that uses incremental SAT solving to perform property-directed approximate reachability analysis. Model checkers using IC3-based techniques at the bit level have shown exceptional performance in hardware model checking competitions [3]. However, bit-level analysis can still face scalability issues, particularly when applied to large word-level designs involving wide and complex data operations. Several approaches have been proposed to extend IC3-style algorithms for better scalability [4]–[9]. These approaches suggest different *abstraction refinement* strategies to reduce the burden on the reasoning engines. Some of these approaches (like [4], [5]) exploit high-level information to perform IC3 at the word level, and have shown impressive results. These techniques replace bit-level reasoning using SAT solvers with word-level reasoning using SMT solvers [10].

In contrast to earlier studies where only bit-level problems were analyzed (e.g. [3], [11]), this paper compares the behavior of different IC3-based techniques on Verilog RTL designs. We performed a rigorous evaluation on 535 Verilog RTL instances including a variety of open-source designs and real-world industrial problems. Our experiments resulted in several key findings, most notably that word-level IC3 solvers can handle designs that are beyond the reach of bit-level solvers.

II. Background

Model checking can be defined as, given a transition system $\mathcal{T}$ (defined by a transition relation T and a set of initial states I), check whether it meets a given property P, and, if not, produce a counterexample demonstrating how $\mathcal{T}$ violates P. Our focus in this paper will be on *safety* properties defined on *finite* transition systems.

The reader is referred to [1], [2] for a detailed description on ideas underlying IC3. Let R_k represent the set of states reachable from the initial states within k steps ($k \geq 0$). The property holds if $R_\infty \subseteq P$. Finding the exact set of reachable states is intractable in practical systems. Instead, IC3 iteratively derives overapproximations of R_k (called frames F_k) such that $F_0 = I$ and $R_k \subseteq F_k$. In its simplest form, the procedure requires two major steps to prove a property:

- **Initiation** - Prove that the property is not trivially violated i.e. $I \subseteq P$ and $I \wedge T \wedge \neg P'$ is unsatisfiable (using primes to denote next states).
- **Consecution** - Derive $F_1 \ldots F_n$ such that $F_i \subseteq P$ and $F_{i-1} \subseteq F_i$ ($i \in \{1, \ldots, n\}$) till two frames converge (i.e. $F_j = F_{j+1}$ for some $j \in \{1, \ldots, n\}$). This is done by iteratively deriving restrictions on frames using 1-step backward reachability queries "SAT? [$F_k \wedge T \wedge c'$]" that check if a state in cube c is reachable from F_k in one step.

III. Review of IC3-based techniques

Several techniques have emerged that extend bit-level IC3 engines to offer better performance. The authors of [7] use *lazy abstraction*, the authors of [8] suggest using uninterpreted functions (UFs) to abstract away expensive data operations, while the authors in [9] use unconstrained new primary inputs to abstract away parts of the system. All these approaches use bit-level IC3 as the core reachability engine.

Certain approaches suggest deploying IC3 at the word level using SMT solvers. The authors in [4] use *implicit predicate abstraction*, perform word-level IC3, and refine the abstraction by adding more predicates. The Averroes system [5] demonstrated the effectiveness of applying *data abstraction* for word-level verification. Averroes 2 [12] is a major upgrade that introduces numerous enhancements in the abstraction, generalization, and refinement stages, is completely incremental in its word-level reasoning, and produces compact word-level inductive invariants when the property holds. These approaches exploit the power of SMT solvers in different ways to perform word-level clause learning, and offer better scalability than techniques that rely on bit-level IC3.

IV. Experimental Setup

We analyzed 535 safety checking problems (Verilog RTL files with SystemVerilog assertions (SVA)) classified as follows:

978-3-9819263-2-3/DATE19/©2019 EDAA

- *opensource*: a set of 141 problems collected from benchmark suites accompanying tools *vcegar* [13] (23 problems), *v2c* [14] (32 problems) and *verilog2smv* [15] (86 problems).
- *industry*: a set of 370 problems collected from industrial collaborators[1]. Their code sizes range from 109 to 22065 lines and number of flip-flops from 6 to 7249.
- *crafted*: a set of 24 simple problems synthetically created.

The six tools we evaluated in this experiment were:

- From ABC version 1.01 [16]:
  - *pdr*: *pdr* is one of the best implementations of the IC3 algorithm at the bit level.
  - *dprove*: *dprove* employs a pre-processing stage using a portfolio of techniques (bounded MC, retiming, simulation, interpolation, etc.) with carefully-tuned heuristics to quickly solve/reduce the problem. If still unsolved, *dprove* invokes *pdr* on the reduced problem.
  - *pdr-nct*: the -nct flags configure *pdr* to use better generalization [17] and to enable *localization* abstraction [6].
- From nuXmv version 1.1.1 [18]:
  - *nuxmv-ic3*: this is the bit-level IC3 implementation in nuXmv based on simplic3 [11]. It starts with a pre-processing step (latch equivalency and temporal decomposition) followed by a state-of-the-art implementation of IC3.
  - *nuxmv-ic3ia*: this is the word-level IC3 implementation in nuXmv using *implicit* predicate abstraction [4].
- *avr*: Averroes 2 [12] is a word-level IC3 implementation using data abstraction and is particularly suited for verifying control-centric properties.

We set up the experiment as shown in Fig. 1. The Verilog designs and SVA are parsed by *yosys* [19] which removes any hierarchy and produces flat RTL. For the nuXmv tools and *avr*, this flat RTL is syntactically translated into the equivalent input formats used by these tools. For the ABC tools, *yosys synthesizes* the RTL to a bit-level implementation.

All experiments were conducted on a cluster of 163 2.5 GHz Intel Xeon E5-2680v3 processors (cores) running 64-bit Linux. Each run was given exclusive access to a single core, with a memory limit of 16 GB and a time limit of 5 hours.

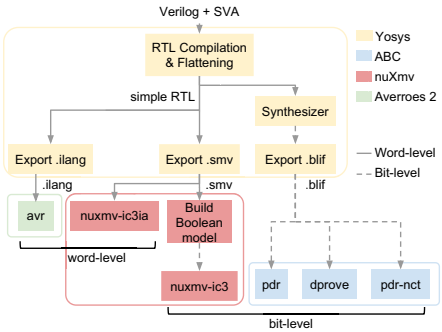


Fig. 1: Verification Tool Chain

[1]We obtained these designs under non-disclosure agreements and, unfortunately, cannot make them publicly available.

V. Experimental Analysis

Space limitations preclude inclusion of all data visualizations generated by this experiment. The raw data, including details on the benchmarks, are available in [20].

A. Aggregate results

Overall, each instance in the suite of 535 problems was successfully solved by at least one tool. A total of 483 problems were proved safe and 52 problems proved unsafe. Table I and Fig. 2 lead to the following observations:

- *avr* solved more problems than any other solver. It especially dominated in the *industry* category, solving all but 2 with 55 of them uniquely solved. *dprove* marginally dominated in the *opensource* category.
- Implicit predicate abstraction helped *nuxmv-ic3ia* solve 4 more problems from the *industry* category than its bit-level version *nuxmv-ic3*.
- In general, the bit-level IC3 implementations in the ABC tools performed significantly better than *nuxmv-ic3*.

B. Runtime comparison

The scatter plots in Fig. 3 compare *avr*'s runtime with the runtimes of the other tools leading to the following observations:

- For the vast majority of problems in the *industry* category, *avr* is able to solve them much faster than the other tools. These problems have wide data paths and expensive data operations that cause the other tools to run slower and time out in many cases, while the data abstraction used by *avr* is particularly effective.
- *avr* is not as effective in the *opensource* category, clearly under performing the ABC tools. These problems involve data-centric properties that are ill-suited for *avr*'s data abstraction and cause *avr* to perform many expensive data refinements to repair its initial abstraction.

Fig. 4 shows additional runtime comparisons that shed further light on the behavior of these tools as follows:

- Fig. 4.a shows that the extra optimizations in *pdr-nct*'s do not result in a discernible improvement over *pdr*.
- Fig. 4.b confirms our earlier observation that the ABC tools, represented by *pdr*, significantly outperform *nuxmv-ic3* in runtime on almost all categories.
- Fig. 4.c shows that *nuxmv-ic3ia* is usually faster than *nuxmv-ic3*, especially for the *industry* category. This seems to confirm that abstraction is critical for large-scale problems that tend to tax the capacity of bit-level analysis.
- Fig. 4.d shows that for problems that can be solved during the pre-processing stage of *dprove*, the runtime of *pdr* without pre-processing can sometimes be faster! Still, pre-processing does help, especially for the *industry* problems.
- Fig. 4.e provides further evidence of the importance of pre-processing. Specifically, in most cases where *dprove* is faster than *avr*, it is because the problem is solved in the pre-processing stage. This suggests that word-level pre-processing techniques similar to the bit-level techniques used in *dprove* may further help scale abstraction based tools such as *nuxmv-ic3ia* and *avr*.

C. Number of solver calls

Fig. 5 shows sample comparisons in the total number of solver calls made by the tools. Generally speaking, these comparisons correlate with the runtime comparisons in Fig. 3 and suggest the following additional observations:

- *avr* requires orders of magnitude fewer solver calls in the *industry* category. This is explained by the fact that the properties being checked in this category seem to be weakly-dependent on data state and that *avr*'s data abstraction eliminates much of the bit-level details leading to a significantly simpler abstract transition system. These differences are less pronounced between *avr* and the other word-level tool *nuxmv-ic3ia*, suggesting that word-level reasoning scales better than bit-level reasoning.
- In the *opensource* category, the mismatch between the data-centric properties being checked and *avr*'s data abstraction causes it to require more solver calls than the other tools.
- Surprsingly, *pdr* and *nuxmv-ic3* differ significantly in the number of solver calls, with *nuxmv-ic3* requiring many more calls. This could potentially be an artifact of the synthesis tool chain used (Fig. 1) where *yosys* synthesizes for *pdr* whereas nuXmv synthesizes for *nuxmv-ic3*. The discrepancy could also be due to implementation choices such as the SAT solver used, the cube generalization procedure, pre-processing, etc., as indicated in [11]. For many other cases, *nuxmv-ic3* requires fewer solver calls compared to *pdr*. However even for these cases, the runtime performance of *nuxmv-ic3* is worse than that of *pdr* implying that perhaps a better SAT engine can benefit *nuxmv-ic3*.
- As expected, *nuxmv-ic3ia* requires fewer solver calls compared to *pdr* confirming our earlier observation that word-level reasoning can scale much better than bit-level.

D. Techniques based on abstraction refinement

Figs. 6.a and 6.b compare the number of refinement iterations required by the three tools that use an abstraction refinement procedure (*pdr-nct*, *nuxmv-ic3ia*, *avr*) and show that *avr* solves most of the *industry* problems without any refinement (left edges of the Figures). This demonstrates the applicability of data abstraction for large industrial designs and justifies previous observations.

E. Bit-level versus word-level verification

Figs. 6.c-e compare different IC3 statistics between *pdr* (bit-level IC3 engine) and *avr* (word-level IC3 with data abstraction). These plots indicate the following:

- In many cases, *pdr* makes orders of magnitude more counterexample to induction (CTI) checks than *avr* and learns many more clauses before solving the problem (Fig. 6.c-d). This affirms why *avr* requires many fewer solver calls compared to *pdr* (Fig. 5.a), and shows the strengths of using a word-level IC3 procedure that can exploit the word-level semantics to learn strong clauses.
- Some *crafted* and *opensource* problems require *avr* to perform more CTI checks and clause learning compared to *pdr* due to *avr*'s data abstraction being ineffective when the property is data-dependent.
- Fig. 6.e compares the number of clauses in the inductive invariant produced by the two tools (for cases where the property is true). Again due to word-level clause learning, *avr* learns strong word-level inductive invariants with many fewer clauses compared to *pdr*.

VI. Conclusions

Our goal in this preliminary study was to better understand the landscape of IC3-based MC techniques by comparing not just bit-level engines, but also word-level techniques to shed light on their suitability for various types of MC problems. Word-level IC3 engines add a layer of abstraction to lift the reasoning above the bit level, and takes advantage of the word-level information to potentially scale to much larger designs.

Acknowledgment

We would like to thank Alberto Griggio for providing a custom version of nuXmv with detailed statistics output.

References

[1] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, 2011, pp. 70–87.

[2] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[3] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *FMCAD*, 2017, pp. 9–9.

[4] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *TACAS*, 2014, pp. 46–61.

[5] S. Lee and K. A. Sakallah, "Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction," in *CAV*, 2014, pp. 849–865.

[6] Y.-S. Ho, A. Mishchenko, R. Brayton, and N. Eén, "Enhancing PDR / IC3 with localization abstraction," 2017.

[7] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and sat-based reachability in hardware model checking," in *FMCAD*, 2012, pp. 173–181.

[8] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *FMCAD*, 2016, pp. 65–72.

[9] Y.-S. Ho, A. Mishchenko, and R. Brayton, "Property directed reachability with word-level abstraction," in *FMCAD*, 2017, pp. 132–139.

[10] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.

[11] A. Griggio and M. Roveri, "Comparing different variants of the IC3 algorithm for hardware model checking." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 1026–1039, 2016.

[12] A. Goel and K. Sakallah, "Averroes 2," http://www.github.com/aman-goel/avr.

[13] "VCEGAR Verification Benchmarks," http://www.cprover.org/hardware/benchmarks/vcegar-benchmarks.tgz.

[14] R. Mukherjee, M. Tautschnig, and D. Kroening, "v2c–a verilog to c translator," in *TACAS*, 2016, pp. 580–586.

[15] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2smv: A tool for word-level verification," in *DATE*, 2016, pp. 1156–1159.

[16] B. L. Synthesis and V. Group, "ABC: A system for sequential synthesis and verification," http://www.eecs.berkeley.edu/ alanmi/abc/, 2017.

[17] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *FMCAD*, 2013, pp. 157–164.

[18] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *CAV*, 2014, pp. 334–342.

[19] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[20] https://github.com/aman-goel/date19eval.

	Tool	Solved (535)	TO	Error / MO	Unique	IN (370)	OS (141)	CR (24)
B	*pdr*	466	69	**0**	1	308	137	21
B	*dprove*	477	57	1	3	315	**138**	**24**
B	*pdr-nct*	466	68	1	1	308	137	21
W	*nuxmv-ic3*	385	133	17	0	228	134	23
W	*nuxmv-ic3ia*	389	92	54	0	232	133	**24**
W	*avr*	**526**	**0**	9	**55**	**368**	134	**24**

TABLE I: Number of problems solved by each tool
TO: timed out, MO: out of memory, Unique: solved uniquely (not solved by others), IN: *industry*, OS: *opensource*, CR: *crafted*, B: bit-level, W: word-level

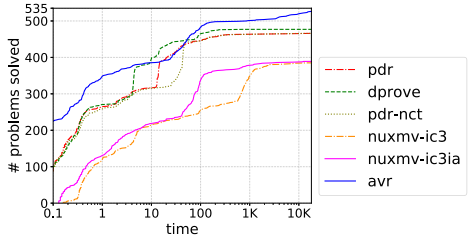


Fig. 2: Survival plot comparing the number of problems solved versus runtime

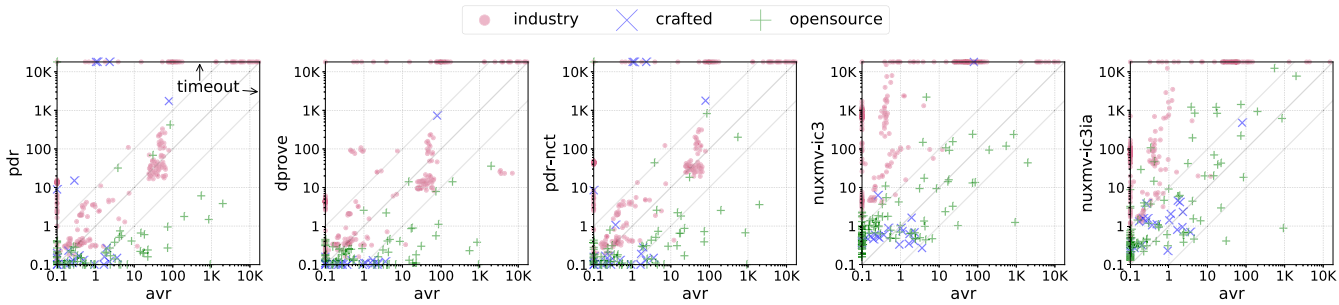
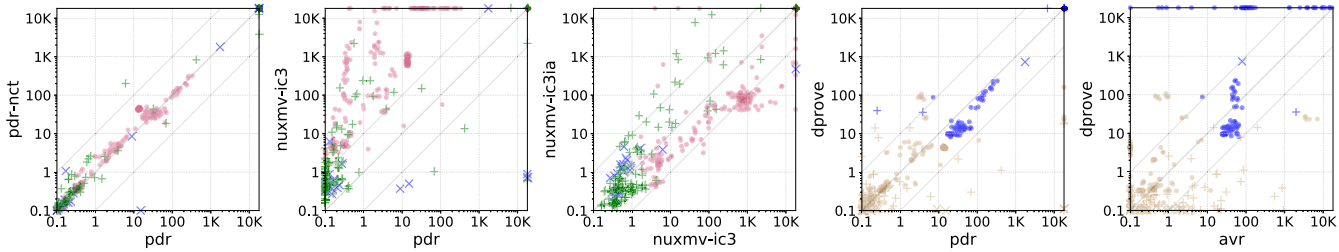
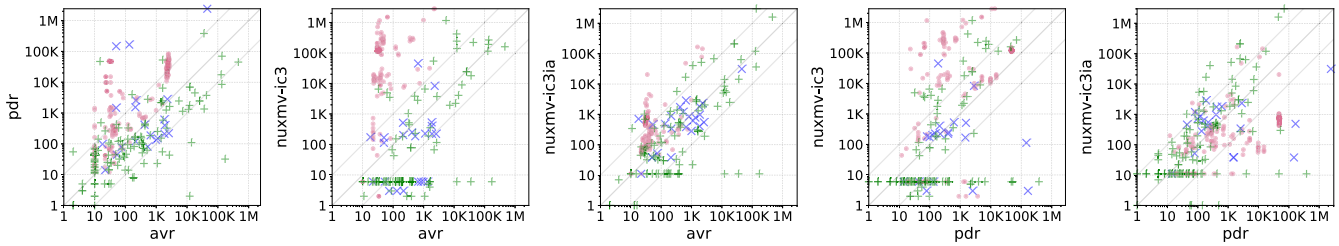


Fig. 3: *avr* runtime comparisons. *avr*'s times are better (resp. worse) above (resp. below) the diagonal.



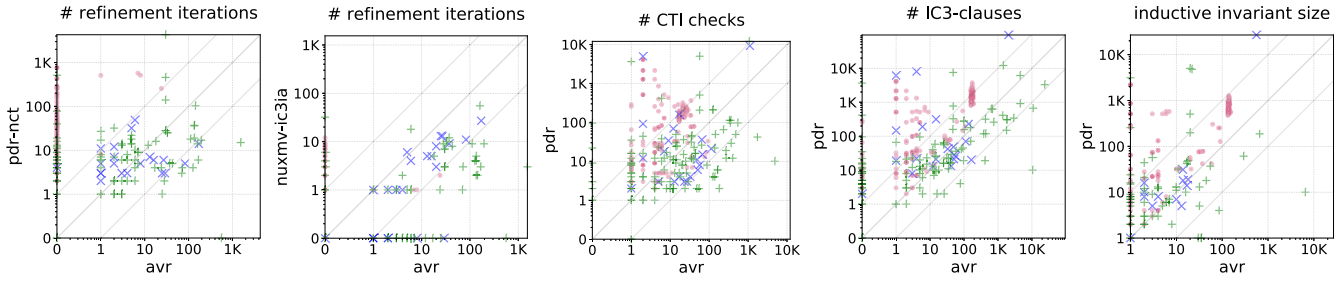
(a) (b) (c) (d) (e)

Fig. 4: Additional runtime comparisons. In parts d and e, instances solved (resp. unsolved) during the pre-processing stage of *dprove* are colored brown (resp. blue).



(a) (b) (c) (d) (e)

Fig. 5: Comparison of number of solver calls (SAT solver calls for *pdr* & *nuxmv-ic3*, SMT solver calls for *nuxmv-ic3ia* & *avr*)



(a) (b) (c) (d) (e)

Fig. 6: Comparison of number of refinement iterations (parts a and b) and IC3 statistics (parts c to e) w.r.t. *avr*
All plots exclude runs in which a tool reported an error or ran out of memory, and all runtimes refer to CPU time in seconds.