Diagnosing Performance Differences in Model Checkers via Runtime-Guided Problem Generation

Yibo Dong*†, Yicong Xu†, Wenjing Deng†, Yu Chen‡, Xiaoyu Zhang†, Jianwen Li†,
Chengyu Zhang§, Geguang Pu†
*National University of Singapore, Singapore
†East China Normal University, China
‡Chuzhou University, China
§Loughborough University, United Kingdom

***Abstract*—Model checking has achieved remarkable success in the hardware domain, largely due to the accumulation of intricate optimizations and finely tuned implementation details. As tools evolve, diagnosing performance differences to better understand the interplay of these factors has become increasingly important. Yet existing problems that reveal such differences are often too large for meaningful inspection, limiting their diagnostic value.**

To address the problem, this paper proposes `AIGROW`, a framework for generating hardware model checking problems, and introduces our experience on diagnosing performance differences in model checkers with the generated problems. `AIGROW` uses a feedback-guided process that evolves problems based on runtime information, selectively retaining those that become more difficult for a target checker. Performance differences are then revealed by evaluating these problems across hardware model checkers that have similar algorithms.

Our evaluation demonstrates that `AIGROW` generates problems that are more than 100 times smaller than those produced by existing generators, while still revealing substantial performance differences. Diagnosing the performance differences has led to concrete improvements in CAR-based checkers: (1) uncovering structural inefficiencies in their exploration strategies, (2) solving 18 previously unsolvable HWMCC'24 problems, and (3) reducing runtime from hours to minutes in several cases.

I. Introduction

Formal verification has been an essential approach in ensuring the correctness of complex systems. Compared to testing, which can only detect the presence of bugs, formal methods aim to prove their absence by exhaustively analyzing all possible behaviors of a system under a formal model. Among these techniques, model checking [12] has emerged as a particularly successful approach in the hardware domain. Unlike software systems which require generalization and refinement, hardware designs are naturally described as finite-state transition systems with well-defined states and transitions, making them highly amenable to exhaustive formal analysis [11].

In hardware model checking (HWMC), the task is to automatically verify whether a hardware design M satisfies a given specification P [2], [11]. If the model checker finds that $M \not\models P$, it returns a counterexample; otherwise, it may produce an inductive invariant as a proof of correctness [41]. Despite the problem being PSPACE-complete in general [36], model checkers have achieved remarkable practical success in verifying real designs [16], [18], [20].

This practical success is driven largely by a rich combination of heuristics, algorithmic optimizations, and low-level implementation decisions. Modern checkers apply sophisticated optimizations [8], [9] in areas such as clause learning, lemma generalization, and SAT encoding, where even small variations can lead to significant differences in performance [22]. Understanding and diagnosing these differences is crucial for both optimization and development.

To support such diagnosis, we argue that the field needs a new class of benchmarks: ones that are **challenging** enough to expose performance gaps, yet **compact** enough to enable precise, fine-grained analysis. Existing large-scale industrial benchmarks, such as those from HWMCC [23], [24], are essential for overall evaluation, but their complexity often masks the internal behavior of model checkers. Their sheer size makes it nearly impossible to trace procedural behaviors in detail. This is especially important for recent advances in model checking, e.g., those related to lemma predication [37], [39], clause management [14], and SAT query design [13], [38], which are closely related to the detailed procedural behavior of model checkers. For such developments, transparency and traceability are essential. Additionally, the total number of such benchmarks is limited (only about a thousand) making them insufficient for evaluating incremental improvements.

Existing tools such as `AIGFUZZ` [1] and `AIGEN` [26] make initial progress toward automated benchmark generation, but they exhibit significant limitations in generating useful benchmarks for identifying potential optimization opportunities in hardware model checkers. `AIGFUZZ` employs purely random generation techniques to rapidly produce AIGER problems. However, despite its efficiency, it rarely generates challenging problems. `AIGEN`, in contrast, aims to generate more complex and varied logic structures by uniformly sampling Boolean functions over a fixed number of inputs. While this approach can lead to structurally complex problems, the resulting benchmarks are often not especially challenging and the exhaustive nature of its generation process severely limits scalability. Moreover, both tools produce circuits with over 15,000 components, making them unsuitable for manual inspection or detailed procedural analysis.

† Geguang Pu is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00019

In this work, we propose a novel and **efficient** approach for generating **compact yet challenging** hardware model checking problems, implemented as a tool called AIGROW. Here, "efficient" refers to the ability to rapidly generate meaningful problems; "compact" refers to small circuit size; and "challenging" to problems that cannot be solved within the typical one-hour time limit used in competitions. Unlike prior tools, AIGROW is **purpose-driven**: its aim is not random diversity or structural complexity, but to reveal performance differences for identifying potential optimization opportunities.

Our approach is inspired by feedback-guided input generation techniques widely adopted in the software testing community, such as AFL [17] and Randoop [33]. These systems use runtime feedback — such as code coverage or execution time — to guide the search for inputs that exercise diverse or difficult behaviors. We adopt a similar strategy: AIGROW evolves hardware circuits by querying a model checker and using its solving time and result status as feedback. This feedback loop is crucial in the hardware domain, because unlike software where adding loops or constraints often correlates with increased difficulty, hardware verification lacks such strict monotonicity. A small structural change, like inserting a latch, can unexpectedly simplify or complicate the verification task. Starting from basic circuits, AIGROW applies small-step structural extensions — such as inserting gates, inverters, or latches — and consults the model checker after each step. If the new instance proves more difficult, it is retained and further evolved. This incremental strategy ensures that the circuit remains small while steadily increasing in difficulty.

Our evaluation demonstrates that AIGROW produces challenging hardware model checking problems significantly more efficiently than current tools. The resulting circuits typically contain fewer than 250 components, orders of magnitude smaller than those circuits that have over 15,000 components produced by AIGEN or AIGFUZZ; yet remain diverse and challenging. These problems effectively reveal performance differences among different model checkers, providing a valuable asset for tool developers and the wider hardware model checking community. Moreover, the problems generated by AIGROW has already led to targeted optimizations in the state-of-the-art CAR checkers, helping identify and fix performance bottlenecks. This underscores AIGROW 's practical utility not only as a benchmark generator, but also as a tool for advancing model checking itself.

II. Preliminary

And-Inverter Graph: An And-Inverter Graph (AIG) is a directed acyclic graph used to represent gate-level hardware circuits in a compact form [5]. It serves as a simplified sequential hardware model tailored for hardware model checking competitions, consisting of only three fundamental components: AND gates, inverters, and latches.

Beyond hardware circuits, AIGs are also commonly used to encode SAT and model-checking problems. Most modern model checkers accept AIG as an input format, and other representations (such as SMV [10]) can be easily converted into AIG using tools provided in the AIGER release.

Hardware model checking: Hardware model checking (HWMC) focuses on verifying whether a finite-state representation of a hardware design satisfies a given temporal property. Unlike software verification, which often involves abstraction to handle infinite or complex control structures, hardware designs are naturally finite-state and structurally regular [11]. This makes HWMC particularly amenable to symbolic techniques such as SAT solving. Over the past few decades, HWMC has become widely adopted in industry, enabling the verification of deep bugs and ensuring correctness in processor pipelines, memory systems, and control logic.

State-of-the-art HWMC techniques are SAT-based and fall into several families, each with different trade-offs in proof power and efficiency. BMC [3] reduces the search for counterexamples to a sequence of SAT problems with increasing bounds. While effective at bug finding, it is incomplete unless a completeness threshold is set. Techniques like IC3/PDR [6] and CAR [31] avoid full unrolling and instead construct or refine inductive approximations of reachable states. Forward and backward variants of CAR further explore different directionality in proof and counterexample search.

Complementary Reachability Analysis (CAR) For completeness, we briefly summarize the key idea of CAR. It maintains two evolving sequences: a U-sequence, representing an under-approximation of the reachable states, and an O-sequence, representing an over-approximation. The procedure iteratively explores states in the U-sequence and checks, via SAT queries, whether they can reach any frame in the O-sequence. If the query is satisfiable, a new reachable state is discovered and added to the U-sequence. If the query is unsatisfiable, the unsat core is used to refine the O-sequence, eliminating unreachable states.

III. The AIGROW Framework

This section presents the AIGROW framework, which is designed to generate compact and challenging hardware model checking benchmarks that expose performance differences between model checking tools, algorithms, or optimizations. Unlike prior work that focuses purely on generating problems, our framework is explicitly on diagnosing model checkers: it aims to create instances that reveal performance divergences in checker behavior and support detailed analysis.

A. Motivation and Framework Design

Performance differences between model checkers often manifest when one tool successfully solves a benchmark while another fails within the time limit. Such divergences are particularly valuable for diagnostic purposes. However, such benchmarks are often large and structurally complex, which makes them difficult to analyze and understand.

AIGROW addresses this by simplifying the overall objective into two sequential tasks: (1) generate compact yet challenging problems for one target checker, and (2) **filter out** instances

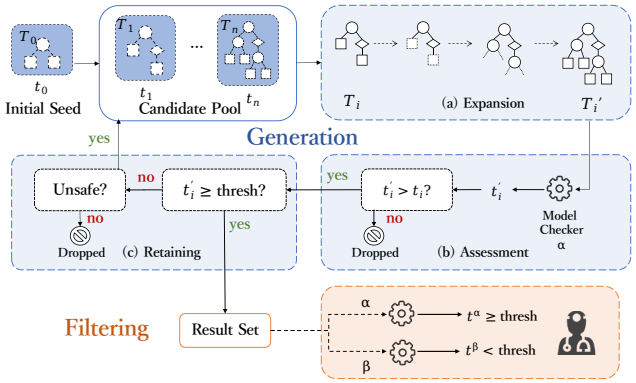


Fig. 1: Overview of AIGROW. T_i represents the candidate i to extend, whose solving time is t_i. At last, those candidates whose solving time exceeds the target limit will be returned together.

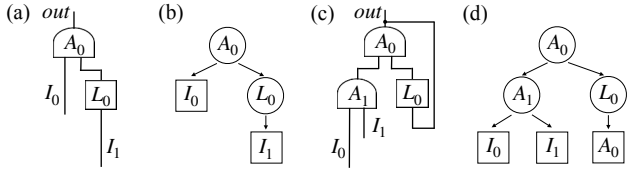


Fig. 2: AIGs and the corresponding tree structures.

that are also difficult for the comparison checker. This workflow enables the isolation of cases that are selectively difficult, thereby providing clearer diagnostic value. An overview of this process is illustrated in Fig. 1.

To generate compact yet challenging instances, AIGROW employs a feedback-guided generation framework inspired by techniques in software fuzzing. The core idea is to use runtime feedback from the target checker to steer structural evolution. Specifically, solving time serves as a proxy for problem difficulty, while the safety status (i.e., safe or unsafe) determines the direction of evolution. Only unsafe instances are expanded, based on the formal guarantee that safe instances remain safe under our expansion rules (which will be proved later). This ensures that the instances in the evolution retain the potential to become increasingly difficult.

The generation loop consists of three key stages:

- **Expanding:** Select a candidate problem and structurally expand it by adding new components and connections.
- **Assessing:** Serialize the expanded design into AIGER format and measure its difficulty using the target checker.
- **Retaining:** Keep the new instance if its solving time surpasses a predefined solving time threshold or if it is unsafe and more difficult than its predecessor.

While AIGROW is compatible with other formats, we implement and demonstrate it using the bit-level AIGER format [5], which is widely adopted in hardware model checking. This choice ensures consistency with standard verification workflows and allows seamless integration with existing tools.

B. Illustrative Example

To illustrate the problem generation process, consider a simple hardware model represented by an AIG, as shown in

Algorithm 1: Runtime guided problem generation

Input : Parameters p, Hardware model checker c, Time thresh *thresh*
Output: Problems $\mathcal{A}$

```
Procedure FeedbackExtension(p, c)
    A ← init()
    queue ← init()
    while timeLimit not reached do
        (a, t_a) ← randomlyChooseFrom(queue);
        a' ← expand(a, p);
        (t'_a, status) ← c.assess(a');
        if t'_a > t_a then
            if t'_a > thresh then
                A ← A ∪ {a'}
            if status == unsafe then
                queue ← queue ∪ {(a', t'_a)}
    return A;
```

Fig. 2(a). This model consists of an AND gate A_0 with inputs I_0 and L_0, where L_0 is a latch, and the output of A_0 serves as the system's output. In our approach, the structural connections between components are represented using a tree, as shown in Fig. 2(b). Each node in the tree corresponds to a component in the AIG — such as an AND gate, a latch, or an input. This hierarchical representation enables systematic modification of connections during the expanding process.

To expand this design, we first "loosen" the tree by disconnecting all the input nodes (I_0 and I_1), making their parent nodes (A_0) extensible. We then randomly generate new components and connections. For example, a new AND gate A_1 might be inserted and connected to the input of A_0, with the output of the existing component A_0 connected to the input of L_0. Afterwards, all the dangling positions are connected to fresh inputs (I_0 and I_1), as shown in Fig. 2(d). This yields a structurally more complex AIG, illustrated in Fig. 2(c). The new AIG is then serialized into the AIGER format and evaluated by a hardware model checker. If the new AIG instance is **unsafe** and its solving time is **longer** than that of the original one, which indicates increased difficulty, the instance is retained for further expanding. Separately, if the solving time exceeds a predefined threshold (e.g., one hour) — regardless of whether the instance is safe or unsafe — it is considered sufficiently challenging and kept for being in the returned benchmarks. This dual retention process allows AIGROW to both progressively generate more challenging problems, while maintaining a manageable size, and accumulate a diverse set of difficult benchmarks.

C. Generation Process

The algorithm of the generation process is shown in Alg. 1. It takes as inputs a parameter setting p and a hardware model checker c, and outputs a set of challenging AIGER problems A. The parameter setting p contains predefined probabilities, which determine the types of new components to generate

(see Line 6). This parameter can be adjusted to control the complexity and characteristics of the hardware design, such as prioritizing certain components or limiting connection patterns. Below is a detailed introduction to each process.

1) Expanding Process: The expanding process begins by selecting a candidate from the pool and applying a *loosen* operation, which disconnects all **input nodes** and marks the corresponding positions as extensible, i.e., available for further structural growth. From this loosened structure, the candidate is expanded by repeatedly applying one of three extension rules to replace extensible positions. Finally, any dangling inputs to the newly generated components are connected to fresh input nodes, completing the model and ensuring syntactic well-formedness. The expanding rules are defined as follows:

- **Rule (a)** insert a new component and connect its output to the input of an extensible node;
- **Rule (b)** connect the output of an existing component to the input of an extensible node, provided that the connection does not form a purely combinational cycle (i.e., a cycle not passing through any latch);
- **Rule (c)** insert an inverter to a newly created edge.

For example, to expand Fig. 2(a) to Fig. 2(c), one needs to apply rule (a), generate a new AND gate A_1 and connect it to the left-hand side of A_0; and apply rule (b), connect the output of A_0 to the input of L_0 (recall that L_0 is loosened in advance). Since no negations exist, no application of rule (c) is required in this expansion.

Each rule plays a distinct role in enabling systematic and expressive structure generation:

- Rule (a) expands the structure by introducing new components, and may introduce new extensible nodes (e.g., at the introduction of an AND gate). Without it, the structure cannot sustain growth since no other rules can replenish extensible nodes.
- Rule (b) connects existing logic to extensible points, enabling the integration of previously generated logic, such as feeding complex expressions into latches to store their values. Without it, structural reuse would be impossible, leading to designs that are either shallow or overly large without increased complexity.
- Rule (c) introduces negation and is necessary to ensure logical completeness. Without it, certain Boolean functions, such as $\neg x \wedge x$ would be unrepresentable.

Together, these rules form a minimal yet complete set for structural extension: (a) and (b) provide controlled growth and interconnection, while (c) ensures logical expressiveness.

This design is driven by a key intuition: **input nodes represent unconstrained behavior** where no logic limits their values, allowing their outputs to vary freely. This is why input nodes are used as the initial values, acting as a starting point with no inherent restrictions. As inputs are progressively replaced with components (e.g., gates, latches, inverters, and their combinations), each replacement introduces new constraints that reduce the system's freedom, gradually narrowing the possible state space. For example, replacing an input with an AND gate connected to both a and $\neg a$ enforces a contradiction; Similarly, replacing an input with an AND gate that connects to any component c and a latch l that stores the negation of its previous value ($\neg \mathsf{Pre}(c)$) prevents the output from being $true$ in two consecutive cycles. Through this process of replacing inputs with components and adding constraints, our method guides the model toward greater behavioral restriction, leading to more complex and challenging verification tasks.

2) Assessing Process: Once the candidate has been expanded, it is serialized into an AIGER problem that can be directly processed by hardware model checkers. The model checker then assesses the problem's difficulty by attempting to solve it. We use solving time as a proxy for problem difficulty, under the assumption that more complex or constrained structures generally require more effort to verify. Our intuition is that modern model checkers — including those based on BMC, PDR, CAR, and interpolation — typically explore the state space incrementally. As such, a longer solving time generally reflects increased structural complexity or deeper reasoning requirements.

While solving time can be influenced by solver heuristics and implementation details, these factors remain consistent when using the same checker, making it a stable and meaningful feedback signal within our framework. Moreover, solving time is a widely accepted metric in this domain: for example, the par-2 score used in the HWMCC competition is based on it, and similar conventions are followed in the SAT community.

Although richer feedback such as proof or counterexample size might seem informative, not all model checkers are designed to produce minimal or even comparable artifacts (e.g., CAR and PDR do not guarantee minimal counterexamples), making such metrics unfair for evaluation. Solving time, in contrast, provides a uniform and practical signal that aligns with standard practice.

3) Retaining Process: After a candidate has been extended and evaluated, the system determines whether the new instance T' should be retained for further expanding or discarded. This decision hinges on its **difficulty**, as measured by the solving time, and its **safety status**, which affects its potential for further transformation.

We adopt the following dual retention strategy:

- If T' is unsafe and its solving time is longer than that of its predecessor T, it is retained in the candidate queue, because this reflects an increase in complexity, suggesting that the new structure T' poses a greater challenge to the model checker.
- Independently, if T' takes longer than a predefined threshold (e.g., one hour) to verify — regardless of its status — it is kept for future return. These hard instances are considered valuable as challenging problems potentially for showing the performance differences.

Importantly, we **only update the queue with unsafe cases**. This decision is grounded in a fundamental property of our expanding process: *safe cases remain safe after expanding*. The underlying reason is that our approach replaces input

nodes with constrained sub-circuits. Since these inputs are originally unconstrained, any replacement narrows the space of allowable behaviors and therefore cannot introduce new unsafe behavior. More formally,

Theorem 1 (Safety Closure under Expanding)**.** *Let T be a sequential circuit with input variables $\vec{x} \in \{0,1\}^n$ and a single output variable o. Suppose that T is* safe*, meaning that for all input valuations $\vec{x}$, the output evaluates to* ***true****:*

$$\forall \vec{x} \in \{0,1\}^n,\ T(\vec{x}) = 1.$$

Then, for any extended circuit T' obtained from T by applying any of the input-replacement rules (i.e., replacing an input with a sub-circuit), the extended circuit T' also satisfies for all input valuations $\vec{y}$:

$$\forall \vec{y} \in \{0,1\}^m,\ T'(\vec{y}) = 1.$$

Proof. Let $\vec{x} = (x_1, \ldots, x_n)$ be the original inputs to T. In the extended circuit T', one or more of these inputs are replaced with computed signals $g_i(\vec{y}_i)$, forming a function$f : \vec{y} \to \vec{x}$ that maps the new inputs $\vec{y} \in \{0,1\}^m$ to a derived vector $\vec{x} \in \{0,1\}^n$.

By construction, the output of T' is:

$$T'(\vec{y}) = T(f(\vec{y})).$$

Since T is safe, we have $T(\vec{x}) = 1$ for *all* $\vec{x} \in \{0,1\}^n$. Therefore, for any $\vec{y} \in \{0,1\}^m$, the derived input $f(\vec{y}) \in \{0,1\}^n$ lies within the domain over which T always outputs 1. Thus,

$$T'(\vec{y}) = T(f(\vec{y})) = 1.$$

In other words, T' restricts the input space of T to a subset of $\{0,1\}^n$, and since T outputs 1 on the entire space, it must also output 1 on any subset. Hence, T' is also safe. □

Here's a concrete example:

Example III.1 (To extend safe cases can only yield safe cases)**.** *Consider a simple circuit $T(x_1, x_2) = \neg(x_1 \wedge \neg x_1 \wedge x_2)$, which is safe since it always evaluates to 1. Indeed, the sub-formula $x_1 \wedge \neg x_1$ is always false, making the entire expression true regardless of x_2.*

Now suppose we construct T' by replacing x_1 with a sub-circuit $x_1' = y_1 \wedge y_2 \wedge \mathsf{Pre}(y_2)$, and x_2 with $x_2' = y_3 \wedge y_4$. Here, $\mathsf{Pre}(y_2)$ denotes the value of y_2 in the previous cycle, retrievable via a latch storing its value. Then:

$$T'(y_1, y_2, y_3, y_4) = T(y_1 \wedge y_2 \wedge \mathsf{Pre}(y_2), y_3 \wedge y_4).$$

Since $T \equiv 1$, any evaluation of $T'(y_1, y_2, y_3, y_4)$ still yields 1. This illustrates that restricting the input space via functional substitution does not violate safety.

As proved, safe cases will always remain safe under our construction rules. What's more, the original safe core also applies to the expanded circuit. This implies that they cannot lead to new unsafe behaviors and therefore cannot contribute to the synthesis of more diverse cases.

By contrast, unsafe cases retain the potential to evolve in multiple directions: they may remain unsafe or become safe[1], while become more complex through further structural changes. By focusing only updates on these unsafe cases, AIGROW ensures that each retained instance has the capacity to guide the generation process toward more difficult, more interesting, and more diverse benchmarks.

This targeted update mechanism distinguishes our approach from purely random generation strategies, which lack directional guidance and often struggle to produce problems of increasing difficulty. By selectively retaining only the most informative and promising instances, AIGROW incrementally drives the evolution of the benchmark set in a principled and efficient way.

IV. Evaluation

The central goal of AIGROW is to generate benchmarks that reveal performance differences between model checkers in a way that is both efficient and practically useful. To validate this capability, we organize our evaluation around the following three research questions, progressing from tool efficiency to benchmark quality and ultimately to practical impact:

- **RQ1: Can AIGROW efficiently generate challenging problems?** We show that AIGROW can quickly produce problems that are unsolvable within standard time limits for various state-of-the-art model checkers, demonstrating its effectiveness as a generator of difficult instances.
- **RQ2: Are the generated benchmarks compact enough for meaningful analysis?** We show that the challenging benchmarks produced by AIGROW are significantly smaller than those generated by existing tools, making them ideal for fine-grained analysis and diagnosis.
- **RQ3: How does retaining safe cases affect the generation of challenging verification problems?** We conducted an ablation study and show that unsafe cases are the primary source of challenging benchmarks, which empirically justifying the design choice of AIGROW to discard safe cases.
- **RQ4: Can the generated benchmarks help developers diagnose and improve modern checkers?** We demonstrate that AIGROW uncovers non-trivial performance gaps, which in turn lead to actionable insights and concrete optimizations. These improvements translate to measurable gains across both synthetic and real-world benchmarks.

A. Evaluation Setup

Environment Setup. We conducted the experiments in a cluster, consisting of 240 nodes with 6720 processor cores altogether (14 processor cores per node) and running at 2.6GHz with 96GB of RAM per node. The operating system is RedHat 4.8.5-16 and the Python version is 3.8.10.

[1]The formal proof is omitted due to space. In short, 'Unsafe' means there exists a violating input valuation, which could be blocked by new constraints.

TABLE I: The number of challenging (unsolved within 1 hour) problems generated by different tools in 24 hours. `AIGROW`-random denotes `AIGROW` without the runtime guided strategy.

	ABC-PDR	IC3ref	F.CAR	B.CAR
AIGROW	**10**	**2**	**10**	5
AIGFUZZ	0	1	8	**12**
AIGEN	0	0	0	0
AIGROW-random	0	0	0	0

Model Checker Candidates. State-of-the-art model checking algorithms include BMC [2], IC3/PDR [7] and CAR [31]. We choose the model checkers ABC [8], IC3ref [25] and SimpleCAR [30] for the evaluation of our methods. The PDR algorithm implemented within ABC is quite efficient and mature. IC3ref is a basic yet efficient implementation of the IC3 algorithm. SimpleCAR is an efficient model checker that implements the CAR algorithm, which can perform the search in two ways, referred to as Backward CAR and Forward CAR.

B. Evaluation results

RQ1: Can `AIGROW` efficiently generate challenging problems? We evaluate the generation efficiency of AIGROW by comparing it with three baselines: AIGFUZZ, AIGEN, and a variant of AIGROW without runtime guidance, referred to as AIGROW-random. Each tool is given 24 hours on a single thread to generate problems targeting four state-of-the-art model checkers: ABC-PDR, IC3ref, Forward-CAR (Forward CAR), and Backward-CAR (Backward CAR). A problem is considered challenging if it cannot be solved by the checker within 1 hour.

Summary of Results. As shown in Table I, AIGROW consistently generates more challenging problems than all the other tools across most checkers. It is the **only** tool capable of producing any unsolved instances for ABC-PDR, a widely used and highly optimized tool. Even without any domain-specific tailoring, AIGROW generates 10 timeouts on ABC-PDR and 2 on IC3ref. AIGROW-random, which disables feedback guidance while retaining the same structural mutation strategy, generates no challenging problems. These results highlight the strength of runtime-guided generation in steering the search toward problem regions that stress the checker's reasoning capabilities.

In contrast, AIGEN fails to produce any challenging problems across all checkers, suggesting that its structural diversity does not translate into hardness. AIGFUZZ performs better on Backward CAR and Forward CAR due to its aggressive random mutation strategy but remains ineffective on IC3ref and ABC-PDR. This indicates that random approaches can occasionally produce hard instances, but do so inconsistently and inefficiently.

Generation Dynamics Over Time. Fig. 3 provides a temporal view of generation performance by plotting the maximum solving time of generated instances as a function of wall-clock time. Despite the computational overhead incurred by runtime assessments, AIGROW is still the first tool to produce timeout-inducing problems on both ABC-PDR and IC3ref. This underscores the efficiency of runtime feedback:

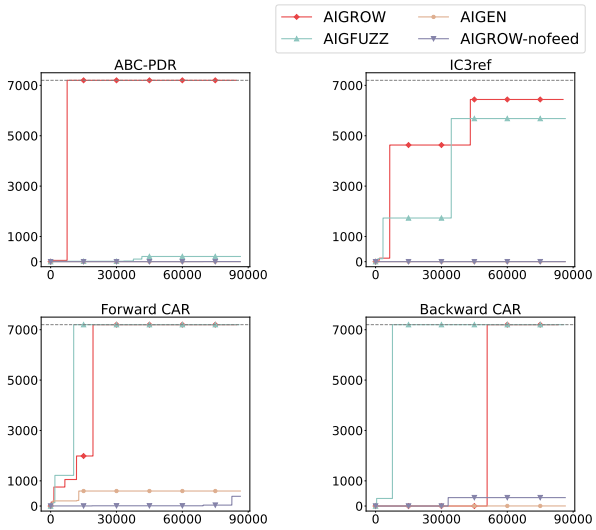


Fig. 3: Comparison of the max solving time. The x-axis denotes the CPU time, and the y-axis indicates the max solving time of a single case. Time limit for model checkers is set to 2 hours.

TABLE II: The parameter settings. Each row represents the appearance probability of input, latch, and AND gate.

	Input	Latch	AND Gate
Setting 1	15%	70%	15%
Setting 2	25%	50%	25%
Setting 3	20%	40%	40%
Setting 4	40%	40%	20%

runtime data not only filters better instances but also accelerates convergence toward difficult regions of the problem space.

Robustness Across Parameters. To test the sensitivity of AIGROW to structural parameters, we evaluate it under four configurations that vary the probability of introducing input nodes, latches, and AND gates (Table II). The default (*Setting 1*) prioritizes latch-heavy designs, while the others emphasize richer combinational logic or higher input variety.

As shown in Fig. 4, AIGROW consistently generates timeout cases on ABC-PDR and Forward CAR across all parameter settings, typically within the first 15,000 seconds. Although the precise timing of hard instances vary across settings, AIGROW remains effective across the board. On IC3ref and Backward CAR, performance differences between settings are more pronounced, but in most scenarios, AIGROW still manages to generate challenging benchmarks. These results demonstrate that the runtime feedback mechanism adapts well to different structural configurations, allowing AIGROW to maintain robust performance under diverse generation heuristics.

> *We conclude that AIGROW effectively generates challenging problems for all evaluated model checkers.*

RQ2: Are the generated problems compact enough for meaningful analysis? We assess the quality of the benchmarks

TABLE III: Average solving time and average size (latches + AND gates) of the top 50 problems on each model checker.

	AIGROW		AIGFUZZ		AIGEN	
	avg. time (s)	*avg. size*	*avg. time (s)*	*avg. size*	*avg. time (s)*	*avg. size*
PDR	1532.59	**219**	11.96	22,790	0.79	180,120
IC3ref	371.37	**224**	562.75	29,036	1.81	180,186
B. CAR	775.94	**51**	1694.85	20,815	2.07	180,204
F. CAR	1599.03	**159**	1389.48	24,570	133.34	180,214

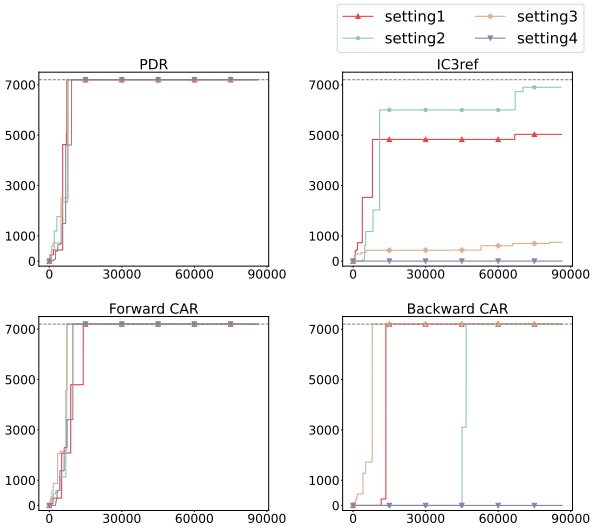
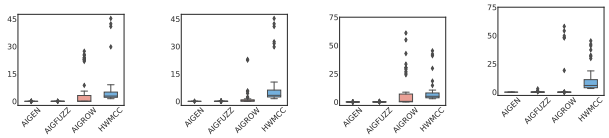
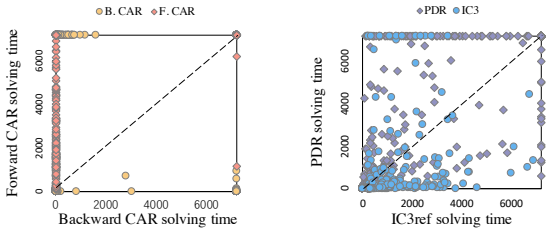


Fig. 4: The performance of AIGROW on different model checkers with different parameter settings. The meaning of x-axis/y-axis is the same as that in Fig.3.



(a) PDR (b) IC3ref (c) Forward CAR (d) Backward CAR

Fig. 5: Box plot of the *quality* of different tools. HWMCC denotes the *quality* of the problems from HWMCC 2015-2017.



(a) Forward vs Backward CAR (b) PDR vs IC3ref

Fig. 6: Comparison of the solving time of different checkers, with problems represented by different colors and shapes. Points close to the diagonal suggest that both checkers exhibit similar performance, while points farther from the diagonal reveal performance disparities.

produced by AIGROW, focusing on two critical criteria: (1) compactness, which ensures that problems are easy to inspect and analyze, and (2) diversity, which enables broad diagnostic coverage. Together, these properties determine whether generated benchmarks are practically useful for exposing and studying performance differences between model checkers.

Compactness. Compact yet difficult benchmarks are ideal for tool developers: they stress-checker capabilities while remaining understandable. To quantify this, we define a quality metric combining solving time and size:

$$quality = \frac{solving\ time}{latch\ nums + AND\ gate\ nums}$$

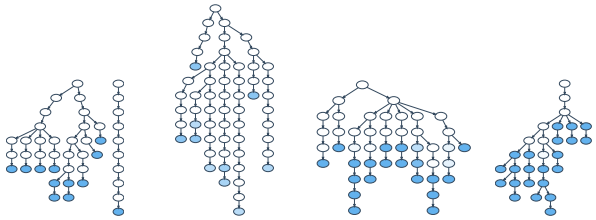
Higher values indicate problems that are both small and hard.

We select the top 50 problems (ranked by the solving time) on each checker generated in RQ1 and compare against AIGFUZZ and AIGEN. This ensures a consistent basis for evaluating benchmark *quality*—defined by difficulty and compactness—which is more informative for diagnosis than raw quantity. Table III shows that AIGROW produces problems that are over **100 times smaller** than those of other tools, while maintaining or exceeding their difficulty levels. This compactness arises naturally from AIGROW's incremental generation process, which builds complexity from minimal circuits rather than mutating large designs. On PDR and Forward CAR in particular, AIGROW achieves both higher average solving time and dramatically reduced size — confirming its ability to isolate "minimal hard cases."

Fig. 5 compares the quality metric across tools and also includes HWMCC 2015–2017 as a gold standard. AIGROW matches the quality of HWMCC cases while using only a fraction of the structural resources, reinforcing its practical advantage for diagnosis-focused applications.

Revealing Performance Differences. In addition to being compact, benchmarks generated by AIGROW are effective at highlighting **performance discrepancies** between model checkers — even when the tools implement similar algorithms.

Fig. 6 compares solving times between pairs of checkers. Each point represents a benchmark, with coordinates determined by the solving time of each checker. Points close to the diagonal indicate similar performance, while those far from it reveal disparities. The upper-right region of each plot corresponds to benchmarks that are difficult for both checkers. AIGROW applies a filtering step to discard these cases, as they offer little insight into performance differences. Instead, it retains benchmarks located away from the diagonal and outside the timeout region, which are selectively difficult for one checker but not the other, and are therefore highly effective at revealing performance issues for in-depth diagnosis.



(a) PDR (b) IC3ref (c) Forward CAR (d) Backward CAR

Fig. 7: The generation procedures of the top-10 hard-to-solve benchmarks (the leaf nodes). In addition to the deepest blue leaf nodes, a branch node representing a timeout AIG is also marked in deep blue. The darker color represents the longer solving time. The node is white when the solving time is less than 1 second.

This design choice is crucial for the diagnostic goal of AIGROW. For example, Fig. 6a shows that Forward and Backward CAR exhibit complementary behavior: several benchmarks are easy for one and hard for the other. A similar pattern is observed in Fig. 6b, where differences between PDR and IC3ref emerge clearly. These off-diagonal instances serve as ideal test cases for understanding implementation-level differences.

Moreover, AIGROW generated **53 problems** that remained unsolved within 7,200 seconds by *all four checkers*. While these cases are not included in the final revealing set due to their lack of differentiation, they form a valuable side product for developers who are interested in generally hard problems.

Problem Diversity. Beyond size and difficulty, structural variety ensures that problems test different reasoning paths in checkers. Fig. 7 illustrates the generation trees for the top 10 hardest problems per checker. Each node represents a generated AIG, with color denoting solving time. The figure shows that hard problems are not evolved from a single seed but stem from a diverse set of ancestors — many of which are trivially solvable. This branching behavior indicates that AIGROW explores multiple structural trajectories, and that small changes can lead to large differences in difficulty — a property valuable for robustness testing. In several cases, a simple mutation causes a sudden leap in solving time, suggesting that AIGROW may help uncover fragile points in the checker's strategy.

We conclude that AIGROW generates high-quality problems that effectively highlight performance differences between model checkers and are useful for analyzing and optimizing model checkers.

RQ3: How does retaining safe cases affect the generation of challenging verification problems?

In Section III, we established that mutating a safe case can only yield another safe case. While this ensures correctness, it is unclear how retaining safe cases affects the generation of challenging verification problems. To answer this, we conduct an ablation study in which we retain safe cases in the generation process for further extension. This allows us to isolate the impact of discarding safe cases and evaluate whether unsafe cases are indeed the primary source of challenging benchmarks.

TABLE IV: The number of challenging (unsolved within 1 hour) problems generated by different tools in 24 hours. AIGROW-safe denotes AIGROW in which safe cases are retained.

	ABC-PDR	IC3ref	F.CAR	B.CAR
AIGROW	**10**	**2**	**10**	**5**
AIGROW-safe	8	0	4	0

TABLE V: Challenging problems generated, along with solving time and parent status. All challenging problems originate from unsafe parents.

Checker	ProblemID	Time (s)	ParentStatus
F.CAR	fcar-697	3819.33	Unsafe
F.CAR	fcar-711	7200.00	Unsafe
F.CAR	fcar-726	7200.00	Unsafe
F.CAR	fcar-875	7200.00	Unsafe
ABC-PDR	abcpdr-681	7200.00	Unsafe
ABC-PDR	abcpdr-731	7200.00	Unsafe
ABC-PDR	abcpdr-736	7200.00	Unsafe
ABC-PDR	abcpdr-771	7200.00	Unsafe
ABC-PDR	abcpdr-794	7200.00	Unsafe
ABC-PDR	abcpdr-797	7200.00	Unsafe
ABC-PDR	abcpdr-805	7200.00	Unsafe
ABC-PDR	abcpdr-816	7200.00	Unsafe

Specifically, we compare two configurations of our tool:

- AIGROW: Original setup, where only unsafe cases are retained.
- AIGROW-safe: Modified setup, where safe cases are retained and extended using the same rules as unsafe cases.

Table IV shows that retaining safe cases (AIGROW-safe) results in fewer challenging problems across all solvers compared to the original setup (AIGROW). Consequently, discarding safe cases in the original design is justified: it focuses the generation process on instances that are more likely to challenge checkers, while retaining correctness. These empirical results also align with our earlier estimation that extending safe cases only produces trivially safe instances.

A closer inspection of the challenging problems confirms that all of them are derived from unsafe parents, as shown in Table V. Recall that extending an unsafe case may yield either a safe or an unsafe instance, whereas mutating a safe case always yields another safe case. Therefore, if a challenging instance has an unsafe parent, its entire lineage must also originate from unsafe cases. This observation further strengthens the conclusion that unsafe cases are the true drivers of benchmark difficulty.

Taken together, these ablation results provide strong empirical justification for the original design choice to discard safe cases. By retaining only unsafe cases for extension, the generation process effectively produces instances that are more likely to challenge checkers. [2]

[2]Since extending a safe case yields only safe instances, we also used them as an oracle to test checker soundness. No soundness bugs were detected.

We conclude that retaining safe cases performs worse than discarding safe cases on generating challenging benchmarks. Therefore, the design of discarding safe cases is empirically justified.

RQ4: Can the generated benchmarks help design new optimizations? Having the revealing problems generated by AIGROW, we further investigate whether the problems can help developers improve model checkers. We choose the best variant [3] [39] of **SimpleCAR** to optimize.

We construct a diagnostic benchmark set by collecting the 50 most time-consuming problems generated for each of the four checkers in RQ1, resulting in 200 total instances. Although benchmarks could be generated specifically for a target tool, we reuse those from RQ1 to demonstrate that the diagnostic value of AIGROW-generated problems generalizes across different model checkers, highlighting their broader utility. **SimpleCAR** fails to solve 14 of these within the standard time limit, while other checkers like ABC-PDR could solve, indicating areas for improvement.

To analyze these cases, we instrument **SimpleCAR** with probes that log multi-dimensional statistics during execution, including the sizes of O- and U-sequences, SAT query behavior, and clause-learning metrics. Thanks to the compact size of AIGROW-generated problems, the search process proceeds quickly, and performance bottlenecks are more easily exposed. In particular, we observe cases where sequence sizes grow unexpectedly large or query behavior fluctuates significantly.

Based on these observations, we propose and implement several targeted optimizations for the model checker:

- **Traversal Order in U-Sequence**: In some cases, the U-sequence grows rapidly, slowing down the search. Moreover, redundant exploration within a single round can stall progress. To address this, we experiment with alternative strategies for prioritizing U-states: (i) prioritize initial states, (ii) prioritize states that produced more unsat cores in previous rounds, and (iii) prioritize states with more descendants across the search history.
- **Proof Obligation Container**: The default LIFO stack sometimes causes starvation of older states. We replace the stack with a priority queue that selects the state with the smallest target frame index, enabling more balanced and strategic exploration.
- **Clause Reduction Frequency.**: SAT query performance was observed to vary with the clause database size. We tune the clause reduction frequency parameter ($r = 1/n$) to find a better trade-off between clause retention and propagation overhead.

We evaluate all optimization variants on the 318 public benchmarks from HWMCC'24 [4]. Table VI reports the number of additional problems solved compared to the original baseline. Each category yields measurable improvements: traversal-based heuristics add up to 6 new safe cases, container changes improve the performance, and clause reduction tuning uncovers both safe and unsafe problems. When combined under a virtual-best configuration, these optimizations solve 18 more benchmarks, including 3 additional unsafe ones.

[3]It is a key component of the multi-core checker **SuperCAR**, which has recently won a bronze medal in the HWMCC'24 competition [4]. Unless otherwise specified, all references to **SimpleCAR** in this RQ refer to this best-performing variant.

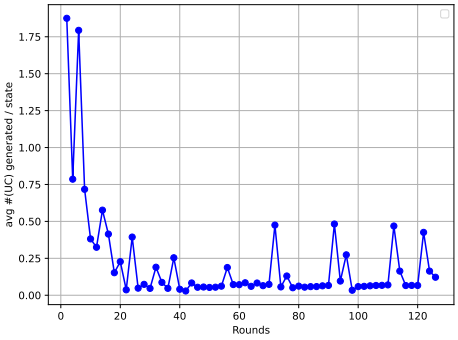


Fig. 8: Runtime statistics of pdr_101. The average number of UCs generated per state decreases steadily, indicating redundancy.

These results demonstrate that AIGROW-generated problems are not only diagnostically useful but can also directly inform the design of impactful optimizations. Furthermore, those optimizations generalize to real-world benchmarks, confirming their broader applicability.

We conclude that the problems generated by AIGROW are valuable to model checker developers and can help them design new useful optimizations.

V. Case Study: How a Tiny Generated Problem Led to an Algorithmic Breakthrough

This case study demonstrates how AIGROW facilitates algorithmic innovation by generating **compact yet challenging** benchmarks that expose inefficiencies hidden in large-scale benchmarks. [4] We describe how a minimal instance produced by AIGROW revealed a critical weakness in a state-of-the-art CAR-based model checker, leading to a new traversal strategy that significantly improved its performance.

Discovery and Diagnosis. During the experiments in RQ1, a small benchmark named pdr_101[5] was generated under the guidance of the ABC-PDR checker. Despite containing only 4 inputs, 28 AND gates, and 152 latches, it consistently triggered a timeout in a competitive CAR-based checker [39], even under a 3600-second time limit. In contrast, other solvers were able to verify it efficiently. This anomaly prompted a detailed diagnostic analysis.

[4]The optimization described in this section is implemented on the same version of the checker evaluated in RQ4. It serves as a detailed example of one category (traversal order) explored in that study.

[5]This case is chosen for exposition. The anomalous behavior was observed in other generated benchmarks as well.

TABLE VI: Results of optimizations on the HWMCC'24 set. 'Basic' refers to the best variant of forward CAR checker [39]. 'PickOrder', 'Container' and 'ReduceRatio' refers to the three categories of optimization, respectively. 'Virtual Best' refers to parallel running and taking the best.

		Safe		Unsafe	
Approach		#(solved)	Gain/Loss	#(solved)	Gain/Loss
Basic		106	/	26	/
PickOrder	Pick-Init	**112**	10/4	26	2/2
	Pick-UC	**112**	7/1	26	0/0
	Pick-Desc	111	6/1	25	1/2
Container	PriorityQueue	108	2/0	26	0/0
ReduceRatio $r = \frac{1}{n}$	n=1	103	2/5	24	0/2
	n=1.4	106	3/3	**27**	1/0
	n=1.8	110	5/1	25	0/1
	n=4	106	2/2	26	0/0
	n=8	107	2/1	26	0/0
	n=16	109	4/1	26	0/0
Virtual Best		**121**	/	**29**	/

To diagnose the issue, we instrumented the model checker with probes to record key metrics during the verification process. These metrics included the number of visited U-sequence states and the number of unsatisfiable cores (UCs) generated in each iteration. One particularly revealing metric was the average number of UCs generated per U-sequence state per round. As shown in Fig. 8, the average number of UCs generated per state generally decreases over time. Although the number of visited U-states continues to grow, the number of distinct UCs saturates. This suggests that many new states do not contribute meaningful refinements to the O-sequence but incur non-negligible overhead.

This finding challenges a common assumption in CAR model checking: that growing the U-sequence necessarily improves convergence. Instead, `pdr_101` revealed that unselective expansion introduces redundant states, leading to slower progress.

Proposed Optimization. To address this issue, we introduced a dynamic traversal strategy called **PickUC**. The idea is to prioritize U-states that have historically produced more distinct unsat cores, under the hypothesis that they are more likely to contribute meaningful refinements. States with lower UC generation are deprioritized or skipped.

We implemented this strategy as a variant of the checker, referred to as CAR-DT, and tested it on `pdr_101`. Table VII shows the result: while the original implementation timed out after reaching frame 275 and generating 79,751 UCs, the optimized version completed the proof in 166 seconds, reaching frame 412 and producing 57,875 UCs — over 70% of the total UCs from the 1-hour baseline.

Impact Across Benchmarks. We applied CAR-DT to additional `AIGROW`-generated problems that exhibited similar saturation patterns. Table VIII shows consistent improvements, often reducing runtime from timeouts to under 10 minutes.

To evaluate generalizability, we ran CAR-DT on a set of real-world benchmarks from the HWMCC'24 benchmark suite [4]. As shown in Table IX, the optimized version achieved up to 4.55% more problem completions. Even when both versions succeeded, CAR-DT typically solved problems faster.

TABLE VII: Detailed runtime comparison on `pdr_101`.

	Original CAR	**CAR-DT (PickUC)**
Runtime	$>$ 3600 s (timeout)	166 s
Final Frame Reached	275	412 (proved safe)
Total # of UCs	79,751	57,875

TABLE VIII: Impact of CAR-DT optimization inspired by `pdr_101`.

Benchmark	**#(And+Latch)**	**Original CAR (s)**	**CAR-DT (s)**
`pdr_97`	40 + 204	timeout	**130.03**
`pdr_103`	39 + 169	timeout	**464.58**
`fcar_153`	58 + 180	856.23	**109.16**
`ic3ref_487`	18 + 53	2333.53	**633.04**
`ic3ref_560`	16 + 55	3185.52	**2.15**

Conclusion. This case study highlights how `AIGROW` enables algorithmic advances by generating small, diagnosable problems that reveal performance pathologies in otherwise mature tools. In this instance, it led to the discovery of a redundant traversal pattern in CAR and inspired an optimization that generalizes to broader benchmarks.

VI. Related Work

A. Feedback-Guided Generation Tools

Feedback-guided generation dynamically guides test case creation using runtime feedback. Pioneering work in software testing, such as Eclat [32] and Randoop [34], [35], demonstrated the effectiveness of this approach for generating behaviorally diverse test cases. Subsequent efforts enhanced feedback mechanisms: GenRed [27] integrated coverage-guided diversification, while Garg *et al.* [19] combined feedback with concolic execution to improve test driver quality.

In hardware model checking (HWMC), *Hammer* [43] adapts feedback-guided techniques to mutate existing benchmarks, exposing bugs in model checkers. While Hammer represents

TABLE IX: Performance comparison on latest HWMCC benchmarks.

HWMCC'24 Benchmark	Original CAR (s)	CAR-DT (s)
ILA-Piccolo-BEQ-sanity	$>$ 3600 (timeout)	**3397.90**
ILA-Piccolo-BGEU-sanity	$>$ 3600 (timeout)	**3519.51**
Problem05-label42+token-ring.08.cil-2	$>$ 3600 (timeout)	**2883.56**
Problem10-label08	$>$ 3600 (timeout)	**2812.81**
a16-p148	$>$ 3600 (timeout)	**1870.37**
brp2.3.prop2-func-interl	$>$ 3600 (timeout)	**1310.36**
qspiflash-dualflexpress-divfive-p162	$>$ 3600 (timeout)	**1156.33**
yosyshq-appnote-123-cv32e40x-p500	$>$ 3600 (timeout)	**2099.85**
yosyshq-appnote-123-cv32e40x-p502	$>$ 3600 (timeout)	**2774.43**
yosyshq-appnote-123-veer-axi-p62	$>$ 3600 (timeout)	**121.03**

a significant advancement in HWMC testing, its mutation-based strategy has critical limitations: (1) it relies on fixed input seeds, restricting diversity and scalability; (2) generated benchmarks are non-compact, hindering developer adoption due to bloated instance sizes; and (3) it provides no guarantee on the validity of the generated cases. A direct experimental comparison with Hammer would be methodologically unsound: their mutation-based approach critically depends on seed corpus quality, whereas our generation-first paradigm eliminates such dependency. Furthermore, Hammer's outputs require post-hoc filtering to remove invalid cases [43], while our method guarantees semantic validity by construction. Any comparison would either (1) use non-optimized seeds (unfair to Hammer) or (2) introduce bias via curated seeds, thus precluding rigorous evaluation.

Our work diverges fundamentally: instead of mutating seeds, we employ a *generation-first* feedback-guided approach. This strategy **eliminates seed dependency**, ensures semantic validity by construction, and systematically optimizes for compact and challenging problems, which addresses Hammer's key shortcomings. Furthermore, our reliance on generic runtime feedback (time and status) makes the `AIGROW` approach more readily generalizable across different model checking paradigms compared to tools reliant on domain-specific mutations or oracles.

B. Hardware Model Checking Benchmark Generators

Existing HWMC-specific generators prioritize scalability over benchmark quality. `AIGEN` [26] produces syntactically valid benchmarks but lacks mechanisms to ensure challenge, often generating trivial cases. Furthermore, its outputs are not compact, limiting practical utility. Fuzzing tools like `AIGFUZZ` [1] and *Fuzz-btor2* [40] focus on detecting model checker bugs via large-scale input exploration. While they support diverse gate types and coverage metrics, their outputs are inherently bloated, as they prioritize volume over minimality. The hardness of their generated instances often stems from the large size (e.g., triggering state sequence explosion in checkers like Backward CAR) rather than nuanced structure. Such benchmarks are unsuitable for use cases requiring human analysis, such as developer-authored case studies.

Our tool bridges this gap by generating *compact yet challenging* benchmarks through a novel mutation strategy. Unlike prior mutation-based tools like Hammer that require existing seed instances as input, we mutate primitive components rather than pre-defined complex cases. This allows us to construct benchmarks *from scratch* while preserving semantic validity through constrained expansion rules.

Our feedback mechanism is uniquely driven by *solver runtime*—a direct proxy for problem difficulty. By iteratively mutating components and selecting variants that maximize solver time, we systematically evolve simple initial structures into compact yet challenging benchmarks. Crucially, this eliminates the need for post-hoc filtering (e.g., cyclic dependency detection in Hammer), as our mutation process inherently avoids generating semantically invalid cases.

C. Testing Verification Tools

The reliability of verification tools is critical for safety-critical systems. Recent efforts to test software model checkers include differential testing [29] and reachability-guided fault injection [42]. For hardware verification, Kaufmann *et al.* [28] uncovered multiplier verification flaws using mutation-based fuzzing. However, these methods depend on high-quality benchmarks to reveal deep flaws—a requirement unmet by prior generators like Hammer and *AIGen*.

Our benchmarks address this need: their compactness enables rapid iteration, while their inherent complexity stresses model checkers more effectively than prior tools. This combination supports both bug detection and performance profiling, advancing the state of verification tool testing.

VII. Conclusion

This experience paper presents `AIGROW`, a framework for generating hardware model checking problems that are both compact and challenging—a combination essential for diagnosing performance differences among model checkers. Unlike prior tools that focus on maximizing structural complexity or randomness, `AIGROW` is explicitly designed to support fine-grained analysis of model checker behaviors.

By guiding problem construction with runtime feedback and reducing the discovery of performance divergence to a one-checker-driven search followed by multi-checker filtering, `AIGROW` efficiently identifies instances that are difficult for one tool but easy for others. This diagnostic framing is critical for uncovering the potential optimization opportunities.

Our experiments show that `AIGROW` produces orders-of-magnitude smaller problems than the problems generated by existing generators, while still exposing significant performance discrepancies. These problems are not only useful for empirical evaluation, but have already driven algorithmic improvements—leading to optimizations that solve previously intractable problems and reveal long-hidden bottlenecks.

All our implementations and generated problems can be accessed from the GitHub [21] and Zenodo [15].

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. We sincerely thank Manuel Rigger for his valuable suggestions on improving the work and thank Xueying Du for her assistance in adjusting and improving the figures.

References

[1] AIGER, http://fmv.jku.at/aiger/

[2] 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)

[3] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Handbook of satisfiability **185**(99), 457–481 (2009). https://doi.org/10.3233/978-1-58603-929-5-457

[4] Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Narodytska, N., Rümmer, P. (eds.) Proceedings 24th International Conference on Formal Methods in Computer-Aided Design (FMCAD'24). p. 7. TU Wien Academic Press (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_6

[5] Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)

[6] Bradley, A.R.: SAT-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011). https://doi.org/https://doi.org/10.1007/978-3-642-18275-4_7

[7] Bradley, A.R.: SAT-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)

[8] Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22. pp. 24–40. Springer (2010)

[9] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Proceedings of the 16th International Conference on Computer Aided Verification. p. 334–342. Springer-Verlag, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-319-08867-9_22

[10] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**(4), 410–425 (2000). https://doi.org/https://doi.org/10.1007/s100090050046

[11] Clarke, E., Gupta, A., Jain, H., Veith, H.: Model Checking: Back and Forth between Hardware and Software, pp. 251–255. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[12] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(2), 244–263 (1986)

[13] Dong, Y., Chen, Y., Li, J., Pu, G., Strichman, O.: Revisiting assumptions ordering in car-based model checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2025)

[14] Dong, Y., Wu, X., Li, J., Pu, G., Strichman, O.: Accelerating car-based model-checking with multiple unsatisfiable cores (2025)

[15] Dong, Y., Xu, Y., Deng, W.: Docker image (Aug 2025). https://doi.org/10.5281/zenodo.16948503

[16] Dong, Y., Zhang, X., Xu, Y., Cai, C., Chen, Y., Miao, W., Li, J., Pu, G.: Lightf3: A lightweight fully-process formal framework for automated verifying railway interlocking systems. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1914–1925. ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3611643.3613874

[17] Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++: combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. WOOT'20, USENIX Association, USA (2020)

[18] Fix, L.: Fifteen Years of Formal Property Verification in Intel, pp. 139–144. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_8

[19] Garg, P., Ivančić, F., Balakrishnan, G., Maeda, N., Gupta, A.: Feedback-directed unit test generation for c/c++ using concolic execution. In: ICSE. pp. 132–141 (2013)

[20] Gerth, R.: Model checking if your life depends on it: a view from intel's trenches. In: Dwyer, M. (ed.) Model Checking Software. pp. 15–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)

[21] Artifact. https://github.com/AnonymousAccO-O-O/AIGROW-artifact

[22] Griggio, A., Roveri, M.: Comparing different variants of the ic3 algorithm for hardware model checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(6), 1026–1039 (2015)

[23] HWMCC 2015. http://fmv.jku.at/hwmcc15/ (2015), http://fmv.jku.at/hwmcc15/

[24] HWMCC 2017. http://fmv.jku.at/hwmcc17/ (2017), http://fmv.jku.at/hwmcc17

[25] IC3Ref. https://github.com/arbrad/IC3ref

[26] Jacobs, S., Sakr, M.: Aigen: Random generation of symbolic transition systems. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 435–446. Springer (2021)

[27] Jaygarl, H., Lu, K., Chang, C.K.: Genred: A tool for generating and reducing object-oriented test cases. In: COMPSAC. pp. 127–136 (2010)

[28] Kaufmann, D., Biere, A.: Fuzzing and delta debugging and-inverter graph verification tools. In: TAP. vol. 13361, pp. 69–88 (2022)

[29] Klinger, C., Christakis, M., Wüstholz, V.: Differentially testing soundness and precision of program analyzers. In: ISSTA. pp. 239–250 (2019)

[30] Li, J., Dureja, R., Pu, G., Rozier, K.Y., Vardi, M.Y.: Simplecar: An efficient bug-finding tool based on approximate reachability. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30. pp. 37–44. Springer (2018)

[31] Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: ICCAD. pp. 95–100 (2017)

[32] Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: ECOOP. vol. 3586, pp. 504–527 (2005)

[33] Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. p. 815–816. OOPSLA '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1297846.1297902

[34] Pacheco, C., Lahiri, S.K., Ball, T.: Finding errors in. net with feedback-directed random testing. In: ISSTA. pp. 87–96 (2008)

[35] Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE. pp. 75–84 (2007)

[36] Schnoebelen, P.: The complexity of temporal logic model checking. Advances in modal logic **4**(35), 393–436 (2002)

[37] Su, Y., Yang, Q., Ci, Y.: Predicting lemmas in generalization of ic3. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. DAC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3649329.3655970

[38] Su, Y., Yang, Q., Ci, Y., Li, Y., Bu, T., Huang, Z.: Deeply optimizing the sat solver for the ic3 algorithm. arXiv preprint arXiv:2501.18612 (2025)

[39] Xia, Y., Becchi, A., Cimatti, A., Griggio, A., Li, J., Pu, G.: Searching for i-good lemmas to accelerate safety model checking. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 288–308. Springer Nature Switzerland, Cham (2023)

[40] Xiao, S., Zhang, C., Li, J., Pu, G.: Fuzzbtor2: A random generator of word-level model checking problems in btor2 format. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 36–43. Springer (2023)

[41] Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model checking results. In: Computer Aided Verification. pp. 363–386. Springer International Publishing, Cham (2021)

[42] Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G., Su, Z.: Finding and understanding bugs in software model checkers. In: ESEC/FSE. pp. 763–773 (2019)

[43] Zhang, C., Sun, M., Li, J., Su, T., Pu, G.: Feedback-guided circuit structure mutation for testing hardware model checkers. In: ICCAD (2021)