IC3-Evolve: Proof-/Witness-Gated Offline LLM-Driven Heuristic Evolution for IC3 Hardware Model Checking

Mingkai Miao[1,*], **Guangyu Hu**[2,*], **Ziyi Yang**[1] and **Hongce Zhang**[1,2,†]

[1]Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China
[2]Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
{mmiao815, zyang957}@connect.hkust-gz.edu.cn, ghuae@connect.ust.hk, hongcezh@hkust-gz.edu.cn

Abstract

IC3, also known as property-directed reachability (PDR), is a commonly-used algorithm for hardware safety model checking. It checks if a state transition system complies with a given safety property. IC3 either returns UNSAFE (indicating property violation) with a counterexample trace, or SAFE with a checkable inductive invariant as a proof of safety. In practice, the performance of IC3 is dominated by a large web of interacting heuristics and implementation choices, making manual tuning costly, brittle, and hard to reproduce.

This paper presents `IC3-Evolve`, an automated offline code-evolution framework that utilizes an LLM to propose small, slot-restricted and auditable patches to an IC3 implementation. Crucially, every candidate patch is admitted only through proof-/witness-gated validation: SAFE runs must emit a certificate that is independently checked, and UNSAFE runs must emit a replayable counterexample trace, preventing unsound edits from being deployed. Since the LLM is used only offline, the deployed artifact is a standalone evolved checker with zero ML/LLM inference overhead and no runtime model dependency. We evolve on the public hardware model checking competition (HWMCC) benchmark and evaluate generalization to unseen public and industrial model checking benchmarks, showing that `IC3-Evolve` can reliably discover practical heuristic improvements under strict correctness gates.

1 Introduction

Formal property checking is a cornerstone of modern hardware verification. As integrated circuits grow in complexity, their state spaces quickly exceed what simulation can effectively explore, leaving deep corner-case behaviors untested, causing severe economic loss, thus motivating formal analysis in hardware verification, exemplified by hardware safety model checking.

For hardware safety model checking, IC3 [Bradley, 2011] (also known as property-directed reachability, PDR [Eén *et al.*, 2011]) is a widely-used state-of-the-art algorithm. Importantly, IC3 is *proof-producing*: it either returns `UNSAFE` with a replayable counterexample trace, or returns `SAFE` with a checkable inductive invariant that certifies the property.

While IC3 provides a strong algorithmic foundation, its practical performance depends critically on numerous heuristics and engineering decisions—including generalization, clause propagation, and interactions with the underlying Boolean satisfiability (SAT) engines. As a result, improving an IC3 checker is often more about heuristic engineering than changing the core algorithmic framework. Unfortunately, this engineering loop is costly and brittle [Le *et al.*, 2021; Hu *et al.*, 2024]: even small heuristic changes typically require re-running large benchmark suites to reach stable conclusions; gains on one family of instances can easily induce regressions elsewhere. Manual tuning therefore demands substantial domain expertise [Le *et al.*, 2021; Hu *et al.*, 2024], making it difficult to reliably discover robust heuristic combinations.

Recent advances in machine learning have also begun to influence Electronic Design Automation (EDA) and formal verification. In the context of IC3, prior work such as NeuroPDR [Hu *et al.*, 2023] and DeepIC3 [Hu *et al.*, 2024] explores learning-augmented solvers: a trained model predicts candidate clauses to guide IC3's proof search, showing promising results on selected benchmarks.

However, learning-augmented IC3 solvers often face practical barriers to industrial deployment. Learned components introduce training and inference costs, and their benefits can degrade on large-scale or heterogeneous workloads where robustness and throughput are paramount. Moreover, placing ML inference in the runtime loop adds latency and increases performance variance, prohibiting predictable behavior and integration into production flows. In recent HWMCC editions, the strongest-performing entries remain predominantly conventional model checkers rather than learning-augmented solvers [hwm, 2024; hwm, 2025a].

Another line of work uses LLMs *in* the verification loop to provide per-instance auxiliary hints (e.g., candidate invariants or guidance) to an existing checker [Janßen *et al.*, 2024; Pirzada *et al.*, 2024; Wu *et al.*, 2024]. Such hints must be independently validated before use; moreover, integrating online LLM calls at verification time raises practical concerns about latency, reproducibility, and deployment.

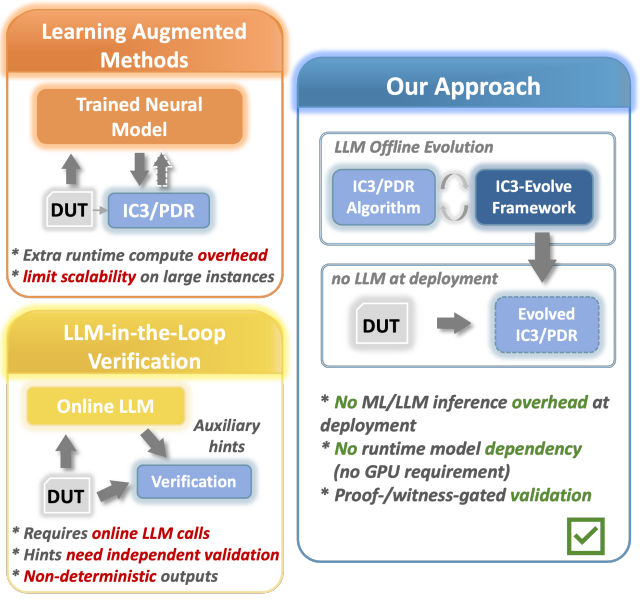

Figure 1: Comparison of learning-augmented IC3, LLM-in-the-loop verification, and our approach. `IC3-Evolve` evolves IC3 code offline and promotes edits only via proof-/witness-gated validation, yielding a standalone checker with no runtime model dependency.

These trends suggest both an opportunity and a bottleneck: today's strongest IC3 checkers remain largely heuristic-engineered, yet manual tuning is labor-intensive and fragile. Meanwhile, recent LLM-based "code evolution" systems show that iterative propose–evaluate–refine loops can autonomously improve nontrivial algorithmic code under evaluation feedback. For example, AlphaEvolve [Novikov *et al.*, 2025] demonstrates evaluator-driven evolutionary search for algorithmic optimization, while AutoSAT [Sun *et al.*, 2024] and SATLUTION [Yu *et al.*, 2025] apply related ideas to evolve SAT-solver heuristics under rigorous validation.

Motivated by this trend of research, we present `IC3-Evolve`, an offline LLM-driven framework that treats IC3 heuristic engineering as controlled code evolution. As shown in **Figure 1**, rather than placing ML/LLM inference in the verification-time loop, `IC3-Evolve` uses LLMs to propose constrained, auditable code edits to an IC3 checker and promotes edits only after proof-/witness-gated validation. The resulting artifact is a standalone evolved checker: the LLM is used only offline, so deployment incurs no ML/LLM inference overhead and no runtime model dependency. Building on this foundation, we introduce a workflow that explores both local and cross-module heuristic choices and evaluate the evolved engine on public and industrial benchmarks.

We make the following contributions:

- We introduce `IC3-Evolve`, an offline LLM-driven framework for evolving IC3 heuristics through slot-restricted, auditable code edits, compiling validated improvements directly into the checker (with no inference-time overhead at deployment).
- We design and implement a heuristic-evolution workflow that explores local refinements and cross-module changes, including the **Compass & Jump** strategy to guide search across the solver's modification space.
- We enforce correctness via proof-/witness-gated validation: a proposed edit is accepted only if it passes hard checks with validation, preventing unsound changes from being deployed.
- We evaluate `IC3-Evolve` by evolving on the public HWMCC suite and assessing generalization on unseen public and industrial verification benchmarks from production flows, demonstrating that `IC3-Evolve` can discover practical heuristic improvements under strict correctness gates.

2 Background

2.1 IC3 in a Nutshell

IC3 is a proof-producing safety model checking algorithm: it tries to prove that no reachable execution can enter a "bad" state. Operationally, it alternates between (i) searching for counterexamples to the current proof attempt and (ii) learning clauses that rule them out. It either returns `UNSAFE` with a replayable counterexample trace, or returns `SAFE` with a checkable inductive invariant that certifies the property. Throughout the paper, we use proof to denote the SAFE certificate (inductive invariant), and witness to denote the UNSAFE counterexample trace.

We consider a finite-state transition system with state variables $\boldsymbol{x}$, input variables $\boldsymbol{y}$, initial-state predicate $I(\boldsymbol{x})$, transition relation $T(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}')$, and a safety property $P(\boldsymbol{x})$, where $\boldsymbol{x}'$ denotes the next-state copy of $\boldsymbol{x}$. IC3 [Bradley, 2011] decides whether all reachable states satisfy P.

IC3 maintains a sequence of *frames* $F_0, \ldots, F_k$, where $F_0 = I$ and each F_i is represented as a formula over $\boldsymbol{x}$. Intuitively, F_i over-approximates the set of states reachable within i steps. The frames are monotone ($F_i \Rightarrow F_{i+1}$), and for $i \geq 1$ they are constrained to satisfy the property ($F_i \Rightarrow P$). Additionally, the sequence satisfies the step condition: for all $0 \leq i < k$, $F_i(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}') \Rightarrow F_{i+1}(\boldsymbol{x}')$. IC3 strengthens this sequence by learning *lemmas* that exclude unreachable states while preserving all reachable behaviors.

Algorithm 1 sketches the main loop of IC3. We use a *cube* to denote a conjunction of literals representing a (partial) state assignment, and a *clause* to denote a disjunction of literals used as a lemma. At a high level, IC3 alternates between two phases: (i) *blocking*, which finds a counterexample-to-induction (CTI) cube s by checking $F_k \wedge T \wedge \neg P'$ (an F_k-state with a bad successor) for satisfiability and attempts to block s by learning a new lemma; and (ii) *propagation*, which tries to push learned lemmas forward to higher frames. If blocking a proof obligation reaches level 0, IC3 can construct a counterexample trace and returns `UNSAFE`. If two adjacent frames become identical ($F_i = F_{i+1}$), that common frame is a 1-step inductive invariant implying P, and IC3 returns `SAFE`.

2.2 Key Subroutines and Heuristic Touchpoints

Key point. Most implementation choices in IC3 control *search order* and *resource allocation* (e.g., which obligation to tackle next, how aggressively to generalize, and how much

Algorithm 1: IC3 Algorithm Overview

```
Input  : ⟨I(x), T(x, y, x′), P(x)⟩
Output: SAFE or UNSAFE
// x′ denotes the next-state copy of x;
   P′ abbreviates P(x′)
F_0 ← I; F_1 ← P; k ← 1; Q ← ∅;
// Q stores proof obligations (s, i)
if I ∧ ¬P is SAT or I ∧ T ∧ ¬P′ is SAT then
    return UNSAFE
end
while true do
    while F_k ∧ T ∧ ¬P′ is SAT with witness cube s do
        if not BlockOne(s, k) then
            return UNSAFE
        end
    end
    PushClauses(k);
    if ∃i < k : F_i = F_{i+1} then
        return SAFE
    end
    k ← k + 1; F_k ← P;
end
Function BlockOne(s, k):
    insert (s, k) into Q;
    // internally calls PredGen and
       IndGen while discharging Q
    return BlockProofObligations(Q);
```

Algorithm 2: IC3 Key Subroutines

```
// s′ and c′ denote the primed versions
   obtained by substituting x ↦ x′
Function BlockProofObligations(Q):
    while Q ≠ ∅ do
        (s, i) ← Pop(Q);
        if F_i ∧ s is UNSAT then continue;
        if i = 0 then
            return false
        end
        if F_{i−1} ∧ T ∧ s′ is SAT then
            p ← PredGen(GetWitness(), i − 1, s);
            insert (s, i) into Q;
            insert (p, i − 1) into Q;
        end
        else
            c ← IndGen(s, i);
            for j = 1 → i do
                F_j ← F_j ∧ c
            end
        end
    end
    return true;
Function PredGen(p, j, s):
    // Drop literals from predecessor
       cube p while preserving
       reachability into s
    return MinimizePred(F_j, T, p, s);
Function IndGen(s, i):
    // Return c ⊆ ¬s such that I ⇒ c and
       F_{i−1} ∧ c ∧ T ⇒ c′
    return RelInductiveMinimize(I, F_{i−1}, T, ¬s);
Function PushClauses(k):
    for i = 1 → k − 1 do
        foreach c ∈ F_i \ F_{i+1} do
            if F_i ∧ T ∧ ¬c′ is UNSAT then
                F_{i+1} ← F_{i+1} ∧ c;
        end
    end
```

SAT effort to spend). They do not affect soundness as long as every learned lemma is validated by the corresponding SAT checks and SAFE is returned only at a validated fixpoint; under resource limits they mainly affect practical completeness via timeouts. This makes these choices prime targets for systematic heuristic engineering (and for our offline code evolution later).

Algorithm 2 expands the key subroutines and makes explicit where heuristics dominate performance in practice. We use the cube/clause terminology as defined above. Most of IC3's runtime is spent answering incremental SAT queries of two types: (1) *predecessor-search* queries (for recursive blocking), and (2) *relative-inductiveness* queries (for generalization and pushing).

Proof obligations (BLOCKPROOFOBLIGATIONS). IC3 maintains a worklist Q of proof obligations (POs) (s, i), where s is a cube to be blocked at frame i. The policy for popping and re-inserting POs (e.g., depth-first vs. best-first, priority rules, aging) can strongly influence convergence: it can bias the solver toward quickly refuting shallow CTIs (bug finding) or toward accumulating stronger lemmas for deeper proofs. Practical implementations also rely on budgeting/backoff to avoid repeatedly spending SAT effort on "stuck" POs. Since frames are continuously strengthened, queued POs can become stale and are typically skipped once they are already blocked by the current target frame (i.e., $F_i \wedge s$ becomes UNSAT).

Predecessor generalization (PREDGEN). When a PO (s, i) cannot be blocked directly, IC3 queries $F_{i-1} \wedge T \wedge s'$ to obtain a predecessor cube p and enqueues $(p, i-1)$. PREDGEN attempts to generalize p (e.g., by dropping literals or using partial models) while preserving reachability into s. Stronger predecessor generalization can reduce search depth and the number of POs, but it typically requires extra SAT work and interacts heavily with the worklist strategy.

Inductive generalization (INDGEN). Given a CTI cube s at level i, INDGEN derives a blocking clause $c \subseteq \neg s$ such that c is relatively inductive w.r.t. F_{i-1} and does not exclude any initial state: $I \Rightarrow c$ and $F_{i-1} \wedge c \wedge T \wedge \neg c'$ is UNSAT. Most implementations start from $\neg s$ and attempt to drop literals while keeping relative inductiveness, yielding MIC-style generalization and its extensions. They often leverage UNSAT cores/conflict clauses to guide minimization. Optimizations here can substantially reduce SAT calls and improve lemma quality, but they introduce trade-offs between generalization strength and computational overhead [Su *et al.*, 2024].

Push and propagation (PUSHCLAUSES). To accelerate convergence, IC3 tries to push lemmas forward: if a lemma $c \in F_i$ is also inductive relative to F_i, it can be added to F_{i+1} (equivalently, if $F_i \wedge T \wedge \neg c'$ is UNSAT). The effectiveness of pushing depends on clause ordering, caching, and simplification/subsumption policies, as well as on budgeting decisions that prevent spending too much effort on low-yield push attempts.

3 Methodology

3.1 `IC3-Evolve` Framework Overview

Figure 2 summarizes `IC3-Evolve`, our offline framework for evolving IC3 implementations via iterative, auditable code changes. Starting from a *champion* solver, each iteration selects a constrained modification scope (a set of heuristic slots) and asks a programmer agent to propose a small patch together with a hypothesis about its expected impact and a fallback plan in case the change proves invalid. The candidate is then built and evaluated by an evaluator agent. Crucially, we separate *optimization* from *correctness*: a patch is considered only if it passes proof-gated hard checks with independent witness validation—`SAFE` runs must emit a checkable inductive invariant/certificate, and `UNSAFE` runs must emit a counterexample trace that can be replayed by simulation. Among the validated candidates, we retain improvements based on benchmark performance (e.g., PAR2 and solved counts) and use the evaluator's diagnosis to steer subsequent iterations via the Compass&Jump policy.

3.2 Slot-Restricted Patch Space

To make code evolution controllable and reviewer-auditable, our `IC3-Evolve` exposes a number of *heuristic slots* that align with IC3's dominant runtime subroutines (cf. Section 2.2). Each iteration restricts the programmer agent to one (Compass) or a small set (Jump) of slots, so that proposed patches are localized, interpretable, and easy to revert. **Table 1** summarizes the slots and representative optimization knobs. Importantly, slots constrain *where* we edit, not *what* constitutes correctness: soundness is enforced exclusively by the proof-gated validation described next.

Slot-specific knowledge packs. A slot specifies a curated set of files/functions (and optionally line ranges) that the programmer agent is allowed to modify in a given iteration. For each slot, we maintain a small, auditable knowledge pack that collects the most relevant external and internal references, including solver code excerpts, open-source implementations, and slot-specific papers and documentation. At each iteration, the Compass&Jump search policy retrieves and injects only the slot-relevant subset into the agents' prompts to reduce context noise and constrain edits toward well-understood patterns. These knowledge packs only improve proposal quality and interpretability; correctness is enforced solely by the proof-/witness-gated validation.

3.3 Two-Agent Closed-Loop Evolution

To operationalize heuristic engineering as a repeatable optimization process, `IC3-Evolve` runs a two-agent loop orchestrated by a lightweight Compass&Jump search policy.

Slot	Scope (examples)
PO handling	BLOCKPROOFOBLIGATIONS: worklist order; stale-PO skip; requeue/budget / PO depth; SAT/PO; requeue rate
Ind. gen.	INDGEN: literal dropping; UNSAT cores; CTG/MIC budgets / core size; lemma yield; generalization SAT calls
Pred. gen.	PREDGEN: assumption order; predecessor selection; cooldown / chain length; predecessor SAT calls; depth
Push/prop.	PUSHCLAUSES: push order; caching; subsumption / push success; redundancy; frame stabilization
Cross-slot	Small multi-slot changes and synergies / overall PAR2 and bucketed regressions

Table 1: Heuristic slots for code evolution.

The framework maintains a current *champion* implementation and applies the promotion rule (promote or revert) for each proposed challenger. Validation and benchmarking are executed by the evaluator agent, which also produces a diagnostic *MoveSet*—a ranked list of structured recommendations $m = (\mathsf{slot}, \mathsf{direction}, \mathsf{conf}, \mathsf{risk}, \mathsf{cost})$ summarizing the diagnosis and suggested next edits. Each iteration records full provenance (patch diffs, logs, metrics, and witnesses) to keep the process auditable and to support post-hoc diagnosis.

Iteration protocol. Given a champion solver, one evolution iteration proceeds as follows:

(i) Select scope. The framework uses Compass&Jump to choose the next modification slot(s) and assembles slot-relevant context (knowledge pack excerpts, code snippets, and the previous *MoveSet* if available).

(ii) Propose. The programmer agent produces a slot-restricted patch together with a hypothesis and a fallback plan.

(iii) Validate. The evaluator agent applies the patch, builds the challenger, and runs hard checks including proof-/witness-gated validation.

(iv) Test. The evaluator agent evaluates the challenger on the evolution workload (a fixed HWMCC subset) under fixed timeouts, producing PAR2 and bucketed solved/-timeout statistics.

(v) Diagnose & guide. The evaluator agent inspects the patch diff and evaluation artifacts, confirms (or refutes) the hypothesis, identifies bottlenecks, and returns a ranked MoveSet to guide the next slot and edit direction.

The challenger is promoted only if it passes hard gates and improves the objective without unacceptable regressions; otherwise, the framework rolls back to the champion and continues with the next *MoveSet*.

3.4 Proof-/Witness-Gated Evaluation & Promotion

Proof and witness hard gates. For each benchmark instance that a challenger solves within the timeout, we require

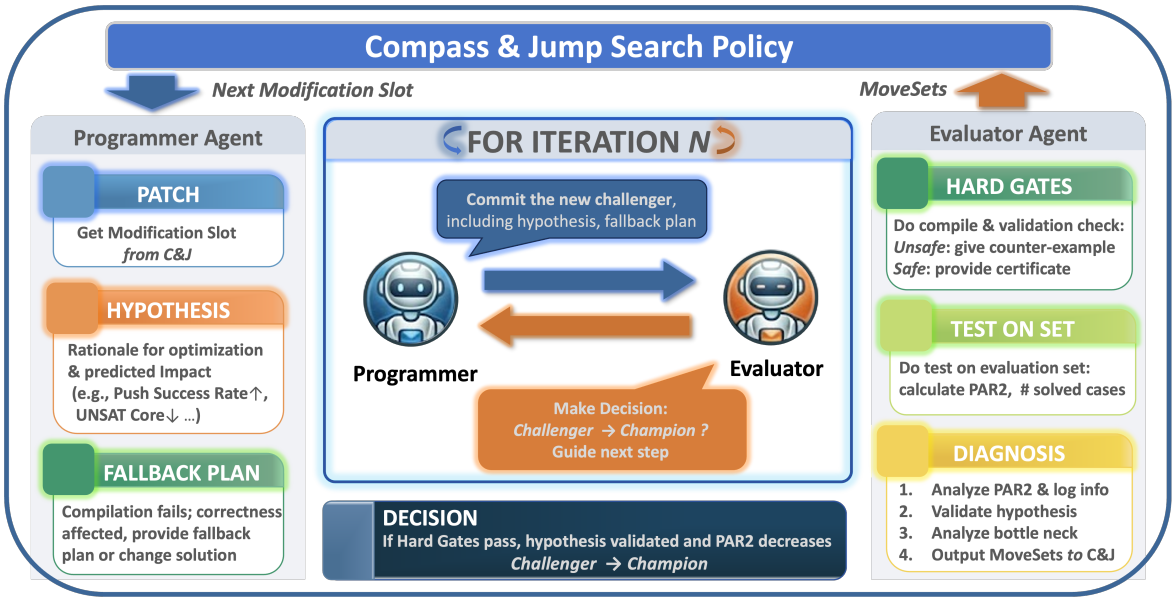

Figure 2: Overview of `IC3-Evolve`. Compass&Jump selects slot(s); a programmer proposes a slot-restricted patch (with hypothesis and fallback). A tool-using evaluator builds and witness-validates the challenger, benchmarks it (e.g., PAR2/solved), and returns a *MoveSet*. Challengers are promoted only if they pass hard gates and improve under a regression budget.

Algorithm 3: Compass & Jump search policy

Input : slots $\mathcal{S}$; MoveSet $\mathcal{M}$; progress history $\mathcal{H}$; jump prob. p_{JUMP}; jump size $J \in \{2, 3\}$
Output: allowed slots $\mathcal{A} \subseteq \mathcal{S}$; guidance moves $\mathcal{G}$; updated p_{JUMP}

if $\mathcal{M} = \emptyset$ **then**
  sample $s \sim \text{UNIFORM}(\mathcal{S})$;
  return $(\{s\}, \emptyset, p_{\text{JUMP}})$;
end
$p_{\text{JUMP}} \leftarrow \text{ADJUSTJUMP}(p_{\text{JUMP}}, \mathcal{H})$; `// decrease if steady, increase if stagnant`
$\mathcal{M} \leftarrow \text{RANKBYSCORE}(\mathcal{M})$; `// using the linear score in Section 3.5`
if $\text{RAND}() < p_{\text{JUMP}}$ **then**
  $\mathcal{G} \leftarrow \text{TOPDISTINCTSLOTS}(\mathcal{M}, J)$;
end
$m^\star \leftarrow \text{SELECTBEST}(\mathcal{M}, \mathcal{H})$; `// optionally risk-averse when volatile`
$\mathcal{G} \leftarrow \{m^\star\}$;
$\mathcal{A} \leftarrow \{\, m.\mathsf{slot} \mid m \in \mathcal{G} \,\}$;
return $(\mathcal{A}, \mathcal{G}, p_{\text{JUMP}})$;

it to emit an artifact that can be checked independently of the evolved solver. If the solver reports `UNSAFE`, it must dump a counterexample trace that is replayed by AIGSIM [Biere, 2014] to confirm that the trace indeed reaches a bad state. If the solver reports `SAFE`, it must dump a proof (inductive invariant) that is validated by CERTIFAIGER [Yu *et al.*, 2021]. Any mismatch (missing proof/witness artifact, replay/check failure, or inconsistent return code) immediately rejects the challenger regardless of its performance.

Promotion and feedback on failures. Only challengers that pass the hard gates are scored on performance metrics such as PAR2 and solved counts. When a challenger is rejected—either by witness validation or by failing to improve under the promotion rule—the evaluator inspects the diff and logs to produce a diagnosis and a ranked *MoveSet*, which guides the programmer agent in the next iteration. Following HWMCC-style certifying model-checking practice [hwm, 2025b], we count a solved instance only when its `SAFE` certificate or `UNSAFE` witness is independently checked; this is instance-level validation under trusted checkers and SAT infrastructure.

3.5 Search Policy: Compass&Jump

Compass&Jump is a lightweight *search policy* for choosing the next editable slot(s) in the slot-restricted patch space. After each iteration, the evaluator returns a *MoveSet* $\mathcal{M}$, i.e., a ranked list of candidate moves $m = (\mathsf{slot}, \mathsf{direction}, \mathsf{conf}, \mathsf{risk}, \mathsf{cost})$. We rank moves by a linear score $\mathsf{score}(m) = w_c \cdot \mathsf{conf}(m) - w_r \cdot \mathsf{risk}(m) - w_k \cdot \mathsf{cost}(m)$ (fixed weights). With probability p_{JUMP} the policy enters *Jump* mode and enables edits to the top $J \in \{2, 3\}$ moves from *distinct* slots, encouraging cross-slot synergies; otherwise it enters *Compass* mode and enables only the single best move (optionally preferring a lower-risk move when progress is volatile). The jump probability p_{JUMP} is adapted from recent progress (decrease under steady improvement; increase under stagnation). Compass&Jump controls only search scope and guidance; correctness is guaranteed solely by the proof-/witness-gated validation in Section 3.4.

4 Experiments

4.1 Experimental Setup

Platform. All experiments are conducted on Ubuntu 20.04.6 LTS, equipped with dual Intel® Xeon® Platinum 8375C CPUs at 2.90 GHz.

LLM model. We use GPT-5-Codex [OpenAI, 2025] as the underlying model for both the programmer and evaluator agents during offline evolution.

Benchmarks. We evolve on a workload of 100 public HWMCC AIGER instances [hwm, 2025b], selected to cover a broad range of difficulty, including trivial, non-trivial, and possibly timeout cases (spanning both `SAFE` and `UNSAFE`). To assess transferability to production settings, we evaluate the final evolved engine on another 100 HWMCC instances (not used during evolution) and 302 industrial verification instances provided by industry.

Evolution configuration. We start from IC3ref [Bradley, 2013], a lightweight textbook-style IC3 implementation with minimal built-in heuristics, which provides a clean and lightweight starting point for controlled evolution. We run 200 offline evolution iterations in total. In the first 100 iterations, we use a structured slot sweep to ensure broad coverage: edits are restricted to one slot at a time, and we move to the next slot after several consecutive non-improving attempts. In the remaining 100 iterations, we enable the Compass&Jump search policy to select slots based on the evaluator's *MoveSet* feedback and to occasionally allow multi-slot edits for exploring cross-slot synergies. We manage context by prioritizing local code/diffs and truncating logs and metrics under fixed budgets.

Timeout and metrics. We use a timeout of 1800 s per instance and report PAR2 (the averaged solving time, where a timeout is penalized as taking $2\times$ the time limit).

4.2 Evolution Trace and Case Study

Figure 3 visualizes a typical offline evolution run. PAR2 improvements arrive as a sequence of punctuated "jumps" in the best-so-far line, interleaved with many non-improving (and thus rejected) challengers. The shaded phases correspond to different exposed heuristic slots, illustrating how the evolution process explores and composes improvements across solver subroutines while keeping correctness guarded by witness validation.

Case study: Push propagation (PUSHCLAUSES). We zoom into rounds r000–r013, where edits are restricted to the PUSHCLAUSES slot (all other code frozen). Across three accepted edits, PAR2 decreases from 1050.61 s to 943.07 s and timeouts drop from 25 to 21. Overall, the edits follow a consistent theme: avoid low-yield cleanup and reallocate effort toward frames with observed progress.

Milestones (green bars in Figure 3). We summarize best-so-far PAR2 (s) and timeouts (TO): **r000** 1050.61/25, **r002** 1050.42/25, **r004** 983.73/22, **r013** 943.07/21.

r002: observe & gate simplification. Simplification inside propagation can be expensive when few clauses are moving. The edit adds lightweight progress observation and triggers simplification only after successful pushing or at periodic checkpoints:

```
if pushedThisRound or
   periodicCheckpoint():
    simplify()
record pushSuccessRate()
```

This reduces low-yield cleanup without changing IC3's proof checks, and sets up later stall-aware scheduling.

r004: skip repeatedly stalled frames. The next edit makes propagation stall-aware by tracking per-frame stall streaks and skipping push attempts on repeatedly unproductive frames:

```
if stallStreak[i] >= limit: continue
pushFrame(i)
updateStallStreak(i)  // reset
                      // else ++
```

By redirecting effort to frames that still show improvement, the solver reduces churn and drops timeouts (25→22) with a clear PAR2 gain.

r013: stall-aware adaptive budgets. Finally, evolution learns an adaptive budgeting rule: allocate more push attempts to frames with recent successes, and cut budgets aggressively for deeply stalled frames with early termination:

```
budget <- adaptBudget(i)
// stall/success history
while clausesRemain(i) and budget>0:
    attemptPush(i)
    budget--
if earlyCut(i): break
```

Compared to the binary skip/try rule, adaptive budgets continuously reallocate work based on observed yield, further improving PAR2 to 943.07 s.

4.3 Comparison with Strong IC3 Baselines

Table 2 compares `IC3-Evolve` against several well-known IC3-based engines on public HWMCC benchmarks and an industrial benchmark suite. We report #Solved (split into #UNSAFE/#SAFE) and PAR2 under the same timeout limit. The baselines include the vanilla IC3ref [Bradley, 2013], IC3ref with a stronger SAT backend (IC3ref-CaDiCaL), ABC [Brayton and Mishchenko, 2010] (whose IC3 engine is a widely-used industrial-strength model checker), and rIC3 as a strong recent IC3 implementation.

A key fairness consideration is that a practical model checker is a pipeline that couples preprocessing, the model checking algorithm, and the SAT backend. In particular, rIC3 [Su *et al.*, 2025a] is the recent HWMCC winner (HWMCC'24 [hwm, 2024] and HWMCC'25 [hwm, 2025a], in the bit-level safety tracks), making it a natural reference point for performance comparison. Since `IC3-Evolve` targets *IC3 heuristic/code evolution* rather than SAT-solver engineering, we control for the SAT backend by uniformly using CaDiCaL [Biere *et al.*, 2024] in our evolved engine and comparing primarily against rIC3-CaDiCaL (rIC3 with CaDiCaL as the backend SAT solver, following the comparison setup in [Su *et al.*, 2025b]). This setup accurately measures improvements attributable to IC3-level implementation choices.

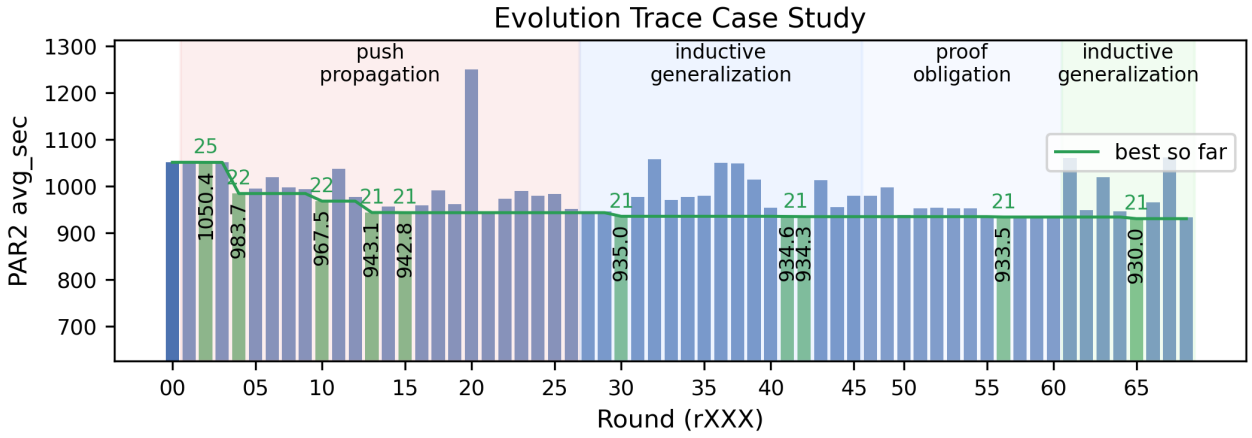

Figure 3: Representative evolution trace on the HWMCC optimization workload. Bars: attempted challengers; line: best-so-far PAR2 (lower is better). Green bars: accepted new best; shading: editable slot(s) per phase.

Solver	HWMCC Set A[a]				HWMCC Set B[a]				Industry Benchmark[d]			
	#S/#T[b]	#U	#Safe	PAR2(s)	#S/#T	#U	#Safe	PAR2(s)	#S/#T	#U	#Safe	PAR2(s)
IC3ref[c]	75/100	21	54	1050.61	75/100	26	49	1068.33	187/302	39	148	1403.31
IC3ref-CaDiCaL	85/100	22	63	718.87	85/100	28	57	732.28	198/302	38	160	1302.13
ABC	86/100	22	64	665.59	88/100	26	62	618.59	185/302	36	149	1417.50
rIC3-CaDiCaL	86/100	21	65	639.61	89/100	27	62	524.35	182/302	36	146	1449.78
`IC3-Evolve`	**92**/100	**24**	**68**	**490.67**	**94**/100	**28**	**66**	**464.56**	**225**/302	36	**189**	**1044.26**

[a] HWMCC Set A is the offline evolution set; HWMCC Set B is unseen and not used in evolution.
[b] #S/#T denotes #Solved/#Total; #U and #Safe denote solved UNSAFE and SAFE instances.
[c] IC3ref serves as the starting point for our offline code evolution.
[d] The industry benchmark is provided by X-EPIC.

Table 2: Baseline comparison on public HWMCC and industrial benchmarks.

Variant	HWMCC Set A (#Solved / PAR2(s))	HWMCC Set B (#Solved / PAR2(s))
Baseline	75 / 1050.61	75 / 1068.33
PO-only	78 / 995.35	75 / 1093.42
PredGen-only	79 / 954.36	78 / 970.38
IndGen-only	78 / 927.80	84 / 803.73
Push-only	76 / 1016.05	75 / 1139.22
Naive-Compose	78 / 944.86	85 / 775.19
`IC3-Evolve`	**92 / 490.67**	**94 / 464.56**

Table 3: Slot-isolated and naive-composition ablation. Single-slot-only variants vs. Naive-Compose vs. full `IC3-Evolve`.

4.4 Ablation Study

No-gate counterfactual (why proof-/witness gating is necessary). In rounds r093–r094, `IC3-Evolve` proposed a new *clause-cleaning* heuristic that would have increased solved instances on the evolution workload by +9, and would be promoted under performance-only selection. However, the challenger failed independent witness validation (certificate checking / trace replay) and was rejected by our proof-/witness gate. This counterfactual shows that witness gating is essential: it prevents unsound "improvements" from becoming champion and propagating into later rounds.

Slot-isolated evolution (IC3 optimization is not a low-hanging fruit). We restrict `IC3-Evolve` to one heuristic slot at a time and also test Naive-Compose, which merges the best single-slot edits without joint tuning. As shown in **Table 3**, these variants improve only modestly and often fail to transfer, indicating that robust gains require coordinated, regression-aware cross-slot evolution.

5 Conclusion and Future Work

We presented `IC3-Evolve`, an offline LLM-driven framework that evolves IC3 heuristics through slot-restricted, auditable edits admitted only by proof-/witness-gated validation, producing a standalone checker with no runtime model dependency. Across public unseen and industrial workloads, `IC3-Evolve` improves over strong baselines, and ablations show that robust gains require coordinated cross-slot evolution with regression-aware promotion.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (grant no. 62304194).

Contribution Statement

Mingkai Miao and Guangyu Hu (∗) contributed equally to this work. Hongce Zhang (†) is the corresponding author.

References

[Biere *et al.*, 2024] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

[Biere, 2014] Armin Biere. Aiger and-inverter-graph library and utilities (incl. aigsim). https://github.com/arminbiere/aiger, 2014.

[Bradley, 2011] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Bradley, 2013] Aaron R. Bradley. IC3ref: Ic3 reference implementation. https://github.com/arbrad/IC3ref, 2013.

[Brayton and Mishchenko, 2010] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[Eén *et al.*, 2011] Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011.

[Hu *et al.*, 2023] Guangyu Hu, Wei Zhang, and Hongce Zhang. Neuropdr: Integrating neural networks in the pdr algorithm for hardware model checking. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, 2023.

[Hu *et al.*, 2024] Guangyu Hu, Jianheng Tang, Changyuan Yu, Wei Zhang, and Hongce Zhang. Deepic3: Guiding ic3 algorithms by graph neural network clause prediction. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 262–268. IEEE, 2024.

[hwm, 2024] HWMCC 2024: Hardware model checking competition 2024 (results). https://hwmcc.github.io/2024/, 2024.

[hwm, 2025a] HWMCC 2025: Hardware model checking competition 2025 (results). https://hwmcc.github.io/2025/, 2025.

[hwm, 2025b] HWMCC: Hardware model checking competition. https://hwmcc.github.io/, 2025.

[Janßen *et al.*, 2024] Christian Janßen, Cedric Richter, and Heike Wehrheim. Can chatgpt support software verification? In *International Conference on Fundamental Approaches to Software Engineering*, pages 266–279. Springer, 2024.

[Le *et al.*, 2021] Nham Le, Xujie Si, and Arie Gurfinkel. Data-driven optimization of inductive generalization. In *FMCAD*, pages 86–95, 2021.

[Novikov *et al.*, 2025] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[OpenAI, 2025] OpenAI. Addendum to gpt-5 system card: Gpt-5-codex, 2025.

[Pirzada *et al.*, 2024] Muhammad AA Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C Cordeiro. Llm-generated invariants for bounded model checking without loop unrolling. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1395–1407, 2024.

[Su *et al.*, 2024] Yuheng Su, Qiusong Yang, and Yiwei Ci. Predicting lemmas in generalization of ic3. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery.

[Su *et al.*, 2025a] Yuheng Su, Qiusong Yang, Yiwei Ci, Tianjun Bu, and Ziyu Huang. The ric3 hardware model checker, 2025.

[Su *et al.*, 2025b] Yuheng Su, Qiusong Yang, Yiwei Ci, Yingcheng Li, Tianjun Bu, and Ziyu Huang. Deeply optimizing the sat solver for the ic3 algorithm, 2025.

[Sun *et al.*, 2024] Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. Autosat: Automatically optimize sat solvers via large language models. *arXiv preprint arXiv:2402.10705*, 2024.

[Wu *et al.*, 2024] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–417, 2024.

[Yu *et al.*, 2021] Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 363–386, Cham, 2021. Springer International Publishing.

[Yu *et al.*, 2025] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous code evolution meets np-completeness. *arXiv preprint arXiv:2509.07367*, 2025.