ReCAP: Boosting Multi-Property Bug Hunting via Similarity-Guided Counterexample Relaying

Jieying Wu[1], Yan Jiang[1], Shenghu Han[1], Jiongming Chen[1], Wenji Fang[2], Xingtong Yang[3], Zhiyao Xie[2], and Min Li[1,*]
[1]*Southeast University*, Nanjing, China
[2]*Hong Kong University of Science and Technology*, Hong Kong, China
[3]*Huawei Technologies Co. Ltd.*, China
[*]Corresponding author: min.li@seu.edu.cn

***Abstract*—Bug hunting plays an important role in hardware formal verification, where a violated property yields a concrete counterexample that exposes a design error. Industrial verification campaigns must expose as many counterexamples as possible across a large property set under a fixed budget. This becomes difficult when violations are reachable only after long execution traces, since a single search can consume much of the budget. Existing multi-property techniques reduce verification cost through joint reasoning or information sharing, but target searches still originate from the initial states and do not directly reuse the concrete reachability progress already captured by counterexamples of other properties. This paper presents textitRelaying Counterexamples Across Properties (RECAP), a similarity-guided framework that reuses a counterexample found for one property to accelerate the search for violations of others. RECAP maintains an online counterexample pool and ranks its traces for each target by property-level similarity. A selected trace is replayed to a concrete reachable state, from which bounded search continues. The replayed prefix and the new continuation are concatenated into a complete counterexample from a legal initial state. Experiments on HWMCC benchmarks and industrial designs demonstrate that RECAP improves multi-property bug hunting efficiency, discovering up to 30 counterexamples compared with 21 by standalone engines on a representative benchmark and substantially accelerating counterexample discovery on challenging industrial properties.**

I. Introduction

Formal property verification (FPV), the most widely deployed form of hardware formal verification, checks large sets of SystemVerilog Assertions (SVA) against a single design under test (DUT) in a functional sign-off flow [1]. Early in such a campaign, the RTL, the properties and the environment constraints are all still evolving, so a large fraction of the checks fail rather than converge to a proof. The objective at this stage is therefore not to close proofs but to expose design errors. Each failing check returns a counterexample that must be diagnosed before the design or the specification is corrected. A counterexample serves this purpose because it exhibits the violation as a concrete execution trace and localizes its cause [2]. A proof carries no such information, and becomes valuable only once the design has stabilized. The main obstacle is depth. When a violation is reachable only after tens of clock cycles, the search consumes a large share of the budget before it yields any feedback.

Both commercial and academic flows provide engines dedicated to bug hunting. Cadence JasperGold [3] and Synopsys VC Formal [4] ship such engines alongside their proof engines, but these remain proprietary, and their search strategies can be neither studied nor extended. Among open-source engines, ABC assembles several Bounded Model Checking (BMC) [5], [6] engines into the abcdeep portfolio [7], and rIC3 [8] combines IC3 [9], also known as Property Directed Reachability (PDR) [10], with step-skipping BMC for deep instances. SimpleCAR [11] searches by complementary approximate reachability, alone or combined with BMC [12]. Despite these differences, every search starts from the initial states, whether properties are checked individually or jointly over the same DUT, and the information shared between them never includes a concrete state that an earlier counterexample reached. Prior multi-property work [13]–[16] lowers the cost of checking many properties through coordination, sharing, or artifact reuse. These methods target overall verification efficiency, and none is designed for *multi-property bug hunting*, where the objective is to expose as many violations as possible under a fixed budget. The closest prior work, SISCO [17], modifies PDR to share invariants of proven properties and to store counterexample traces of falsified ones. A stored trace is used only when the target's own proof obligations reach a state on it, which makes the reuse opportunistic.

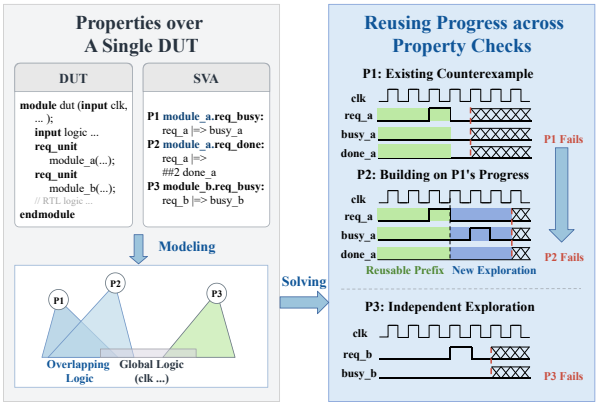


Fig. 1. Motivating example of counterexample relaying. P_2 adopts a prefix of P_1's counterexample and searches only the remaining suffix, whereas P_3 shares only global logic with P_1 and is searched independently.

This gap suggests a different view. A counterexample is also a concrete path into the reachable state space, whose states are legitimate starting points for other properties. Fig. 1 shows when reuse is promising. Properties P_1 and P_2 are

instantiated under the same submodule (`module_a`), so their hierarchical names share a prefix and their logic cones overlap. The counterexample for P_1 therefore carries a *reusable prefix*, and the search for P_2 need only explore the part that differs. Property P_3 sits in another submodule (`module_b`) and shares only global logic such as the clock, so the same prefix offers little benefit. Property relatedness therefore indicates where prefix reuse is likely to help.

We build on this observation with *Relaying Counterexamples Across Properties* (RECAP), an online framework for similarity-guided counterexample relaying. RECAP operates alongside a verification portfolio and consumes the counterexample traces those engines produce. Each discovered counterexample enters a shared pool and becomes a relay source for the properties that remain unsolved. For a given target, RECAP selects promising sources by lightweight property-level similarity, replays the prefix of a selected trace to reproduce a concrete reachable state, and relays that state to the target search. A successful continuation from that state is combined with the prefix, yielding a complete counterexample from a legal initial state.

RECAP thus converts each counterexample from an isolated verification outcome into reachability progress available to the rest of the property set. It modifies no solving algorithm and only adds a relay path around an existing engine portfolio, so it complements rather than replaces the engines a flow already runs. Because the relay path consumes nothing but counterexample traces, it is not confined to assertion checking. Other tasks that report failures as traces, such as X-propagation analysis and sequential equivalence checking, can feed the same pool. The main contributions of this paper are as follows:

- We present RECAP, an online framework that relays concrete reachability progress across properties without modifying the underlying verification engines.
- We propose a similarity-guided selection strategy that ranks relay sources by structural cone-of-influence correlation and hierarchical property-name similarity.
- We develop a relay-based bounded search that extracts reachable states by trace simulation and, on failure, retreats the relay point frame by frame within a bounded budget.

II. Related Work

A. Hardware Model Checking

Hardware model checking decides whether a design satisfies a temporal property, typically a safety property, and returns either a proof or a counterexample. SAT-based checkers [18] differ in how they organize this search. BMC [5], [6] unrolls the transition relation and searches bounded executions, k-induction [19], [20] strengthens bounded reasoning with an inductive step, interpolation [21] over-approximates the reachable states from refutation proofs, and IC3/PDR [9], [10], [22], [23] builds inductive clauses incrementally without explicit unrolling. All of them report a counterexample when a property fails, but only BMC searches for one directly. Industrial checkers such as Cadence JasperGold [3] and Synopsys VC Formal [4] remain proprietary.

B. Bug Hunting

Bug hunting aims to expose deep counterexamples under a limited budget. The abcdeep engine [7] of the HWMCC'17 DEEP track [24] combines several BMC engines with PDR-based reasoning to reach deeper bounds and focuses on individual-property search. ABC's mprove engine [25] provides a multi-property verification flow that progressively reducing unsolved properties. rIC3 [8] advances the bound in strides instead of one frame at a time, and SimpleCAR [11] maintains complementary over- and under-approximations of the reachable states, which combine with BMC [12]. Loop acceleration [26] shortens the unrolling needed to reach a deep state. Commercial portfolios include dedicated engines such as JasperGold's Engine L [3], which targets hard-to-reach counterexamples and cover points. A separate line of work leaves the formal setting, guiding simulation with formal reasoning [27]–[29] or refining an abstract model from spurious counterexamples [30]–[33]. RECAP stays inside the formal flow and reuses concrete counterexamples across properties.

C. Multi-Property Hardware Verification

Prior work exploits the multi-property setting for cross-property reuse. An early approach [34] shares reachable-state information and invariants between property checks, and JA-verification [14] treats other properties as assumptions while reusing IC3 clauses. A second line groups properties by structural affinity over Cone of Influence (COI) overlap and Strongly Connected Components (SCCs), and combines the grouping with semantic partitioning and parallel strategy exploration [13], [35]–[38]. More recent methods move beyond static structure. PURSE [15] orders properties from runtime PDR statistics, and MPBMC [16] clusters them with Graph Neural Network (GNN) embeddings of their COIs. SISCO [17] confines both the shared invariants and the stored counterexample traces to a cluster, and reuses a trace once a proof obligation coincides with one of its cubes.

RECAP differs in both objective and reuse mechanism. It targets multi-property bug hunting, where the goal is to expose as many counterexamples as possible under a limited budget. It accepts counterexamples from any base engine and transfers reachable states before target search, making the reuse proactive rather than opportunistic.

III. Methodology

A. Overview of the RECAP Workflow

Fig. 2 illustrates the overall workflow of RECAP, which consists of three main stages: similarity preprocessing, similarity-guided candidate selection, and counterexample relaying. Given a multi-property verification task over property set Φ, RECAP operates alongside existing verification engines without modifying their solving procedures. In particular, all properties in Φ are modeled under a unified assumption set, so that any counterexample trace collected for one property remains assumption-consistent, and thus legal, when relayed to another. The base engine continuously performs conventional verification and asynchronously publishes discovered counterexamples

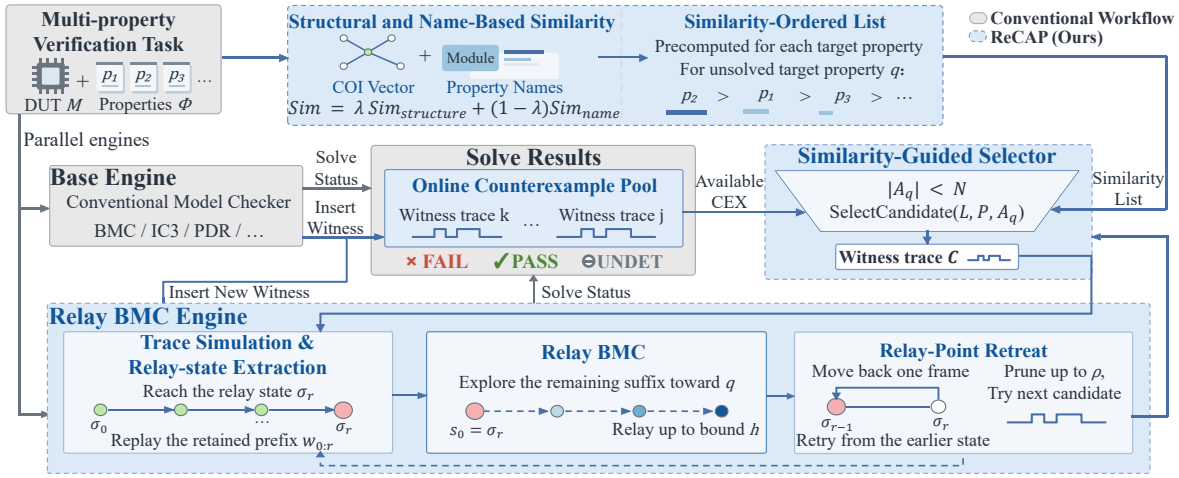


Fig. 2. Overview of RECAP: a base verification engine, an online counterexample pool, a similarity-guided candidate selector, and a relay BMC engine. The base engine populates the pool asynchronously; the selector draws traces from it, and relay BMC simulates each trace to recover a reachable state and continues the bounded search for an unsolved property from there.

to the online pool, which serves as the shared source of reachability progress.

During preprocessing, RECAP computes property-level similarity and builds a ranking of potential source properties for each target property. The similarity-guided selector uses this ranking to select available counterexamples from the online pool, as described in Section III-B. The selected trace is then processed by the relay BMC engine, which extracts a reachable relay state and continues the target search from that state; by the unified assumption modeling above, this relay state is guaranteed to satisfy the target property's assumptions and can be legally used to resume the search. The detailed counterexample-relaying mechanism, including relay-state extraction and frame-by-frame prefix pruning, is presented in Section III-C. Algorithm 1 summarizes the overall RECAP procedure.

B. Similarity-Guided Counterexample Selection

1) Structural and Name-Based Similarity: RECAP ranks candidate source properties according to their likelihood of providing useful reachable prefixes for a target property. The primary signal is structural similarity derived from the overlap of property dependencies. When hierarchical property names are available, an additional name-based similarity is incorporated as a complementary cue.

For each property p_i, RECAP extracts its latch-level COI, which identifies the sequential state elements that may affect the property evaluation. The COI is encoded as a binary vector:

$$c_i = [b_1, b_2, \ldots, b_m], \tag{1}$$

where m is the total number of latches in the design and $b_k = 1$ if the k-th latch belongs to the COI of property p_i, otherwise $b_k = 0$. Therefore, each property is represented by a compact state dependency signature. The structural similarity between two properties is measured by cosine similarity:

$$\mathrm{Sim}_{\mathrm{structure}}(p_i, p_j) = \frac{c_i \cdot c_j}{\|c_i\|_2 \|c_j\|_2}. \tag{2}$$

When hierarchical property names are available, RECAP further incorporates name similarity as a complementary semantic cue. Each property name is represented as a sequence of hierarchy levels, and the similarity between two properties is computed by the weighted agreement of corresponding levels:

$$\mathrm{Sim}_{\mathrm{name}}(p_i, p_j) = \frac{\sum_{k=0}^{D-1} w_k\, s_k(p_i, p_j)}{\sum_{k=0}^{D-1} w_k}, \tag{3}$$

where D is the maximum hierarchy depth, $s_k(p_i, p_j)$ is computed by splitting each level by underscores and comparing corresponding tokens, and $w_k = 1.5^k$ assigns larger weights to deeper levels, which typically indicate more specific module instances.

The two similarity measures are combined as:

$$\begin{aligned}\mathrm{Sim}(p_i, p_j) = \lambda\, &\mathrm{Sim}_{\mathrm{structure}}(p_i, p_j) \\ &+ (1-\lambda)\, \mathrm{Sim}_{\mathrm{name}}(p_i, p_j),\end{aligned} \tag{4}$$

where λ balances the two signals. When property hierarchy information is unavailable, RECAP naturally degenerates to structural similarity only by setting $\lambda = 1$. COMPUTESIMILARITY evaluates Eq. (4) during preprocessing and builds, for every target property q, a similarity-ordered list L over the remaining properties. The pairwise similarity computation has $O(N^2)$ complexity with respect to the number of POs in the design. In practice, the overhead is small because each comparison only operates on compact property-level features and is performed once before verification starts.

2) Online Pool and Candidate Selection: Both the base engine and successful relay searches populate the shared pool $\mathcal{P}$. The operation PUSHWITNESS$(\mathcal{P}, p, \pi)$ adds a newly discovered witness for property p as an indexed entry $C_j = (p_j, \pi_j)$, where j is unique, p_j is the source property, and $\pi_j = \langle i_0, \ldots, i_{d_j} \rangle$ is the witness trace with i_t the primary-input valuation at frame t. Replaying π_j from the legal initial state recorded with it reproduces every state that execution visits.

For each target q, RECAP keeps an attempted set A_q of the pool entries already tried for that target. SELECTCANDIDATE$(L, \mathcal{P}, A_q)$ walks the precomputed ordering and returns the available entry $C_j = (p_j, \pi_j)$ whose source property ranks highest among those with $j \notin A_q$. The index j joins A_q before the relay attempt starts, so no entry is tried twice for the same target.

Algorithm 1: RECAP Procedure

Input: design M, property set Φ, shared online pool $\mathcal{P}$, candidate limit N, relay upper bound h, prune upper bound ρ
Output: relay status map Status, discovered witnesses $\mathcal{W}$

$\mathcal{W} \leftarrow \emptyset$;
// The base engine concurrently pushes witnesses to $\mathcal{P}$
for each *unsolved* $q \in \Phi$ ***in parallel*** **do**
  Status$[q] \leftarrow$ UNDET;
  $A_q \leftarrow \emptyset$;
  while $|A_q| < N$ **do**
    $C_j = (p_j, \pi_j) \leftarrow$ SELECTCANDIDATE$(L, \mathcal{P}, A_q)$;
    if $j = \perp$ **then**
      break;
    $A_q \leftarrow A_q \cup \{j\}$;
    $(res, W_q) \leftarrow$ RELAYWITHPRUNE(M, q, π_j, h, ρ);
    if $res =$ SAT **then**
      Status$[q] \leftarrow$ FAILED;
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{(q, W_q)\}$;
      PUSHWITNESS$(\mathcal{P}, q, W_q)$;
      break;
  // If relay remains unsolved, use the concurrent base engine result when available
  if Status$[q] =$ UNDET **then**
    Status$[q] \leftarrow$ BaseEngineStatus$[q]$;
return $(\text{Status}, \mathcal{W})$;

Each selection reads $\mathcal{P}$ as it stands at that moment, so a witness pushed by the base engine or by another relay becomes eligible from the next selection onward.

C. Counterexample Relaying

Every witness in the pool was produced by exercising the same design M under the same environment constraints, which belong to M rather than to any single property. A trace found for a source property therefore satisfies the constraints that govern an unsolved target property q as well, and the relay state it reaches is legitimate for q. Algorithm 2 states how RELAYWITHPRUNE turns a selected witness trace into a search for a counterexample to the target property.

1) Conventional bounded search: For a target property q in design M and bound k, conventional BMC solves:

$$\text{BMC}(M, q, k) \coloneqq I(s_0) \wedge \bigwedge_{t=0}^{k-1} T(s_t, s_{t+1}) \wedge \Psi_q(s_0, \ldots, s_k), \tag{5}$$

where I is the initial-state predicate, T the transition relation, and Ψ_q the assertion that the target condition of q occurs within the bound. Input variables are implicit in T and Ψ_q; the state comprises the design latches together with any monitor state the property and the environment constraints require. A satisfying assignment yields a counterexample for q and its witness trace. Searched independently, every property starts again from I, and similar long execution prefixes are unrolled symbolically over and over.

Algorithm 2: RelayWithPrune

Input: design M, target property q, relay witness w, relay upper bound h, prune upper bound ρ
Output: counterexample W, result res

$r \leftarrow |w| - 1$;
while $r \geq |w| - \rho$ **do**
  $\sigma_r \leftarrow$ SIMULATE$(w_{0:r})$;
  for $k = 0$ **to** h **do**
    $(res, \tau) \leftarrow \text{RelayBMC}(M, q, \sigma_r, k)$;
    if $res =$ SAT **then**
      $W \leftarrow$ CONCAT$(w_{0:r}, \tau)$;
      return (res, W);
  $r \leftarrow r - 1$;
return $(res, \emptyset)$;

2) Relay state extraction: The witness trace π_j of the entry selected in Section III-B enters RELAYWITHPRUNE as the relay witness w. For a relay frame r, write $w_{0:r}$ for the first r input valuations of w, that is, the prefix that stops just before frame r. The operation SIMULATE$(w_{0:r})$ applies this prefix from the legal initial state recorded with the witness and returns the reachable relay state σ_r; for $r = 0$ the prefix is empty and σ_0 is that initial state itself. As Fig. 2 illustrates, σ_r then serves as the starting state of the target-specific bounded search.

3) Relay BMC: For an unsolved target property q in design M, relay state σ_r, and relay upper bound h, the relay BMC query is formulated as:

$$\text{RelayBMC}(M, q, \sigma_r, h) \coloneqq (s_0 = \sigma_r) \wedge \bigwedge_{t=0}^{h-1} T(s_t, s_{t+1}) \wedge \Psi_q(s_0, \ldots, s_h). \tag{6}$$

The only difference from Eq. (5) is the first conjunct: the initial-state predicate gives way to the concrete reachable state $s_0 = \sigma_r$. The source prefix thus arrives by replay instead of symbolic unrolling, and the solver spends its effort on the suffix that leads to the target property. A satisfiable query returns a continuation trace τ beginning at σ_r, and the operation CONCAT$(w_{0:r}, \tau)$ appends it to the retained prefix to give a complete witness W for q from the recorded legal initial state.

The choice of relay point trades two effects against each other. A deeper point transfers more reachability progress and leaves a shorter suffix to search, but it also commits the continuation to more of the source property's behavior. RECAP therefore starts from the end of the selected witness trace and prunes the retained prefix frame by frame. Whenever the bounded search from σ_r fails, it moves the relay point backward by one frame and retries from the earlier reachable state. This repeats until a counterexample is found or the allocated relay budget is exhausted.

TABLE I
RESULT OF HWMCC BENCHMARKS.

Design	#Prop	#Solved			PAR-2 (s)			Imprv.	
		bmc3	mprove	RECAP (relay)	bmc3	mprove	RECAP	vs.bmc3	vs.mprove
6s123	8	6	4	6 (2)	501.25	901.19	450.93	10.04%	49.96%
6s207	33	21	21	**30 (20)**	654.81	656.00	164.00	74.95%	75.00%
6s250	1402	1358	1358	1358 (540)	95.05	188.04	93.18	1.97%	50.45%
6s258	80	80	80	80 (37)	2.48	12.56	1.96	21.17%	84.43%
6s259	244	16	16	16 (15)	1682.74	1682.08	1682.59	0.01%	-0.03%
6s301	569	540	539	540 (113)	95.37	102.49	97.77	-2.51%	4.61%
6s309	154	23	23	**31 (19)**	1544.78	1552.33	1479.79	4.21%	4.67%
6s344	158	61	61	64 (13)	1112.56	1126.79	1100.14	1.12%	2.37%

IV. EXPERIMENTS

To evaluate RECAP for multi-property bug hunting, we conduct experiments on industrial designs and public benchmarks. The evaluation focuses on counterexample discovery effectiveness and the impact of similarity-guided relay selection.

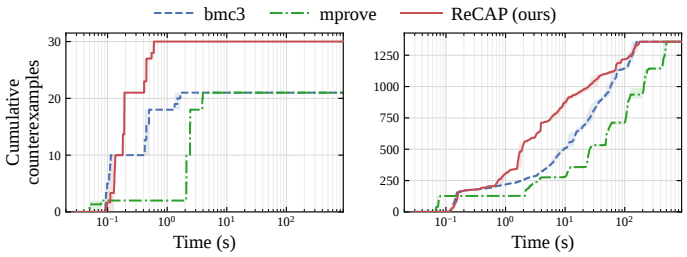
A. Experimental Setup

For public benchmarks, we evaluate RECAP on the multi-property And-Inverter Graph (AIG) designs from HWMCC'13. RECAP is implemented as an extension to ABC, integrating `bmc3` as the base engine thread and an additional Relay BMC engine thread. We fix $h = 35$, $\rho = 5$, and $N = 5$ based on preliminary pilot runs. Since property names are unavailable in AIG inputs, counterexample selection only relies on structural similarity ($\lambda = 1$). The similarity preprocessing overhead is included in the reported runtime. For the largest evaluated HWMCC design (6s250 with 1402 properties), computing all pairwise structural similarities introduces less than 1 second of preprocessing overhead. Experiments are conducted on a server equipped with 256 physical cores (CPU @ 2.70 GHz) and 640 GB memory, with a 900-second wall-clock timeout per instance. Since the implementation of RECAP introduces an additional relay thread while the baseline configurations are single-threaded, wall-clock time alone would mix the benefit of counterexample relaying with additional parallel execution. Therefore, we measure CPU time when computing PAR-2, which is defined as:

$$\text{PAR-2} = \frac{1}{n}\left(\sum_{i\in\text{solved}} t_i + |\text{unsolved}| \cdot 2\tau\right), \qquad (7)$$

where n is the total number of properties, t_i is the CPU time consumed to discover a counterexample for property i, and τ is the timeout limit. Unsolved properties receive a penalty of 2τ. For RECAP, CPU time includes both the master and relay threads. Each instance is repeated three times, and the median solved count and average PAR-2 score are reported. We compare three configurations: `bmc3`, `mprove`, and RECAP. `bmc3` represents a conventional BMC-based bug-hunting engine, while `mprove` is ABC's multi-PO verification flow for jointly handling multiple properties.

For industrial designs, RECAP is integrated into an internal SBY-based [39] portfolio framework, which combines multiple complementary engines, including in-house implementations of ABC's `bmc3` [25], `rIC3` [8], and `superprove` [40], run in parallel with the first returned result adopted. We fix $h = 30$, $\rho = 5$, $N = 5$ and $\lambda = 0.5$ for all experiments based on practical experience from preliminary evaluations. Experiments are conducted on a platform with 16 physical cores (CPU @ 2.40 GHz) and 200 GB memory, using a 24-hour wall-clock timeout to accommodate the complexity of industrial formal verification workloads. We compare three configurations: the original portfolio flow (Baseline Portfolio), Cadence JasperGold (version 2019.09) [3], and the RECAP-enhanced portfolio flow. Following our industrial verification practice, JasperGold is configured with the L, B, Bm, and U engines enabled. JasperGold is included as an industrial reference point rather than a controlled baseline, since it uses a different engine portfolio and internal tuning from our framework.



(a) 6s207 (b) 6s250

Fig. 3. Cumulative counterexample discovery on 6s207 and 6s250 (900 s budget, mean of three runs; shaded bands show min–max).

We evaluate RECAP along three axes: counterexample discovery time, the number of discovered counterexamples, and overall search efficiency (PAR-2), whereas *Imprv.* denotes the percentage reduction in PAR-2 achieved by RECAP relative to the corresponding baseline. The HWMCC benchmarks exercise all three axes over each design's complete property set. On the industrial designs, we report discovery time on representative properties for which relaying is triggered.

B. Bug-Hunting Effectiveness on HWMCC Benchmarks

Table I summarizes the counterexample discovery performance of RECAP on representative HWMCC benchmarks.

Across the eight designs, RECAP achieves a lower, i.e., better, PAR-2 than both `bmc3` and `mprove` on six designs, and its solved-property count never falls below that of `bmc3`. The largest gains appear on 6s207 and 6s250: on 6s207, RECAP discovers 30 counterexamples within the budget, compared with 21 for both standalone engines, and on 6s250, relaying accounts for 540 of the 1358 solved properties. On 6s301 and 6s259, RECAP trails one baseline by a small margin, plausibly because COI-only similarity provides little discrimination among candidate properties on these designs, and because a portion of their properties are PASS properties that neither `bmc3` nor `mprove`, both bug-hunting engines without proof capability, can solve regardless of relaying.

We select 6s207 and 6s250 for the cumulative discovery curves in Fig. 3 because they isolate two distinct regimes of benefit that Table I reports jointly through PAR-2. On 6s207, both baselines plateau at 21 solved properties while RECAP continues to grow, so relaying mainly raises the number of counterexamples exposed within the budget. On 6s250, both baselines already solve nearly all properties, leaving little room

TABLE II
RESULT OF REPRESENTATIVE PROPERTIES ON INDUSTRIAL DESIGNS

Design	#Latch	#AND	#Prop	PropId	PAR-2 (s)			Imprv.
					Baseline Portfolio	Jasper Gold	RECAP	
D1	13710	340017	3	P1	>86400	>86400	779	**99.55%**
				P2	>86400	98	117	**99.93%**
				P3	11078	20	49	**99.56%**
D2	16978	391553	178	P1	4283	4301	3682	14.03%
				P2	3847	104	2536	34.08%
				P3	4361	104	2297	47.33%
				P4	11262	17	1856	**83.52%**
D3	5850	845026	22	P1	30556	29759	7688	74.84%
				P2	8534	58174	2500	70.71%
				P3	38228	9515	3329	**91.29%**
				P4	7882	80724	4290	45.57%
D4	34452	341461	133	P1	>86400	>86400	14203	**91.78%**
				P2	>86400	>86400	15606	**90.97%**
				P3	>86400	>86400	17326	**89.97%**
				P4	>86400	>86400	18129	**89.51%**
				P5	>86400	>86400	18664	**89.20%**
				P6	>86400	>86400	19846	**88.52%**
				P7	>86400	>86400	20299	**88.25%**

for a similar gain in solved count; there, relaying instead front-loads counterexamples that `bmc3` would eventually find on its own, and the benefit shows up as a steeper early rise rather than a higher final count. Together, the two curves separate the exposure gain from the speed gain.

C. Bug-Hunting Effectiveness on Industrial Designs

Table II reports PAR-2 scores for representative properties across four industrial designs, D1–D4. The listed properties are those where RECAP invokes the relay mechanism and are selected to illustrate the effectiveness of counterexample relaying.

RECAP is most effective when a target property's violation requires a long execution prefix that has already been partially explored by a related property. On D1, D2, and D3, RECAP reduces PAR-2 relative to Baseline Portfolio by 99.55%–99.93%, 14.03%–83.52%, and 45.57%–91.29%, respectively. D4 represents a more challenging depth-sensitive scenario: the assertion array contains progressively deeper counterexamples as the index increases. Both Baseline Portfolio and JasperGold time out on all reported properties, whereas RECAP solves them within the budget by reusing states reached by earlier assertions.

Table II also shows that JasperGold achieves lower PAR-2 on several properties in D1 and D2. This is expected because JasperGold applies dedicated industrial bug-hunting strategies optimized for individual properties. In contrast, RECAP focuses on transferring reachability progress across multiple properties and complements existing verification engines rather than replacing them.

D. Ablation Studies

To isolate the two mechanisms of RECAP, we disable the relay mechanism, and separately disable similarity-guided selection. The former is tested on the eight HWMCC designs, while the latter is tested on HWMCC for structural similarity and on one of the industrial cases for both structural and name similarity.

TABLE III
END-TO-END ABLATION OF RECAP'S RELAY AND SIMILARITY-GUIDED SELECTION MECHANISMS ACROSS THE EIGHT HWMCC BENCHMARKS

Configuration	#Solved	PAR-2 (s)	Imprv.
RECAP w/o Relay	2105	431.69	11.85%
RECAP w/o Similarity	2109	461.58	17.56%
RECAP	2125	380.55	–

1) Effectiveness of Counterexample Relaying: To evaluate the contribution of counterexample relaying, we compare RECAP with a variant that disables the counterexample relaying mechanism. As reported in Table III, removing the relay path reduces solved properties from 2125 to 2105 and raises PAR-2 from 380.55 s to 431.69 s, an 11.85% improvement attributable to relaying. This shows that relayed counterexamples yield a substantial end-to-end gain in PAR-2, indicating that the relay mechanism meaningfully improves the overall solving capability.

2) Effectiveness of Similarity-Guided Candidate Selection: To evaluate similarity-guided candidate selection, we compare RECAP with a variant that selects relay sources randomly instead of the similarity-ranked order, keeping the relay engine and budgets unchanged on HWMCC benchmarks. As shown in Table III, disabling similarity guidance reduces solved properties from 2125 to 2109 and raises PAR-2 from 380.55 s to 461.58 s (17.56%). We further evaluate the impact of combined structural and name similarity on the industrial case D4. With 133 properties, disabling similarity guidance reduces solved properties from 129 to 73 and increases PAR-2 from 4846 s to 40809 s (**77.94%**). These results show that the counterexample pool alone is insufficient; similarity-guided selection is critical for identifying effective relay sources, with a particularly significant impact observed on the industrial design.

V. CONCLUSION

This paper presents RECAP, a similarity-guided counterexample relaying framework for multi-property bug hunting. By viewing counterexamples as reusable reachability progress rather than isolated verification results, RECAP transfers concrete states across related properties to accelerate deep counterexample discovery. It combines an online counterexample pool, similarity-guided source selection, and a relay BMC engine with adaptive relay-point retreat, while requiring no modification to existing verification engines. Experiments on HWMCC benchmarks and industrial designs demonstrate that RECAP improves bug-hunting efficiency, showing that existing verification traces are a valuable resource for scalable formal bug hunting.

ACKNOWLEDGMENT

This work is supported in part by by the National Natural Science Foundation of China (Grant No. 62604041) and Huawei Technologies Co. Ltd.

REFERENCES

[1] E. Cerny, S. Dudani, J. Havlicek, D. Korchemny *et al.*, *SVA: the power of assertions in systemVerilog*. Springer, 2015.

[2] R. Dureja, J. Li, G. Pu, M. Y. Vardi, and K. Y. Rozier, "Intersection and rotation of assumption literals boosts bug-finding," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 180–192.

[3] C. D. Systems, "Formal and static verification tools," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, 2026, accessed: 2026-04-02.

[4] Synopsys, "VC Formal: Formal Verification Solution," https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html, accessed: Sep. 14, 2026.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[6] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[7] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017: Results," FMCAD 2017 HWMCC Results Presentation, 2017, describes ABC abcdeep/super_deep and its BMC/PDR portfolio.

[8] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang, "The ric3 hardware model checker," in *International Conference on Computer Aided Verification*. Springer, 2025, pp. 185–199.

[9] Bradley, Aaron R, "Sat-based model checking without unrolling," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2011, pp. 70–87.

[10] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 125–134.

[11] J. Li, R. Dureja, G. Pu, K. Y. Rozier, and M. Y. Vardi, "Simplecar: An efficient bug-finding tool based on approximate reachability," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 37–44.

[12] X. Zhang, S. Xiao, J. Li, G. Pu, and O. Strichman, "Combining bmc and complementary approximate reachability to accelerate bug-finding," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[13] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier, "Boosting verification scalability via structural grouping and semantic partitioning of properties," in *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019, pp. 1–9.

[14] E. Goldberg, M. Güdemann, D. Kroening, and R. Mukherjee, "Efficient verification of multi-property designs (the benefit of wrong assumptions)," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 43–48.

[15] S. Das, A. Hazra, P. Dasgupta, S. Kundu, and H. Jain, "Purse: Property ordering using runtime statistics for efficient multi-property verification," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[16] S. G. Roy, S. Ghosh, A. Banerjee, R. K. Gajavelly, and S. Surendran, "Mpbmc: Multi-property bounded model checking with gnn-guided clustering," *arXiv preprint arXiv:2603.04450*, 2026.

[17] S. Das, A. Hazra, P. Dasgupta, H. Jain, and S. Kundu, "Sisco: Selective invariant sharing, clustering and ordering for effective multi-property formal verification," in *In Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC '25)*. Association for Computing Machinery, 2025, p. 1343–1349.

[18] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[19] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*. Springer, 2000, pp. 127–144.

[20] A. Gurfinkel and A. Ivrii, "K-induction without unrolling," in *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2017, pp. 148–155.

[21] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[22] Bradley, Aaron R., "Understanding IC3," in *Theory and Applications of Satisfiability Testing — SAT 2012*. Springer Berlin Heidelberg, 2012, pp. 1–14.

[23] A. Goel and K. Sakallah, "Empirical evaluation of ic3-based model checking techniques on verilog rtl designs," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 618–621.

[24] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *Proceedings of the 17th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017, p. 9.

[25] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[26] F. Frohn and J. Giesl, "Integrating loop acceleration into bounded model checking," in *International Symposium on Formal Methods*. Springer, 2024, pp. 73–91.

[27] P. K. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii, "The art of semi-formal bug hunting," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[28] C. H. Yang and D. L. Dill, "Validation with guided search of the state space," in *Proceedings of the 35th annual design automation conference*, 1998, pp. 599–604.

[29] B. Kumar, V. Vineesh, P. Nemade, and M. Fujita, "Aries: A semiformal technique for fine-grained bug localization in hardware designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 12, pp. 5709–5721, 2022.

[30] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 737–742.

[31] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[32] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1. IEEE, 2004, pp. 156–161.

[33] N. Amla and K. L. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *International Conference on Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 260–274.

[34] G. Cabodi and S. Nocco, "Optimized model checking of multiple properties," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–4.

[35] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, 2002, pp. 151–165.

[36] G. Cabodi, P. E. Camurati, C. Loiacono, M. Palena, P. Pasini, D. Patti, and S. Quer, "To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 3, pp. 313–325, 2018.

[37] R. Dureja, J. Baumgartner, R. Kanzelman, M. Williams, and K. Y. Rozier, "Accelerating parallel verification via complementary property partitioning and strategy exploration," in *2020 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2020, pp. 16–25.

[38] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, 2010.

[39] YosysHQ, "SymbiYosys (sby): Front-end for Yosys-based formal verification flows," https://github.com/YosysHQ/sby, 2025.

[40] Sterin, "Superprove," https://github.com/sterin/super-prove-build, 2026, accessed: Sep. 2026.