# Path-Sensitive Loop Invariant Inference via Large Language Models and Abstract Interpretation

GUANGSHENG FAN, National University of Defense Technology, China
LIQIAN CHEN*, National University of Defense Technology, China
PEISEN YAO, Zhejiang University, China
BANGHU YIN, National University of Defense Technology, China
WENYU ZHANG, National University of Defense Technology, China
JI WANG*, National University of Defense Technology, China

Loop invariant inference remains a core challenge in program verification, particularly when disjunctive invariants are required. In this paper, we present a path-sensitive loop invariant inference approach based on Large Language Models (LLMs) and abstract interpretation. We apply abstract interpretation to derive initial invariants, which, if insufficient to verify the program, guide an LLM-based agent to generate more precise, path-specific clauses. The core of our method is *Counterexample-Guided Clause Combination* (*CEGCC*), a novel strategy that constructs candidate invariants as disjunctions of clause conjunctions satisfied by counterexamples across loop paths, and iteratively refines them using counterexamples generated by the SMT solver during verification. This shifts the focus of invariant synthesis from blind enumeration to semantic refinement driven by genuine counterexample verification and generation. To further refine this approach, we leverage dependencies across different loop paths, apply abstract interpretation to improve SMT solving, and deploy a second LLM-based agent for counterexample verification and mutation. Results show PAL2Inv matches state-of-the-art LLM methods on linear benchmarks and verifies 53–142% more programs on multi-phase benchmarks, while remaining competitive with specialized symbolic multi-phase verifiers.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Program Verification, Abstract Interpretation, Loop Invariant Inference, Large Language Models.

**ACM Reference Format:**
Guangsheng Fan, Liqian Chen, Peisen Yao, Banghu Yin, Wenyu Zhang, and Ji Wang. 2026. Path-Sensitive Loop Invariant Inference via Large Language Models and Abstract Interpretation. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2026), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

*Liqian Chen and Ji Wang are the corresponding authors.

Authors' Contact Information: Guangsheng Fan, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, guangshengfan@nudt.edu.cn; Liqian Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, lqchen@nudt.edu.cn; Peisen Yao, State Key Laboratory of Blockchain and Data Security, Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Zhejiang University, China, pyaoaa@zju.edu.cn; Banghu Yin, College of Systems Engineering, National University of Defense Technology, China, bhyin@nudt.edu.cn; Wenyu Zhang, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, wenyuzhang08@nudt.edu.cn; Ji Wang, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, wj@nudt.edu.cn.


ACM 1557-7392/2026/1-ART1
https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Loop invariants are central to deductive program verification, serving as inductive assertions that must hold both before and after each loop iteration. Within Hoare logic [32], the soundness of reasoning about loops hinges on the ability to infer such invariants. As such, a wide range of techniques have been developed for invariant inference, including abstract interpretation [13], counterexample-guided abstraction refinement (CEGAR) [10, 11], recurrence-analysis [21, 35, 40], logical abduction [18], syntax-guided synthesis [2], interpolation [37, 44, 49], template-based approaches [8, 62], and learning based approaches [25, 26, 60, 63, 67, 76, 78, 79].

Despite substantial progress, invariant inference remains undecidable in general and practically challenging. The difficulty is amplified in multi-path programs with complex control flow, where conditional branches induce path-sensitive behaviors. Such programs often require disjunctive invariants to capture properties specific to distinct execution paths or states. Moreover, even a single path may exhibit phase-dependent behavior, as in multi-phase programs [55, 64], further complicating the analysis. Accurately verifying these programs requires disjunctive invariants, whose inference is hindered by combinatorial explosion in the candidate space.

Symbolic techniques such as abstract interpretation [13] and constraint-based approaches [61] are grounded in rigorous mathematical foundations, and their extension to support disjunctive invariants often requires careful design [28, 39, 59]. Software model checking algorithms [11, 37, 49], such as interpolation-based CEGAR and trace abstraction [31], are more naturally suited to discovering disjunctive invariants. Besides, data-driven methods [25, 26, 53, 60, 63, 67, 76, 78, 79] follow a *guess-and-check* paradigm, generating candidate invariants and refining them in response to counterexamples. These approaches can, in principle, synthesize disjunctive invariants by, for example, learning decision trees [26] or extending the synthesis grammar to support disjunctions. However, in practice, inferring precise and effective disjunctive invariants remains difficult for existing techniques; for example, they may face scalability challenges due to combinatorial explosion, and the inferred invariants may be too coarse to prove correctness.

Recent work has explored the use of large language models (LLMs) for loop invariant inference [7, 72], leveraging their ability to reason over program structure and semantics. Due to LLM hallucinations, the full invariants generated by LLMs may be invalid; however, some predicates in the answer are still useful for constructing correct invariants. Inspired by this characteristic, LaM4Inv [72] uses bounded model checking [9] to filter invalid LLM-generated predicates and incrementally assemble preserved predicates into conjunctive normal form (CNF) invariants. Furthermore, considering LLMs are not good at generating long correct logical connections, Clause2Inv [7] replaces the traditional *guess-and-check* loop with a *generate-combine-check* strategy: it first prompts the LLM to generate a large set of candidate clauses, i.e., assertions excluding any logical connections, then incrementally combines them into invariants, with the combination strategy adapted based on counterexample type feedback.

Despite their promise, existing LLM-based approaches suffer from two key limitations. First, they lack semantic guidance during clause combination, leading to semantic drift—the inclusion of plausible but irrelevant clauses—and *combinatorial explosion* in the candidate space. For example, Clause2Inv exhaustively builds disjunctions between failed candidates and previously invalidated ones, often producing redundant or vacuous combinations. Second, these approaches depend heavily on the LLM's capacity to synthesize disjunctive structure, a task that remains inherently brittle. LaM4Inv, for example, restricts disjunctions to previously validated predicates, often producing only shallow or overly conservative disjunctive invariants.

The core challenge lies in efficiently navigating the large search space of disjunctive invariants. Almost all existing data-driven and LLM-based approaches get lost in the maze of complex logical

combinations, overlooking that the root cause of disjunction lies in different loop-path executions. Naturally, a loop invariant is the disjunction of invariants of each path. Moreover, though each loop path may have different phases, its invariants can generally be represented by some evidence, such as counterexamples returned during verification on that path.

Motivated by these observations, we introduce PAL2Inv, a path-sensitive loop invariant inference framework that integrates large language model (LLM)-driven clause generation with abstract interpretation and counterexample-guided refinement. PAL2Inv begins with a lightweight abstract interpretation pre-analysis to obtain a coarse initial invariant that accounts for all possible paths, and then employs an enhanced *generate-combine-check* strategy for invariant inference. PAL2Inv improves both the *generate* and *combine* phases to address the above challenges:

- *Structure-Informed Clause Generation*: PAL2Inv decomposes the loop into control-flow paths and uses sound invariants from abstract interpretation, along with path-specific counterexamples, to guide clause generation. An LLM-based agent generates path-specific predicates, thereby reducing hallucination and enhancing relevance.
- *Counterexample-Guided Clause Combination*. PAL2Inv introduces *Counterexample-Guided Clause Combination* (*CEGCC*), a novel strategy that incrementally constructs a disjunctive invariant for each loop path, with each conjunct consisting of clauses satisfied by an inductive counterexample for that path. The initial candidate loop invariant is formed by combining these path-specific invariants and will be refined with counterexamples during verification if necessary.

To accelerate the convergence of the inference process, we adopt three optimizations in *CEGCC*: (i) path dependency analysis using loop unrolling and forward-backward reasoning to generate more genuine counterexamples [10] that can be triggered by some concrete input, (ii) abstract interpretation to reduce spurious counterexamples [10] that cannot actually occur in the program, and (iii) a second LLM agent that verifies counterexample genuineness to stop refinement immediately and mutates spurious examples into variants that are more likely to be genuine.

We conduct experiments on existing linear benchmarks [7, 72] and newly constructed multi-phase benchmarks from [55]. Experimental results show that PAL2Inv achieves performance comparable to the state-of-the-art LLM-based tool, i.e., Clause2Inv [7] and LaM4Inv [72], on linear benchmarks, while verifying 53% to 142% more programs on multi-phase benchmarks. Furthermore, across both benchmarks, PAL2Inv solves the most problems compared to 9 other typical methods, i.e., CPAchecker [5], Ultimate Automizer [30], SeaHorn [29], CVC5 [2], Eldarica [34], LoopInvGen [54], CLN2INV [60], G-CLN [76], and Code2Inv [67], while matching the state-of-the-art multi-phase invariant generation method ImplCheck [55]. Our contributions can be summarised as follows.

- We present a path-sensitive loop invariant inference approach, employing LLMs and abstract interpretation to infer not only the invariants of the loop body but also of each loop path.
- We propose *CEGCC*, a novel strategy that combines clauses returned by LLMs to generate complex disjunctive invariants, enhanced with path-dependence analysis, abstract interpretation, and an LLM-based agent that runs in parallel.
- We implement our approach in the PAL2Inv tool. Experiments indicate that it is comparable to state-of-the-art LLM-based methods on linear benchmarks and achieves significant improvements on multi-phase benchmarks.

## 2 Preliminary

This section introduces the foundational concepts and background necessary to understand our approach, including loop-invariant inference based on Hoare logic [32], the *guess-and-check* framework [25], and the recent LLM-enhanced loop invariant inference method Clause2Inv [7].

### 2.1 Loop Invariant Inference

**Inductive Loop Invariant**. Based on Hoare logic [32], given a loop program `while B do S`, loop invariant inference aims to determine a loop invariant $I$ that satisfies:

$$\frac{P \Rightarrow I \quad \{I \wedge B\}S\{I\} \quad (I \wedge \neg B) \Rightarrow Q}{\{P\} \text{ while } B \text{ do } S\{Q\}} \tag{1}$$

where $P$ is the *pre-condition*, $Q$ is the *post-condition*, $S$ denotes program statements, and $B$ denote the loop condition of the given program. Essentially, there are three conditions that a loop invariant $I$ must satisfy in Eq.(1):

- *reachability*: $I$ is always true when the program first enters the loop.
- *inductiveness*: $I$ is always true during the iterations in the loop.
- *provability*: The meeting of $I$ and the loop exit condition can prove $Q$.

**The Guess-And-Check Approach**. Approaches based on this framework [25] infer the correct loop invariant by iteratively proposing possible invariant candidates, which are then checked by an SMT solver such as Z3 [16] with the following formula:

$$\neg(P \Rightarrow I) \vee \neg(I \wedge B \wedge S \Rightarrow I) \vee \neg(I \wedge \neg B \Rightarrow Q). \tag{2}$$

Corresponding to the conditions in Eq. (1), if candidates are incorrect, the SMT solver will find a formula that satisfies it and return a model (i.e., an assignment to all variables corresponding to a concrete program state, or a pair of states in the case of inductiveness) that makes it true. We call such a model a *counterexample*, as it refutes the candidate's correctness. These counterexamples are classified into three kinds, corresponding to the three disjuncts in Eq. (1) in order [78]:

- positive counterexamples (*PCEs*) from $\neg(P \Rightarrow I)$,
- inductive counterexamples (*ICEs*) from $\neg(I \wedge B \wedge S \Rightarrow I)$,
- negative counterexamples (*NCEs*) from $\neg(I \wedge \neg B \Rightarrow Q)$.

In the next iteration, the *guess-and-check* methods will guess a new candidate invariant *can_I* by utilizing these preserved counterexamples with the following properties:

- $\forall pce \in PCEs$, *pce* violates *reachability*, so *can_I* should be true when assigned to *pce*.
- $\forall nce \in NCEs$, *nce* violates *provability*, so *can_I* should be false when assigned to *nce*.
- $\forall ice \in ICEs$, $ice = (s, s')$ is a pair of states, where $s$ is the pre-state and $s'$ is the post-state of a single execution of the loop body $S$. This pair violates *inductiveness*, meaning *can_I* holds in $s$ but fails in $s'$. Therefore, to correct this, *can_I* must either evaluate to `false` under the assignment $s$, or evaluate to `true` under both assignments $s$ and $s'$.

**Genuine and Spurious Counterexamples**. Following the CEGAR framework [10], we distinguish counterexamples by their feasibility within the concrete program semantics.

- A counterexample is ***genuine*** if it corresponds to a state (or transition) that is concretely reachable during actual program execution.) The correct loop invariant must cover it.
- A counterexample is ***spurious*** if it does *not* correspond to any concretely reachable state (or transition) in the program. It is an artifact of an over-approximation or abstraction. The correct loop invariant must exclude it.

Applied to the three kinds of counterexamples in the *guess-and-check* framework:

- Every *PCE* is genuine, as it corresponds to a state satisfying the precondition $P$, which is trivially concretely reachable.
- Every *NCE* is spurious: since we assume the postcondition $Q$ correctly specifies the intended behavior, any state violating $Q$ cannot be concretely reachable.
- An *ICE* is genuine if its pre-state is concretely reachable; otherwise, it is *spurious*.

### 2.2 LLM-Enhanced Invariant Inference

**The Generate-Combine-Check Approach.** Clause2Inv [7] extends the *guess-and-check* framework by decomposing invariant synthesis into clause generation and clause combination. Rather than generating complete invariants, the LLM is prompted to produce atomic predicates (clauses), which are then systematically combined to form candidate invariants. This design is motivated by the observation that even incorrect invariants often contain useful sub-expressions.

Clause2Inv maintains a clause repository and incrementally constructs larger candidates by combining previously validated candidates and newly generated clauses. Each time the current proposed invariant *cur_I* is not correct, i.e., Eq. (2) with $I$ instantiated as *cur_I* is satisfied, Clause2Inv generates new combination invariants by incrementally constructing longer combinations, guided by counterexample types:

(1) When *cur_I* does not satisfy the *reachability*, it constructs a disjunction of *cur_I* and those preserved *can_I* that were found to be incorrect in the past iterations.
(2) When *cur_I* does not satisfy the *provability*, it constructs conjunctions of *cur_I* and those preserved *can_I*.
(3) When *cur_I* does not satisfy the *inductiveness*, it constructs both conjunctions and disjunctions of *cur_I* and those preserved *can_I*.

This iterative process incrementally builds more expressive invariants while reusing previously validated components, until the correct *cur_I* is found that makes Eq.(2) unsatisfied.

**Limitations of Clause2Inv.** Despite its effectiveness, Clause2Inv has several limitations:

- *Clause Information Loss.* To generate loop invariants, LLMs often distinguish different paths and simulate their execution to reason and reflect. Unfortunately, Clause2Inv overlooks this characteristic and just performs clause generation at the whole-program level, without distinguishing between control-flow paths. As a result, the generated clauses may be overly general or irrelevant to specific paths.
- *Combinatorial Explosion.* Clause combination is performed in a brute-force manner, leading to exponential growth in candidate expressions. The lack of semantic filtering results in redundant or contradictory invariants.
- *Limited Counterexample Integration.* The framework treats all counterexamples uniformly and uses a coarse-grained attention mechanism that shifts focus between syntactic regions (e.g., precondition vs. loop body). This approach fails to exploit the semantic informativeness of counterexamples, reducing their effectiveness in guiding clause refinement.

## 3 Overview

This section presents a motivating example that illustrates the challenges of inferring disjunctive invariants in multi-phase programs. We then describe how PAL2Inv addresses these challenges through a path-sensitive, counterexample-guided synthesis strategy.

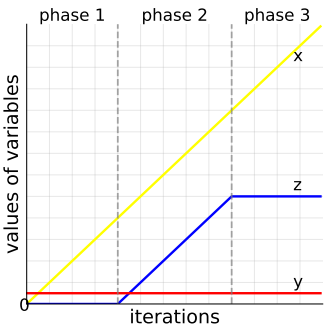
### 3.1 A Motivating Example

Figure 1a presents a C program[1] translated from the corresponding CHC program [23, 57] in the CHC-COMP benchmark suite [58], which is originally from the multi-phase CHC benchmarks [22] used in ImplCheck [55]. The program exhibits nontrivial multi-phase behavior: its control flow and variable updates induce up to five distinct execution phases, depending on the input value of $y$ and the outcome of the conditional at `Line B`. These phases are illustrated in Figs 1b and 1c. It is worth

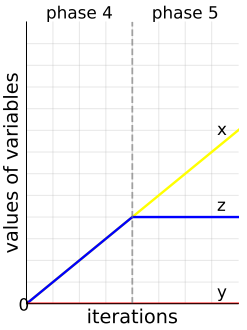
[1]This program is modeled in `QF_LIA` theory, where integer variables are interpreted under standard mathematical semantics with infinite precision, e.g., $x$ can increase unboundedly toward $+\infty$ without wrapping around to negative values. Notably, nearly all existing work in invariant inference is grounded in this theoretical foundation.

```c
int x, y, z;
// pre-conditions
x, z = 0;
assume(y >= 0);
// loop body
while(unknown()){//Line A
  if(y == (x / 1000))//Line B
    z = z + 1;
  x = x + 1;
}
// post-condition
if(x > (1000 * (y + 1)))
  assert(z == 1000);
```

(a) A multi-phase program.




(b) Phases when $y > 0$.




(c) Phases when $y = 0$.

Fig. 1. Motivating example program with phase illustrations.

noting that the program in Figure 1a differs from the motivating C program presented in [55] in that the postcondition's if condition in Figure 1a does not include the constraint $y > 0$ , whereas the version in [55] does. For the program in Figure 1a, we need to precisely characterize the five phases illustrated in Figures 1b and 1c.

In each loop iteration, variable $x$ increments by 1, $y$ remains constant, and $z$ may increment depending on the branch condition at Line B. The execution proceeds through five distinct phases:

- phase1: When $y > 0$ and $x \leq 1000y$, we have $z == 0$.
- phase2: When $y > 0$ and $1000y \leq x \leq 1000(y + 1)$, we have $z == x - 1000y$.
- phase3: When $y > 0$ and $x > 1000(y + 1)$, we have $z == 1000$.
- phase4: When $y = 0$ and $x \leq 1000$, we have $z == x$.
- phase5: When $y = 0$ and $x > 1000$, we have $z == 1000$.

Due to these complex phase situations, the loop invariant that can verify the assertion must be the disjunction of the following invariants of each phase:

- phase1 invariant: $y > 0 \ \wedge \ 0 \leq x \ \wedge \ x \leq 1000y \ \wedge \ z == 0$
- phase2 invariant: $y > 0 \ \wedge \ 1000y \leq x \ \wedge \ x \leq 1000(y + 1) \ \wedge \ z == x - 1000y$
- phase3 invariant: $y > 0 \ \wedge \ 1000(y + 1) \leq x \ \wedge \ z == 1000$
- phase4 invariant: $y == 0 \ \wedge \ 0 \leq x \ \wedge \ x \leq 1000y \ \wedge \ z == x$
- phase5 invariant: $y == 0 \ \wedge \ 1000y \leq x \ \wedge \ z == 1000$

The challenge lies in automatically synthesizing this complex disjunctive invariant.

### 3.2 Why Existing Methods Fail

Unfortunately, almost all existing loop invariant inference methods, including Clause2Inv [7], LaM4Inv [72], CPAchecker [5], Ultimate Automizer [30], SeaHorn [29], CVC5 [2], Eldarica [34], ImplCheck [55], LoopInvGen [54], G-CLN [76], CLN2INV [60] and Code2Inv [67], can not find the correct invariants. ImplCheck computes model-based projections to discover program phases, and then applies data learning to get relationships among variables. However, in [55], only a few specific configurations of ImplCheck can correctly generate the required invariants. Furthermore, in our experiments, ImplCheck fails to generate them successfully. Meanwhile, Eldarica and SeaHorn rely on conventional symbolic methods, which make it hard to find complex disjunctive invariants. Finally, typical machine learning-based approaches, such as LoopInvGen, CLN2INV, G-CLN, and Code2Inv, often only find short disjunctive invariants. As this paper focuses on LLM-based loop invariant inference, we discuss LaM4Inv and Claus2Inv in more detail here.

LaM4Inv filters LLM-generated clauses using bounded model checking and assembles invariants in CNF by increasing length. However, bounded model checking struggles to refute incorrect clauses in this example because it must unroll the loop thousands of times to distinguish between the phases. For example, when $y = 1000$, $z$ only increases after the loop has executed more than 1000000 iterations, an unrolling depth that far exceeds the capacity of current bounded model checking techniques. As a result, it is difficult for LaM4Inv to recognize $z == x-1000y \vee z == 0 \vee z == 1000$ as a part of the final loop invariant. Moreover, because LaM4Inv generates loop invariants exclusively in CNF, its ability to capture disjunctive structures critically depends on the clauses produced by LLMs. Unfortunately, LLMs often struggle to produce sufficiently expressive disjunctive components like the one above, thereby fundamentally limiting LaM4Inv's capacity to infer complex, disjunction-rich loop invariants.

The combination strategy of Clause2Inv is program- and semantics-unaware, as it may combine unrelated or unhelpful clauses to generate incorrect invariants, resulting in a rough approach that ultimately fails to identify the correct complex logical combination. Take $z == 0$ as an example. Since it violates both *inductiveness* and *provability*, Clause2Inv constructs both conjunction and disjunction for this clause and all existing preserved candidates in the list, e.g., $[y > 0 \wedge 0 \leq x \wedge x \leq 1000y,\ y == 0 \wedge 0 \leq x \wedge x \leq 1000y]$, and will get 4 candidate combinations. However, it is only meaningful to combine $z == 0$ with $y > 0 \wedge 0 \leq x \wedge x \leq 1000y$ using conjunction, since there are no other cases that can make $z == 0$ true, as shown in Fig. 1b. As a result, the other 3 combinations are, in fact, useless and may lead to combinatorial explosion during inference. Moreover, when considering making disjunctions, $z == 0$ is expected to be combined with $z == x-1000y \vee z == 1000$, which is useful to construct the correct invariants for the first three phases but indeed difficult to realize within a limited time, not to mention other components of the correct loop invariants.

### 3.3 Our Approach

We present PAL2Inv, which addresses these limitations by combining path-sensitive clause generation with counterexample-guided refinement. The core insight is to decompose the loop into loop-body paths and synthesize path-specific invariants, which are then combined using semantic evidence from counterexamples.

**Path-Sensitive Structure-Informed Clause Generation.** In this paper, as with almost all existing learning-based approaches [7, 25, 26, 53, 60, 63, 67, 72, 76, 78, 79], we focus primarily on loop invariant inference for programs with a single loop. A loop program can be represented by a flowgraph [74]:

*Definition 3.1 (Flowgraph).* The flowgraph of a loop is a tuple $G = (V, E, v_s, v_t, v_e, \iota)$, where $V$ is a set of vertices, $E : V \times V$ is a set of edges that connect the vertices, $v_s$ and $v_t$ are used to capture the start and end points of each loop iteration rather than a concrete loop execution, $v_e$ is a virtual node that represents the exit of the loop, and $\iota$ is a function assigning the transfer function of each edge $e \in E$.

Based on the concept of flowgraph, we can also define the loop path like Xie et al. [74] as follows:

*Definition 3.2 (Loop Path).* A loop path is a finite sequence of control-flow nodes from a virtual entry node $v_s$ to either a virtual back-edge node $v_t$ (an iterative path) or a virtual exit node $v_e$ (an exit path). Each path can be represented by the list of branch conditions it satisfies. We denote the exit path by $path_0$ and each iterative path by $path_i$ for $i > 0$.

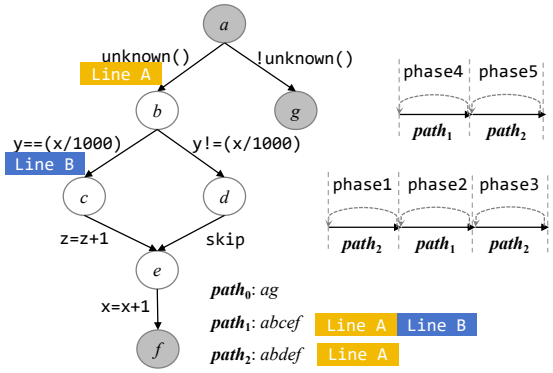



Fig. 2. Flowgraph of the motivating example program.

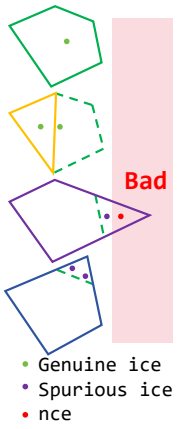



Fig. 3. Example conjuncts under *CEGCC*.

The corresponding flowgraph for the loop program in Figure 1a is shown in Figure 2, where nodes $a$ and $f$ denote the start and end of the loop iteration, respectively, and node $g$ is the exit node. From the flowgraph, we can enumerate all loop paths of the program by a typical depth-first search (DFS) algorithm, and then derive one exit path $path_0$ from node $a$ to $g$, and the other two iterative paths: $path_1$ that goes through $\langle abcef \rangle$, and $path_2$ that goes through $\langle abdef \rangle$.

Moreover, when each branching condition in the loop is annotated with labels like "Line A", the three distinct execution paths can be much more concisely denoted by the list of these labels:

- $path_0$: [], since it is the exit path.
- $path_1$: [`Line A`, `Line B`], since both conditions in these two lines hold.
- $path_2$: [`Line A`], since only the loop condition holds.

Importantly, each iterative path may contain multiple execution phases. For $path_1$, we have phases 2 and 4; for $path_2$, we have phases 1, 3, and 5.

After identifying all these loop paths, we can use this structural information to guide the LLM in generating clauses for each loop path. This contrasts with the approaches of LaM4Inv and Clause2Inv, which generate clauses for the entire program in a single step. Meanwhile, sound over-approximations of loop behavior derived from abstract interpretation provide valuable semantic structure, guiding LLMs to generate more valid candidate invariants. For instance, knowing that $y$ is constant and that $x$ increases monotonically helps the LLM generate phase-relevant clauses such as $x \leq 1000(y+1)$ or $z = x - 1000y$.

**Counterexample-Guided Clause Combination**. After leveraging path-sensitive structure information to guide the LLM in generating clauses for each loop path, we can represent the invariant of the entire loop as the disjunction of the invariants of its individual loop paths. We refer to each such per-path invariant as a path invariant.

*Definition 3.3 (Path Invariant).* For a given loop path, its path invariant is a predicate that holds both before and after the path's execution. For an iterative path, this predicate characterizes the property preserved along any execution from the start node $v_s$ to the end node $v_t$. For an exit path, the path invariant typically corresponds to the conjunction of the loop's preconditions and the negation of its loop condition.

This definition differs from the one in [4], which captures the invariants of so-called *path programs* rather than the loop paths we define in this paper. The invariants for exit paths can be

derived via static analysis, whereas those for iterative paths are generated by our newly designed clause combination strategy *CEGCC*.

For each iterative loop path, instead of blindly combining all generated clauses as in Clause2Inv, the *CEGCC* strategy combines clauses in a manner that is aware of program semantics. The key insight is that counterexamples returned by SMT solvers provide valuable evidence for which clauses to combine. As shown in Figure 3, each inductive counterexample state can be used to construct a conjunct by conjoining all the clauses it satisfies. The disjunction of all these conjuncts, each derived from a distinct inductive counterexample, forms an over-approximation of the correct path invariant for the corresponding loop path.

When a path contains only genuine counterexamples, the corresponding path invariant can be derived from these genuine counterexamples precisely once, given all necessary clauses; e.g., we only get the green polygon in Figure 3. For the motivating example program in Figure 1a, we can get conjuncts of each correct phase invariant when deriving the following genuine inductive counterexamples of program variables $(x, y, z)$:

- `phase1` invariant can be obtained by: $\mathbf{(}(0, 2, 0), (1, 2, 0)\mathbf{)}$ in $path_2$.
- `phase2` invariant can be obtained by: $\mathbf{(}(2000, 2, 0), (2001, 2, 1)\mathbf{)}$ in $path_1$.
- `phase3` invariant can be obtained by: $\mathbf{(}(3000, 2, 1000), (3001, 2, 1000)\mathbf{)}$ in $path_2$.
- `phase4` invariant can be obtained by: $\mathbf{(}(0, 0, 0), (1, 0, 1)\mathbf{)}$ in $path_1$.
- `phase5` invariant can be obtained by: $\mathbf{(}(1000, 0, 1000), (1001, 0, 1000)\mathbf{)}$ in $path_2$.

In this case, the path invariant of $path_1$ is the disjunction of variants of `phase2` and `phase4`, while the path invariant of $path_2$ is the disjunction of variants of `phase1`, `phase3`, and `phase5`.

Unfortunately, each loop path may contain not only genuine but also spurious inductive counterexamples, yielding conjuncts that either use only a subset of the correct clauses (e.g., the purple and blue polygons in Figure 3) or are outright incorrect (e.g., the yellow polygon in Figure 3). However, though the initial candidate loop invariant we derive in these situations is not valid, we can still leverage the counterexamples returned during verification to refine it:

- For the yellow polygon, since it is indeed an under-approximation of the true invariant and violates the inductiveness, the SMT solver will return another certainly genuine counterexample, which will, in the next iteration of *CEGCC*, induce the larger green polygon.
- For the purple polygon, since it violates the post-condition, the SMT solver will return a negative counterexample, by which we can remove this incorrect conjunct from the candidate path invariant.
- For the blue polygon, if it is indeed an over-approximation of the true invariant and violates the inductiveness, the SMT solver will return another spurious inductive counterexample, which we can use to remove this incorrect conjunct from the candidate path invariant.

For the motivating example program, in addition to the genuine counterexamples above in each path, we may also encounter a spurious counterexample $\mathbf{(}(2000, 2, 1000), (2001, 2, 1001)\mathbf{)}$ in $path_1$. It induces a conjunct $y > 0 \land 1000y \leq x \land x \leq 1000(y + 1)$, which is not precise enough and similar to the blue polygon in Figure 3. In this case, the initial candidate loop invariant is the disjunction of the correct loop invariant and this conjunct. When checking inductiveness, the SMT solver may return a counterexample $\mathbf{(}(3000, 2, 2000), (3001, 2, 2000)\mathbf{)}$, of which the first state satisfies the induced conjunct. As a result, we can use this newly generated counterexample to greedily remove the induced, insufficiently precise conjunct and retain only the correct loop invariant. On the other hand, we can also use the LLM to determine whether the state is spurious. If the LLM identifies it as spurious, we remove its corresponding conjunct from $path_1$ and retain only the correct loop invariant.

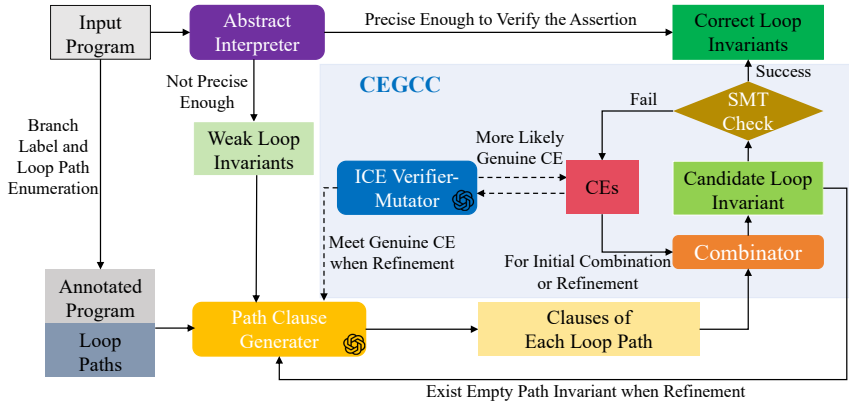



Fig. 4. An overview of PAL2Inv.

## 4 Methodology

In this section, we present PAL2Inv, a path-sensitive loop invariant inference approach based on LLMs and abstract interpretation. PAL2Inv runs in *rounds*. In each round, it (i) asks an LLM agent to generate *many* candidate clauses for each loop path, (ii) performs a CEGCC combination session that builds an initial candidate invariant and then refines it using counterexamples, and (iii) if the session fails to find a correct invariant, it starts the next round and combines the newly generated clauses with the previously retained ones.

### 4.1 Overall Workflow

Figure 4 illustrates the architecture of PAL2Inv, which follows a *generate-combine-check* paradigm and consists of four primary components: (1) an abstract interpreter, (2) a path clause generation (PCG) agent powered by an LLM, (3) a counterexample verification and mutation (V&M) agent, and (4) a clause combinator. Alg. 1 summarizes the full outer loop.

- **Pre-Analysis.** For the input C program (usually containing only a single loop, as considered in almost all existing learning-based methods [7, 25, 26, 53, 60, 63, 67, 72, 76, 78, 79]), the abstract interpreter computes sound (but possibly imprecise) loop invariants `Inv_AI`. We also annotate each condition statement of the input program and then enumerate loop paths `paths`. The exit-path invariant `exit_Inv` is determined by straightforward static analysis.
- **Clause Generation (Per Round).** The PCG agent generates a *batch* of clauses for each iterative path. We store and *retain* all generated clauses in a per-path repository, `PCLDict`; in later rounds, newly generated clauses are added to this repository rather than replacing it.
- **CEGCC Combination (Per Round).** Using the current clause repository and the preserved counterexamples (positive, negative, and inductive counterexamples as defined in §2), we first construct an initial candidate invariant via `InitCombine` (§4.3.1), and then refine it using counterexamples returned by SMT checking (§4.3.2). If the session fails (e.g., any loop path becomes associated with an empty invariant), we start the next round by generating additional clauses, which are then combined with the retained ones.

In short, PAL2Inv repeats: *generate many clauses* → *init-combine* → *refine* using counterexamples. If a CEGCC session cannot reach a correct invariant with the current clause repository, PAL2Inv launches another round and expands the repository.

Refinement proceeds in one of two modes. In the greedy mode, i.e., the `GreedyRefine` algorithm (detailed in §4.3.2), refinement halts when any loop path becomes associated with an empty

**Algorithm 1:** The PAL2Inv Algorithm

**Input :** A program `prg`, sound invariants returned by abstract interpreter `Inv_AI`, exit path invariant `exit_Inv`, loop path enumeration `paths`.
**Output :** Loop invariant `Inv`.

```
Function PAL2Inv(prg, Inv_AI, exit_Inv, paths):
    Inv ← ⊥; CEs ← ∅; refine_CEs ← ∅;
    foreach path_i, i > 0 do
        PCLDict[path_i] ← ∅
    end
    while Inv = ⊥ do
        newClauses ← LLMClauseGenerate(prg, Inv_AI, refineCEs, paths);
        foreach path_i, i > 0 do
            PCLDict[path_i].update(newClauses[path_i])
        end
        Inv, path_conjs, CEs ← InitCombine(prg, PCLDict, CEs, exit_Inv);
        Inv, refineCEs ← GreedyRefine(Inv, path_conjs); ▷or assisted by V&M agent
        CEs.update(refineCEs);
    end
    return Inv;
```

invariant, and PAL2Inv starts the next outer-loop round to generate more clauses. Alternatively, the V&M agent (detailed in §4.4) can be used to determine whether a counterexample is genuine. If so, refinement terminates immediately, and PAL2Inv starts the next round. The V&M agent can also mutate spurious counterexamples into more plausible ones, i.e., by leveraging the LLM's semantic understanding to steer variable assignments toward feasible traces, to guide subsequent clause generation and refinement.

### 4.2 Path-Sensitive Structure-Informed Clause Generation

LLMs are prone to generating invalid full invariants due to hallucination, yet prior work [7, 72] has shown that useful clauses often appear within these invalid outputs. However, these clause generation methods do not fully leverage LLMs' program comprehension ability, as they prompt LLMs to generate clauses [7], i.e., assertions without logical connections, or entire program invariants [72]. In fact, we observe that LLMs can often recognize different loop execution paths and then propose invariants based on the reasoning and reflection on these paths. Furthermore, each generated clause can be related to a corresponding execution path.

Based on the above observations, we employ an LLM-based PCG agent to generate clauses in a path-sensitive manner. The agent is prompted, as shown in Prompt 1, to simulate execution of the given program, infer input conditions for each feasible loop path, and finally generate precise assertions for those paths in the specified JSON format. There are mainly 4 elements we should include in the prompt when inferring the loop invariants for each original C program.

**Annotated Program.** Each condition branch statement of the program is automatically instrumented and annotated with a label of the form "Line *Label*", where *Label* is a capital letter starting from "A" assigned in ascending order according to the source line numbers of the branch statements. The instrumentation is necessary to represent loop paths.

**Prompt 1: Clause Generation Prompt.**

**### Program ###**
{program}
**### Paths ###**
The following are all possible loop paths of the above program:
{paths}
Here, path 0 refers to the path that does not meet the loop condition. Other paths are indicated by the branch conditions they satisfy, which are denoted by their respective "Line" indices.
**### Task ###**
I will provide you with a program. First, simulate the program execution and determine how each program path can be reached from the program inputs. Then, you need to generate as many valid assertions as possible for each feasible path. Finally, you should answer a list of path invariant entries. Each entry contains two fields: a path name and a list containing all possible assertions for the path. Your answer must be in the specified JSON format without any explanations or thought processes.
**### Notes ###**
1. Each assertion is a separate clause, excluding "&&", "and", "||", "or", "==>", "->", "=>", "implies", "if", and "unknown".
2. Limit operators in each assertion to "==", "!=", "<", ">", "<=" and ">=".

**### Useful Information ###**
The following is the loop invariants generated by Clam, an abstract interpretation-based program analysis tool. You need to generate more precise invariants based on them:
{Clam Inv}
**### SMT Check Feedback ###**
Here are some counterexamples returned by Z3. You need to simulate the program execution and answer path invariants following the given advice.
- Positive Counterexamples you need to satisfy: {pce}
Advice: Pay attention to the pre-condition of the given program.
- Negative Counterexamples you need to exclude: {nce}
Advice: Pay attention to the post-condition of the given program.
- Inductive Counterexamples: {ice}
Advice: Pay attention to the inductiveness of the given program. For each inductive counterexample, determine whether some inputs can execute to its pre-state.
1. If No, provide more precise assertions that exclude the pre-state of the counterexample.
2. If Yes, provide additional assertions to include the post-state of the counterexample, while ensuring that the invariant remains inductive.
**### RESPONSE FORMAT ###**
{JSON format}
**### Your Response ###**

**Loop Paths.** Path information is crucial for precise path-sensitive clause generation, as it is a form of static analysis information that can guide LLMs in better understanding the given program. Each loop path is represented by a list of branch statement labels "Line *Label*" that it satisfies. It is worth noting that Clang-fe [68], a tool widely adopted in existing machine learning–based loop invariant inference approaches, employs a standard depth-first search (DFS) algorithm to enumerate all loop

paths when translating C programs into SMT formulas. To maintain consistency with the path index of each inductive counterexample discussed in § 4.3, we adopt the same DFS algorithm to enumerate loop paths. Since the loop path notation does not correspond to concrete execution paths, and our benchmark programs do not exhibit highly complex control flow with an excessive number of loop paths, the DFS algorithm we use does not encounter path explosion.

**Initial Loop Invariants Derived by Abstract Interpretation.** We use the abstract interpretation–based program analysis tool Clam [52] to automatically generate initial loop invariants for the given C program. By selecting different abstract domains, such as the polyhedron domain [15], the octagon domain [50], and the interval domain [14], Clam produces invariants with varying levels of precision, all of which are sound over-approximations of the true loop behaviors along every loop path. We then prompt large language models to refine these initial invariants by suggesting more precise clauses. In this workflow, the LLM effectively assumes a role analogous to abductive inference [18], augmenting formally guaranteed but potentially coarse invariants with human-like intuition or pattern recognition. Compared to methods that rely on simpler template-based or data-driven guesses, using abstract interpretation provides reliable initial invariants quickly, thereby enhancing the LLM's semantic understanding of the program.

**Feedback of SMT Check.** We provide the LLM with the counterexamples returned by the SMT solver during each verification attempt of candidate invariants in the previous *CEGCC* iteration. As illustrated in the related work [7, 72], these counterexamples are also important for generating more useful clauses. Note that, because of the refinement step in *CEGCC*, there may be multiple kinds of counterexamples in the prompt, rather than only one in the prompts of [7, 72]. For positive, negative, and inductive counterexamples, we require the LLM to attend to the program's pre-condition, post-condition, and inductive step, respectively. Moreover, for each inductive counterexample $i = (s, s')$, we prompt the LLM to determine whether the pre-state $s$ is concretely reachable, and to provide clauses based on the following cases: if $s$ is unreachable, provide more precise assertions that exclude $s$; otherwise, provide additional assertions to include the post-state $s'$ while maintaining inductiveness.

### 4.3 Path-Sensitive Clause Combination via *CEGCC*

We combine the clauses generated by the PCG agent in the *Counterexample-Guided Clause Combination* (*CEGCC*) strategy. The core idea of *CEGCC* is to construct loop invariants by combining clauses in a path-sensitive manner, guided by counterexamples from verification attempts. The process works in two phases: (1) constructing an initial candidate invariant by combining clauses that are satisfied by inductive counterexamples on each path, and (2) iteratively refining this candidate using counterexamples returned during verification.

*4.3.1 Initial Combination.* Our approach uses clauses generated along each iterative path and the preserved counterexamples to first construct an initial combination `can_Inv_0`, as shown in Alg. 2. Since our candidate invariants will be refined in decreasing order of length, the initial combination is expected to be at least an under-approximation of the correct loop invariants and should not include too many invalid invariants. We first prepare (Lines 2-5) and then incrementally construct the initial candidate invariant (Lines 6-21).

In the preparation, we first assign `can_Inv_0` to the invariant of the exit path (Line 2), which can be represented as the conjunction of the *pre-condition* and the negation of the loop condition, i.e., $P \wedge \neg B$ and determined by a simple static analysis. Then, we need to prepare the clauses used for combination, which is realized by the `selectClauses` function in Line 3. In each path, it selects either all clauses or only the preferred clauses, using the same probabilities to balance quality and length in the final combination. The preferred clauses are those that pass the inductiveness

**Algorithm 2:** Initial Combination Algorithm

**Input :** A program prg, *iterative path*-clauses dictionary PCLDict, counterexample set CEs, exit path invariant exit_Inv.

**Output :** Initial candidate invariant can_Inv_0, conjuncts of each path path_conjs, counterexample set CEs.

```
1  Function InitCombine(prg, PCLDict, CEs, exit_Inv):
2      can_Inv_0 ← exit_inv; ;                ▷First assigned to the invariant of exit path
3      PCLDict_prefer, CEs_prefer ← selectClauses(PCLDict);
4      CEs ← CEs ∪ CEs_prefer;
5      PathICEsDict,PCEs,NCEs ← sort(CEs);
6      foreach path, Clauses in PCLDict_prefer.items() do
7          path_Inv_i ← ⊥;
8          foreach ice in PathICEsDict[path] do
9              ice_Inv ← ⊤;
10             foreach Clause in Clauses do
11                 if ice.first() ⊨_T Clause and ice.second() ⊨_T Clause then
12                     ice_Inv ← ice_Inv ∧ Clause;;     ▷Compose the respective conjunct
13                 end
14             end
15             if ∀ nce ∈ NCEs, nce ⊭ ice_Inv then
16                 path_Inv_i ← path_Inv_i ∨ ice_Inv; ;          ▷path_Inv_i is in DNF
17                 path_conjs[path].add(ice_Inv);
18             end
19         end
20         can_Inv_0 ← can_Inv_0 ∨ PathI_i;
21     end
22     return can_Inv_0, path_conjs, CEs;
```

check by an SMT solver or have corresponding counterexamples on a different path. We do not always combine only preferred clauses, as a non-preferred clause may still be useful in forming the final correct loop invariant. By the way, the inductive counterexamples CEs_prefer produced during the preference checking step are also useful for the following combination, especially during the first invocation of Alg. 2, where the initial candidate invariant is constructed without prior verification feedback.

Next, we need to sort all counterexamples into *positive counterexamples* PCEs, *negative counterexamples* NCEs, and *inductive counterexamples* for each path, and record them in a dictionary PathICEsDict (Line 5). Note that, for inductiveness checking, the loop body formula [68] that is often used in existing machine learning-based loop invariant inference methods is in the format:

$$(or \;\; (pathf_0) \;\; (pathf_1) \;\; \ldots \;\; (pathf_n))$$

where $\forall pathf_i, 0 <= i <= n$ records all statements of $path_i$ in CNF, and $n + 1$ is the counts of paths.

To determine not only the assignments of the returned counterexample but also which path it belongs to after checking, we optimize this widely used format by instrumenting each boolean

path flag $flag_i$. After that, the loop body formula is updated to:

$$
\begin{aligned}
(and\ \ & (or\ \ (and\ \ flag_0\ \ pathf_0)\ \ (and\ \ flag_1\ \ pathf_1)\ \ \ldots\ \ (and\ \ flag_n\ \ pathf_n)) \\
& (not\ \ (and\ \ flag_0\ \ flag_1))\,(not\ \ (and\ \ flag_0\ \ flag_2))\ \ldots\ (not\ \ (and\ \ flag_0\ \ flag_n)) \\
& (not\ \ (and\ \ flag_1\ \ flag_2))\,(not\ \ (and\ \ flag_1\ \ flag_3))\ \ldots\ (not\ \ (and\ \ flag_1\ \ flag_n)) \\
& \vdots \\
& (not\ \ (and\ \ flag_{n-1}\ \ flag_n)))
\end{aligned}
$$

where the gray-shaded component indicates that whenever $path_i$ is executed, the corresponding flag $flag_i$ must be true, while the additional constraints ensure that at most one flag is true at any time, reflecting that each inductive counterexample corresponds to the execution of exactly one loop path.

After the above preparations, we incrementally build disjunctions of `can_Inv_0` with all candidate path invariants (Line 20). Each of them is determined (Lines 7-19) in Disjunctive Normal Form (DNF), where each conjunct corresponds to an inductive counterexample `ice` $= (s, s')$ on this path. The conjunct, denoted as `ice_Inv`, is the conjunction of clauses satisfied by both the pre-state $s$ and the post-state $s'$ (Lines 9-12). Note that we retain only those conjuncts that cannot be satisfied by preserved *negative counterexamples* `NCEs` (Lines 15-18). This is because the final discovered `can_Inv_0` must be the over-approximation of each `ice_Inv`, thanks to the property of DNF. If `nce` $\models_{\mathcal{T}}$ `ice_Inv`, then we will also have `nce` $\models_{\mathcal{T}}$ `can_Inv_0`, which should be prevented. Regarding the background theory $\mathcal{T}$, we currently consider only `QF_LIA`. During the combination, we also store all retained conjuncts of each path in a dictionary `path_conjs` (Line 17), which is useful for the next refinement step.

*Example 4.1.* For $path_1$ in Figure 1a, suppose all necessary clauses have been generated by the LLM, and consider only constructing the invariants of `phase2` here. For variables $(x, y, z)$, suppose `NCEs`$=\{(1000, 1, 1000)\}$, and `ICEs`$=\{\mathbf{(}(2000, 2, 0), (2001, 2, 1)\mathbf{)}, \mathbf{(}(2000, 2, 2000), (2001, 2, 2001)\mathbf{)}\}$. With the first $ice \in$ `ICEs`, we will derive the first conjunct $y > 0 \wedge 1000y \leq x \wedge x \leq 1000(y+1) \wedge z == x - 1000y$ by conjoining all clauses that the first $ice$ satisfies. Then, with the second $ice$, we first combine $y > 0 \wedge 1000y \leq x \wedge x \leq 1000(y+1) \wedge z == x$, but filter it later by `NCEs`, since the negative counterexample models this conjunct. Finally, we only retain the first conjunct.

*4.3.2 Counterexample-Guided Combination Refinement.* When the current guessed invariant does not satisfy the loop invariant conditions, the SMT solver will return counterexamples, which can then be used to feed the combinator to refine the current invariant to get a new candidate invariant that is more likely to be correct. We propose two refinement algorithms: the first is the greedy refinement algorithm (Alg. 3), and the second builds upon it and employs LLMs for optimization, which will be described in §4.4.

As shown in Alg. 3, given the current candidate invariant `can_I` and the dictionary `path_conjs` used to record the conjuncts that make up `can_I` in each iterative path, this greedy refinement algorithm employs a refinement iteration to obtain the final candidate invariant. It also preserves the counterexamples `refineCEs` returned during verification, which are then used to construct the prompt for the PCG agent, as detailed in §4.2.

In each refinement iteration, we first check whether any iterative path lacks conjuncts, indicating an empty path invariant (Lines 3-5). We assume that all iterative paths are feasible, i.e., every path corresponds to at least one executable branch in the loop body; paths that are syntactically present but semantically unreachable (e.g., `if (false)` branches) are excluded from consideration. Under this assumption, an empty path invariant indicates that the candidate invariant `can_I` cannot be

**Algorithm 3:** Greedy Refinement Algorithm

**Input :** candidate invariant `can_I`, conjuncts of each iterative path `path_conjs`.
**Output :** Invariant after refinement `can_I`, new counterexample set `refineCEs`.

```
1  Function GreedyRefine(can_I, path_conjs):
2      while true do
3          if ∃conjs ∈ path_conjs.values(), conjs = ⊥ then
4              return ⊥, refineCEs ;                          ▷Exist empty path invariant
5          end
6          ce, unknown ← SMT-solver.verify(can_I);
7          if unknown = True then
8              return ⊥, refineCEs ;                          ▷SMT check returns unknown
9          end
10         if ce = None then
11             return can_I, refineCEs ;                      ▷Find a correct loop invariant
12         end
13         refineCEs.add(ce);
14         if ce is positive counterexample then
15             return ⊥, refineCEs ;                          ▷Already violate reachability condition
16         end
17         if ce is negative counterexample then
18             FilterPathConjuncts(path_conjs, ce) ;          ▷Filter those violate provability
19         end
20         if ce is inductive counterexample then
21             FilterPathConjuncts(path_conjs, ce.first()) ;  ▷Consider ce as spurious
22         end
23     end
24 Function FilterPathConjucts(path_conjs, m, can_I):
25     foreach path, conjs in path_conjs.items() do
26         if m ⊨_T ⋁ conj, conj ∈ conjs then
27             foreach conj in conjs do
28                 if m ⊨_T conj then
29                     path_conjs[path].remove(conj)
30                 end
31             end
32         end
33     end
34     can_I ← ⋁ conj, conj ∈ ⋃(path_conjs.values())
```

correct, and the refinement iteration immediately terminates (Line 4). Otherwise, we then verify whether `can_I` is correct by an SMT solver (Line 6). Once the SMT solver returns `unknown` (Line 7), we have to stop the refinement iteration and start the next outer iteration of PAL2Inv. On the other hand, if the verification succeeds (Line 10), we have fortunately discovered the correct loop invariants as `can_I`, and the refinement concludes. Finally, if the check fails (Lines 13-21), the solver returns counterexamples that guide subsequent refinement steps, with the refinement strategy determined by the type of counterexample produced.

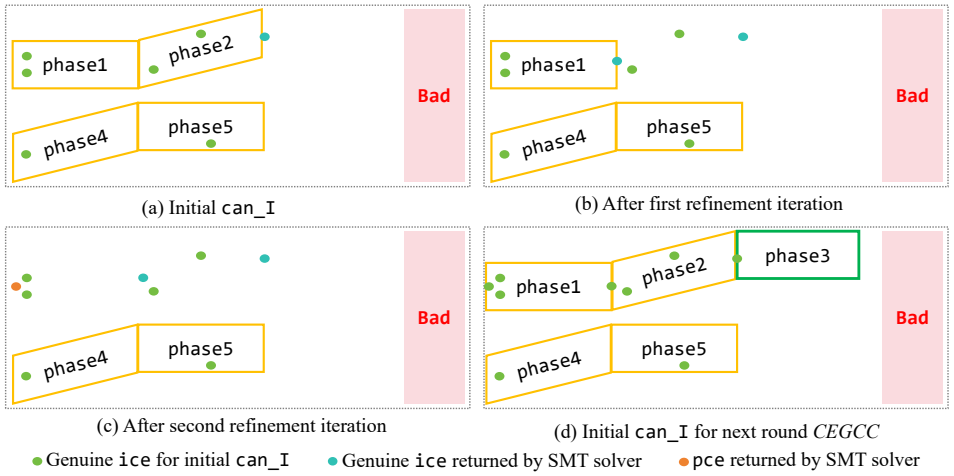



Fig. 5. *CEGCC* Refinement illustration.

When the returned counterexample ce is a positive counterexample (Lines 14-16), we stop the iteration since the current can_I has already violated the *reachability* condition and thus cannot be correct. On the other hand, if ce is a negative counterexample (Lines 17-19), we can filter all conjuncts that ce satisfies in each path safely, since the next proposed invariant must be false when assigned to ce. We use the FilterPathConjuncts function to filter conjuncts and reconstruct a new candidate invariant, which is implemented by updating path_conjs (Lines 24-34).

Once ce is an inductive counterexample (Lines 20-22), we filter the conjuncts that the first assignment of ce satisfies and generate a new candidate using FilterPathConjuncts. Theoretically, if ce is spurious, this refinement is safe because we should prevent satisfied conjuncts from being included in the next proposed candidate. However, if ce is genuine, we should in fact extend the next candidate's range to include the second assignment of ce, rather than shrinking it by filtering. In Alg. 3, we indeed choose to hypothesize that ce is always spurious and thus always refine greedily, because the problem of determining whether an inductive counterexample is genuine or spurious is almost impossible to solve, since determining the reachability of a program state is undecidable in general (reducible to the halting problem). As a result, the length of the newly proposed candidate is always decreasing, and we need to check whether there is an empty path to stop the refinement iteration in Line 3. Fortunately, even though ce is genuine and we have filtered some conjuncts that should be included in correct invariants, we indeed have more potential to generate a correct invariant in the next *generate-combine-check* iteration, since more genuine counterexamples, e.g., ce, have been preserved and will be used to combine clauses.

*Example 4.2.* For the program in Figure 1a, suppose the initial loop candidate invariant can_I is represented as the disjunction of correct invariants of all phases except phase3, as illustrated by the four yellow polygons in Figure 5a. Then, the SMT solver may return an inductive counterexample $\mathbf{(}(2000, 1, 1000),\ (2001, 1, 1000)\mathbf{)}$, which is genuine. Note that its first assignment meets the path invariant of $path_1$, which is now the disjunction of the invariant of phase2 and phase4. Consequently, the phase2 invariant conjunct will be removed, refining can_I to the configuration shown in Figure 5b. In the next refinement iteration, another returned genuine inductive counterexample, e.g., $\mathbf{(}(999, 1, 0),\ (1000, 1, 0)\mathbf{)}$, leads to the removal of the phase1 invariant conjunct. This refines can_I to the two yellow polygons depicted in Figure 5c. At this point, the SMT solver returns a positive counterexample because the condition $y > 0$ is ignored in the current can_I, causing the

current *generate-combine-check* iteration to terminate (Line 12 in Alg. 3). However, in the subsequent *generate-combine-check* iteration, the genuine counterexample $\mathbf{(}(2000, 1, 1000),\ (2001, 1, 1000)\mathbf{)}$ becomes useful for constructing the invariant of `phase3` (green polygon) in $path_2$. As a result, we finally obtain the correct loop invariants as shown in Figure 5d.

*4.3.3 Comparison with CEGAR.* Our invariant inference method shares the high-level iterative *generate-validate-refine* philosophy with Counterexample-Guided Abstraction Refinement (CEGAR) [10]. In particular, we adopt the core idea of using counterexamples returned by a verifier to guide refinement. However, our approach is more naturally positioned as an instance of the CEGIS [1] and ICE-learning [25] frameworks, from which it inherits the *PCE*, *ICE*, and *NCE* taxonomy as well as the guess-and-check loop structure.

We acknowledge that CEGAR is a general schema rather than a commitment to predicate abstraction refined by iteratively shrinking an over-approximation via interpolation alone [10]. Variants such as trace abstraction [31] refine by excluding concrete spurious traces, while loop acceleration techniques [24, 33, 36] explicitly incorporate under-approximations or exact transitive closures. Together, these approaches demonstrate that CEGAR encompasses diverse refinement mechanisms beyond classical interpolation-based predicate discovery. Thus, the choice of refinement mechanism, not merely the direction, is what distinguishes our approach.

The specific novelty of CEGCC lies in its mechanism for refinement, which draws inspiration from CEGAR's counterexample-guided spirit while operating within the ICE/CEGIS paradigm. In our framework, an LLM serves as the candidate proposer inside a guess-and-check loop, where counterexamples guide the composition of candidate clauses into a path-sensitive Disjunctive Normal Form (DNF). While the candidate invariant starts as a non-inductive under-approximation, subsequent verification feedback iteratively prunes conjuncts to eliminate spurious constraints, effectively expanding the represented state space toward a sound inductive invariant. This combination of LLM-based proposal generation and path-sensitive DNF construction fundamentally differs from classical interpolation-based predicate discoverers commonly used in CEGAR loops.

### 4.4 Optimizations for Combination

The performance of our loop invariant inference method mainly depends on two aspects: (1) How to improve the quality of the initially proposed candidate invariant, which is indeed a "how much" genuine inductive counterexample problem, because more genuine inductive counterexamples must make the initial candidate more likely to include the range of a correct loop invariant. (2) How to refine efficiently to stop as soon as possible when meeting genuine inductive counterexamples during verification, which is indeed a "whether" genuine problem.

In this subsection, we present three techniques to address these two problems and optimize our inference process, including abstract interpretation-based optimization, path-dependency analysis, and a second LLM-based agent that runs in parallel. All these techniques primarily address the "how much" problem, while the last one also considers the "whether" problem.

**Abstract Interpretation-Based Optimization.** Abstract interpretation serves not only as a pre-analysis step to produce coarse initial invariants for program verification or to guide LLM-based clause generation, but also plays a crucial role in improving the quality of counterexamples generated during inductiveness checking. Specifically, the coarse invariant derived from abstract interpretation defines a sound over-approximation of the correct inductive invariant; consequently, any genuine counterexample must lie within the region it represents. Counterexamples that fall outside this abstract domain are necessarily spurious. In our implementation, we enforce this constraint by including the coarse invariant in the SMT formula used for inductive checking, thereby generating counterexamples that are more likely to be genuine.

**Leveraging Path Dependency Information.** There are two primary limitations in Alg. 2: (1) the initial candidate loop invariant is formed by the disjunction of all proposed path invariants, each of which is constructed independently, without taking into account the dependencies between different paths; (2) the approach relies only on inductive counterexamples for combination and negative counterexamples for initial filtering, while it does not utilize positive counterexamples.

To address the first limitation, we conduct both forward and backward path-dependency analyses.

- Forward analysis identifies which iterative paths are reachable from the precondition. For each path formula $pathf_i$, if $pre \wedge pathf_i$ is satisfiable, then $path_i$ is reachable from an initial state. To obtain additional inductive counterexamples, we apply loop unrolling by conjoining multiple copies of $pathf_i$, e.g., $pre \wedge pathf_i \wedge pathf_i \wedge pathf_i$, thereby generating multiple genuine counterexamples along the same path.
- Backward analysis identifies predecessor and successor paths by exploiting the structure of inductive counterexamples. In SMT solvers for quantifier-free linear integer arithmetic (`QF_LIA`), counterexamples generated by simplex-based procedures [19] typically lie on the boundaries of feasible regions, which coincide with branch conditions. Given an inductive counterexample $ice_j \in path_i$, we extract its immediate predecessor $ice_{j'}$ and successor $ice_{j''}$, thereby identifying $path_{i'}$ and $path_{i''}$ as the predecessor and successor paths of $path_i$, respectively. Once an inductive invariant $Inv_i$ is synthesized for $path_i$, we use it to generate counterexamples for adjacent paths by solving $Inv_i \wedge pathf_{i''}$ or $pathf_{i'} \wedge Inv_i$. This mechanism enables invariant propagation across the paths.

To address the second limitation, we exploit positive counterexamples (*PCEs*) in a manner analogous to that used in precondition analysis. Since $\forall pce \in PCEs, pce \models pre\text{-}condition$, we can generalize each *pce* to a genuine inductive counterexample in the corresponding reachable path directly from the program inputs.

Remark 1. *Our path-dependency analysis method can leverage the preserved counterexamples generated during invariant verification. Unlike prior approaches such as path-dependency automata [74] or SMT-based pairwise path-dependency checks [39], our method reuses counterexamples generated during invariant verification. Moreover, in contrast to the quantifier-elimination-based forward-backward analysis in [55], we propagate path invariants through counterexample generation, yielding a lighter-weight analysis.*

*Example 4.3.* For the program in Figure 1a, suppose in $path_1$ we have derived the correct invariant for `phase2`, i.e., $y > 0 \wedge 1000y \leq x \wedge x \leq 1000(y+1) \wedge z == x - 1000y$, and there is an inductive counterexample $ice = \mathbf{(}(2000, 2, 0), (2001, 2, 1)\mathbf{)}$ in $path_1$. For this *ice*, we can compute its predecessor $ice' = \mathbf{(}(1999, 2, 0), (2000, 2, 0)\mathbf{)}$ in $path_2$, meaning that the program can execute from $path_2$ to $path_1$. Therefore, we can conjoin the current invariant of $path_1$ with that of $path_2$, and then use the SMT solver to derive another inductive counterexample in $path_2$, which, together with $ice'$, helps combine the clauses in $path_2$.

**LLM-Based Counterexample Verification and Mutation.** As we describe in §4.3.2, the greedy refinement algorithm (Alg. 3) always considers the inductive counterexamples generated during candidate invariant verification as spurious, often leading to iterations until some path invariants become empty. To terminate the refinement iteration as soon as a genuine counterexample is found and to generate additional genuine inductive counterexamples, we employ the LLM-based ICE verifier-mutator agent (V&M) in parallel with the initial combination process and the clause combination-refinement iteration, as shown in Figure 6.

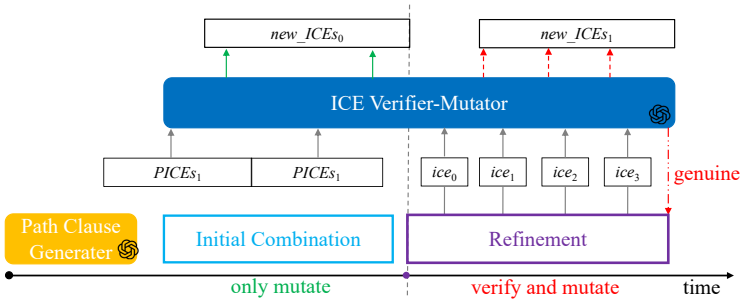



Fig. 6. Pipeline execution of two agents using the same LLM.

Before starting the process of constructing the initial combination of *CEGCC*, there are already some preserved inductive counterexamples in each iterative path, either from the prior *generate-combine-check* iteration or from the process of deriving preferred clauses. We select some typical boundary ones and denote them as $PICEs_i$ for the respective $path_i$ in Fig. 6. When constructing the initial combination, we call the V&M agent to mutate each $PICEs_i$ to generate additional genuine counterexamples using Prompt 2. The generated new counterexamples are stored in $new\_ICEs_0$. Once refinement starts, the V&M agent gives up mutating the remaining $PICEs_i$ and instead verifies the $ice_i$ returned by the verification of the current candidate loop invariant. If the V&M agent considers $ice_i$ as spurious, it will also generate genuine counterexamples that it guesses, which are stored in $new\_ICEs_1$. Otherwise, it will signal to stop the refinement iteration and start the next *generate-combine-check* iteration, in which counterexamples in $new\_ICEs_0$ and $new\_ICEs_1$ will also be used to combine clauses.

Overall, the prior path clause generator agent and the V&M agent execute in a pipeline. It's essential to note that the candidate invariant combination and refinement process, based on *CEGCC*, along with the two agents involved, is both time-consuming. This is due to multiple SMT checks and delays in obtaining answers from agents. By adopting a pipeline architecture, we enable a single LLM, particularly one deployed locally, to handle multiple tasks while improving the efficiency of loop invariant inference.

While the V&M agent operates heuristically, a recently dedicated benchmark study on LLM-based counterexample validation [20] shows that LLMs can effectively identify spurious counterexamples, supporting their usefulness as pre-filters. This is further corroborated by our ablation study in Section 5.3: disabling the agent reduces both the number of solved multi-phase programs and the average efficiency, indicating a net positive impact on coverage and performance. These benefits come with caveats: A false-genuine judgment may terminate a refinement round prematurely, whereas a false-spurious judgment or an unhelpful mutation may lead to additional pruning, SMT queries, outer iterations, and LLM token or time costs. Thus, the agent's non-determinism can affect the search trajectory, convergence, coverage, and computational overhead, but not soundness: every returned invariant is accepted only after SMT validation.

## 5 Evaluation

We evaluate the effectiveness of PAL2Inv by addressing the following research questions:

- **RQ1.** Does PAL2Inv outperform existing invariant inference methods on linear benchmarks?
- **RQ2.** Does PAL2Inv outperform existing invariant inference methods on multi-phase benchmarks?
- **RQ2.** What is the impact of individual design choices on the performance of PAL2Inv?

**Prompt 2: ICE List Verification and Mutation Prompt.**

**### Program ###**
{program}
**### Paths ###**
The following are all possible loop paths of the above program:
{paths}
Here, path 0 refers to the path that does not meet the loop condition. Other paths are indicated by the branch conditions they satisfy, which are denoted by their respective "Line" indices.
**### Task ###**
You are given a program along with some program states. All these program states are expected to occur along the path {index} of the program. However, some program states cannot occur during program execution and are unreachable.
For each given program state, you need to simulate the program execution and determine how to execute the path {index} from the program inputs. Then, you should determine whether the given program states can be reached from the program inputs after loop iterations. If a program state is reachable, include it in the final answer. Otherwise, provide some new program states that are reachable and can be executed along the path {index}.
Finally, you should answer a list of program states that are reachable from the program inputs and can execute in path {index} after loop iterations. The number of program states in your answer should be between 5 and 10, and each should be distinct from the others.
**### Given Program States ###**
{ICE list}
**### RESPONSE FORMAT ###**
{JSON format}
**### Your Response ###**

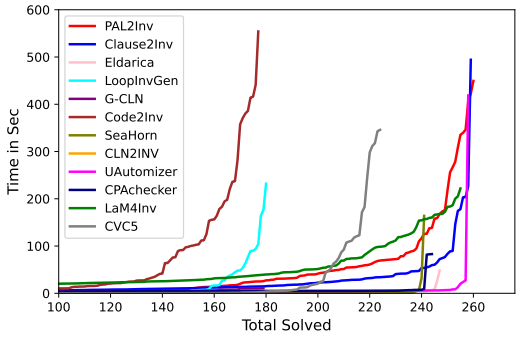
**Implementations**. For the implementation of PAL2Inv [2], we use Clam [52] as the abstract interpreter and ELINA [70] as the numerical domain library, with the polyhedra domain as the default. Note that Clam outputs are translated from LLVM to C automatically in PAL2Inv to facilitate LLM-based processing. As for LLMs, we use GPT-4 by default and also conduct experiments with GPT-4o mini, as well as a series of local QWen2.5 models ranging from 3B to 32B parameters with INT4 quantization. Currently, we do not consider LLMs that have a thinking mode. Besides, we automatically obtain path-sensitive SMT formulas by post-processing the outputs of Clang-fe [69].

**Benchmarks**. Our evaluation considers two benchmark suites. The first suite is the linear-invariant generation suite for 276 C programs from [72], which originally comprised 316 programs. We exclude the remaining 40 programs for the following reasons: (i) 29 programs contain conditional statements expressed in DNF or CNF format rather than as individual clauses; our approach currently relies on syntactic path sorting based on the source line of such statements, which is not well-defined for compound logical forms; (ii) 6 programs feature disjunctive preconditions, whereas our current invariant synthesis for the exit path employs a simple analysis of Clam that assumes conjunctive preconditions; and (iii) 5 programs have loop bodies that cannot be encoded into the disjunctive SMT formula format required by our framework. The second suite consists of 51 multi-phase C programs manually transformed via equivalence-preserving rewrites from the benchmarks in the CHC-COMP repository [58], which are originally from [22, 55]. Although the

[2]https://zenodo.org/records/17700961

| Method | #Solved | Avg. Time (s) |
|---|---|---|
| PAL2Inv | **260** | 34.62 |
| Clause2Inv | 259 | 20.83 |
| LaM4Inv | 255 | 42.82 |
| Code2Inv | 177 | 47.80 |
| CLN2INV | 168 | 0.13 |
| G-CLN | 179 | 1.93 |
| LoopInvGen | 180 | 9.22 |
| CVC5 | 225 | 16.61 |
| Eldarica | 247 | 2.34 |
| SeaHorn | 241 | 1.35 |
| CPAchecker | 245 | 7.44 |
| UAutomizer | 258 | 6.36 |

(a) Performance comparison summary.




(b) Time-to-Solution comparison.

Fig. 7. Comparison of different loop invariant inference methods on linear invariant generation.

original set contains 54 programs, we omit 3 that involve disjunctive preconditions, as they are incompatible with our current invariant generation strategy. All linear and multi-phase benchmarks are based on the `QF_LIA` theory, and all assertions in these benchmark programs are safe.

**Baselines**. The comparison includes recent LLM-based methods (Clause2Inv [7], LaM4Inv [72]), conventional learning-based systems (CLN2INV [60], G-CLN [76], Code2Inv [67]), symbolic synthesis approaches (LoopInvGen [54]), model checkers (CPAchecker [5], Ultimate Automizer ((henceforth referred to as UAutomizer)) [30]), and SMT-based verification engines (SeaHorn [29], CVC5 [2], Eldarica [34], ImplCheck [55]). Since ImplCheck only supports input in CHC format and also relies on syntactic learning, we evaluate it exclusively in **RQ2** on the benchmark programs in their original, unmodified CHC representation in [58]. Moreover, to ensure a fair and robust comparison, we invoke ImplCheck (version [56]) using the command-line configuration that solved the largest number of instances in the evaluation reported in [55]. For CPAchecker, we use the `-predicateAnalysis-linear` configuration, which enables predicate analysis and encodes integer variables in QF_LIA. All the other tools are executed with their default command-line configurations, and are evaluated in both **RQ1** and **RQ2**.

All the experiments were performed on a machine running Ubuntu 22.04 (64-bit), with 256GB RAM, 2 GeForce RTX 3090 GPUs, and a 3.7 GHz 32-core AMD 3970X CPU. We set the time limit for verifying each program to 600 seconds. No explicit memory limit was enforced, as all benchmarks consumed significantly less than the available memory.

### 5.1 Linear Invariant Generation (RQ1)

**Verification Coverage across Tool Categories.** Figure 7a summarizes the results on the linear invariant benchmarks, which consist of 276 instances in total. PAL2Inv solves 260 benchmarks when using GPT-4, achieving the highest coverage among all evaluated methods. Unlike other LLM-based approaches such as Clause2Inv (259 solved) and LaM4Inv (255 solved), PAL2Inv employs a path-sensitive inference strategy that generates loop invariants tailored to individual execution paths, enabling it to handle complex control-flow patterns that global or context-agnostic prompting cannot resolve. This design enables PAL2Inv to consistently outperform its LLM-based counterparts.

Beyond LLM-driven methods, PAL2Inv also surpasses conventional learning-based tools, including LoopInvGen (180 solved), Code2Inv (177 solved), CLN2INV (168 solved), and G-CLN (179 solved), confirming the advantage of leveraging large language models within a formal, path-aware

framework. PAL2Inv outperforms mature symbolic verifiers that provide specialized support for disjunctive invariants or path-sensitive analysis: UAutomizer (258 solved), Eldarica (247 solved), and CPAchecker (245 solved). Despite their dedicated algorithms [4, 27, 59] for handling disjunctions, these tools are still surpassed by PAL2Inv (260 solved), highlighting the unique benefit of LLM-guided path decomposition over purely symbolic predicate abstraction. In contrast, more generic solvers without explicit disjunctive reasoning, such as CVC5 (225 solved), show comparatively lower coverage on our benchmark suite.

**Runtime Efficiency and Search Strategy Trade-offs.** While PAL2Inv achieves the highest coverage, its runtime reflects a deliberate trade-off between expressiveness and efficiency. As shown in Figure 7a, PAL2Inv has a higher average runtime of 34.62 seconds across all programs it solves, compared to Clause2Inv (20.83 seconds), due to their contrasting search strategies. PAL2Inv adopts a top-down refinement approach: it starts from a disjunctive invariant candidate containing the maximal number of conjuncts and iteratively reduces the number of conjuncts until the resulting formula is sufficient for verification. In contrast, Clause2Inv follows a bottom-up strategy, starting with a simple single clause and progressively combining additional clauses to strengthen the invariant. Since most benchmarks in this suite admit loop invariants consisting of only a single clause, Clause2Inv's incremental composition is often more efficient, whereas PAL2Inv incurs overhead from exploring and pruning richer candidate spaces—even though this enables greater robustness on more complex instances.

On the other hand, PAL2Inv is significantly faster than LaM4Inv, which has an average runtime of 42.82 seconds. This difference stems from LaM4Inv's heavy reliance on repeated invocations of an external bounded model checker to validate partial invariants during synthesis. In our experiments, we observed that the original LaM4Inv implementation does not impose time limits on these external verification calls. As a result, individual model-checking queries often run beyond the overall per-instance timeout during program verification, thereby violating the experimental protocol. After fixing this issue by enforcing strict time bounds on all external calls, we found that LaM4Inv solves fewer benchmarks than originally reported in [72].

Moreover, PAL2Inv has a higher average runtime than highly optimized symbolic methods such as SeaHorn (1.35 seconds) and Eldarica (2.34 seconds). Like Clause2Inv and LaM4Inv, this overhead stems from repeated queries to large language models during invariant inference. These LLM calls can be affected by external factors such as network latency or API response latency, which are inherent to current LLM deployment practices and contribute to the overall execution time.

**Verification Throughput over Time.** To better understand these trade-offs, we examine the time-to-solution profile in Figure 7b, which plots the cumulative number of solved instances over time. Thanks to the abstract-interpretation-based pre-analysis, PAL2Inv can solve many programs significantly faster than Code2Inv, Clause2Inv, and LaM4Inv. Though the variance for PAL2Inv is wider than that of Clause2Inv after solving nearly 170 programs, there are still some challenging programs that PAL2Inv solves in less time than Clause2Inv, as these two curves intersect again. Furthermore, the ability to solve additional difficult tasks demonstrates that PAL2Inv prioritizes coverage in contexts where simpler heuristics or symbolic reasoning fall short.

Among mature verification tools, UAutomizer exhibits outstanding performance, rapidly verifying the vast majority of benchmarks and maintaining one of the steepest early-time curves. Nevertheless, on a small number of complex programs that require path-sensitive disjunctive invariants, UAutomizer takes considerably longer than PAL2Inv. This allows PAL2Inv not only to match but ultimately to surpass UAutomizer in the total number of solved instances. This balance indicates that PAL2Inv provides a complementary solution to traditional solvers, trading some runtime efficiency on easy cases for broader coverage on challenging ones.

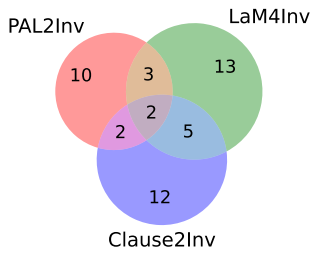



Fig. 8. Overlap in programs not solved by LLM-based tools.

Table 1. Number of successfully verified programs on linear benchmarks using different LLMs.

| LLM | PAL2Inv | Clause2Inv | LaM4Inv |
|---|---|---|---|
| QWen2.5:3b | **235** | 106 | 158 |
| QWen2.5:7b | **254** | 251 | 188 |
| QWen2.5:14b | 254 | **258** | 234 |
| QWen2.5:32b | 257 | **258** | 235 |
| GPT-4o-mini | 247 | **254** | 217 |
| GPT-4 | **260** | 259 | 255 |
| **Total** | **1507** | 1386 | 1287 |

**Systematic Comparison of Different LLM-Based Methods.** We further analyze failure modes by examining the overlap of unsolved programs across LLM-based methods, as shown in Figure 8. PAL2Inv shares very few unsolved programs with others, while Clause2Inv and LaM4Inv share the most 5 unsolved programs. This indicates that these two existing LLM-based methods have the same limitations that PAL2Inv can address. Additionally, we observe that almost all of the 10 programs in PAL2Inv have been unable to solve problems with more than 6 paths. These programs can be easily addressed by Clause2Inv and LaM4Inv, which utilize straightforward loop invariants that typically require very few clauses. However, these programs are challenging for PAL2Inv because it always tries to find invariants for all paths and refines them in decreasing order of length.

We finally compare the performance of PAL2Inv, Clause2Inv, and LaM4Inv on different LLMs. As shown in Table 1, when using the QWen2.5:3B model, PAL2Inv solves many more programs than the other methods, even 2.2 times as many as Clause2Inv. This is because, with the help of the loop invariant prompt returned by the abstract interpreter, even a small model can produce high-quality clauses. In other cases, PAL2Inv performs similarly to Clause2Inv, but still consistently outperforms LaM4Inv by a significant margin. Moreover, PAL2Inv outperforms the other two tools in terms of the total number of verified programs across all models.

### 5.2 Multi-Phase Invariant Generation (RQ2)

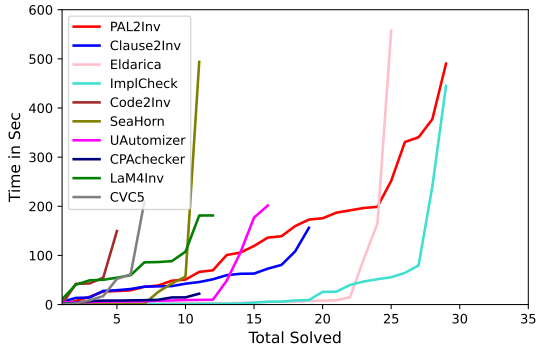
**Verification Coverage across Tool Categories.** Multi-phase programs require reasoning about state transitions across multiple execution stages, posing significant challenges for invariant inference. As shown in Figure 9a, none of the existing methods solves more than half of the total 51 benchmarks, underscoring the high complexity of the required loop invariants, which often involve intricate disjunctive forms. Among all evaluated tools, our method PAL2Inv achieves the highest coverage, successfully solving 29 instances when using GPT-4.

PAL2Inv substantially outperforms all existing LLM-based approaches: it solves 53% more than Clause2Inv (19 solved) and 142% more than LaM4Inv (12 solved). This advantage stems from PAL2Inv's use of `CEGCC`, a path-sensitive clause-combination strategy that enables the construction of disjunctive invariants that accurately capture program behavior across multiple execution phases. Notably, on this benchmark, Clause2Inv and LaM4Inv are outperformed by symbolic methods such as Eldarica and ImplCheck, despite having demonstrated a clear advantage over these same tools in RQ1 on linear invariant inference. Meanwhile, due to poor design for disjunctive invariant generation, learning-based approaches such as Code2Inv (5 solved), G-CLN (1 solved), and CLN2INV (1 solved) perform poorly, and the synthesis-based method LoopInvGen fails on all instances.

These results suggest that the challenge of multi-phase invariant inference does not lie in the inherent unsuitability of machine learning or large language models. Rather, it reflects a limitation in how current methods harness these technologies: without a formal, path-aware framework to

| Method | #Solved | Avg. Time (s) |
|---|---|---|
| PAL2Inv | **29** | 141.04 |
| Clause2Inv | 19 | 51.39 |
| LaM4Inv | 12 | 83.02 |
| Code2Inv | 5 | 58.00 |
| CLN2INV | 1 | 0.07 |
| G-CLN | 1 | 2.20 |
| LoopInvGen | 0 | 0.00 |
| CVC5 | 7 | 50.34 |
| Eldarica | 25 | 36.16 |
| SeaHorn | 11 | 56.68 |
| ImplCheck | **29** | 38.77 |
| CPAchecker | 11 | 9.75 |
| UAutomizer | 16 | 38.49 |

(a) Performance comparison summary.




(b) Time-to-Solution comparison.

Fig. 9. Comparison of different loop invariant inference methods on multi-phase invariant generation.

guide and compose LLM-generated clauses, even powerful models fail to produce invariants of sufficient precision and structure. PAL2Inv addresses this gap by tightly integrating LLM-derived semantic guidance with a rigorous CEGCC process, thereby unlocking the full potential of LLMs for complex verification tasks.

We acknowledge that PAL2Inv does not exhibit a decisive advantage over some symbolic methods, i.e., ImplCheck (29 solved) and Eldarica (25 solved), on this benchmark. Notably, both tools incorporate specialized algorithmic support for disjunctive invariant synthesis, and ImplCheck in particular employs a technique explicitly designed for multi-phase program analysis. Nevertheless, PAL2Inv matches ImplCheck in the number of verified programs while verifies several additional instances than Eldarica. This modest gain is particularly meaningful given the benchmark's highly challenging nature. Moreover, PAL2Inv also outperforms UAutomizer, which solves 16 instances despite being a state-of-the-art symbolic model checker that demonstrated exceptionally strong results in RQ1. In fact, PAL2Inv verifies 81% more programs than UAutomizer on the multi-phase benchmark.

To better understand the differences between PAL2Inv and these established symbolic approaches and to investigate why PAL2Inv still fails to verify all programs, we conduct a detailed comparison of PAL2Inv and Eldarica across the benchmark. We find that 8 programs are verified by Eldarica but not by PAL2Inv. In 7 of these, the postcondition involves conditional guards with specific numeric constants, such as `if (x > 5143523)`. We hypothesize that generating valid clauses for such cases requires the LLM to infer how program variables attain these precise values—a task that demands not only recognition of loop patterns but also deep symbolic reasoning. This capability is a core strength of traditional symbolic methods like Eldarica, yet it remains a known limitation of current large language models.

A similar pattern emerges when comparing PAL2Inv with ImplCheck. PAL2Inv verifies 9 programs that ImplCheck cannot, while ImplCheck verifies 10 that PAL2Inv cannot. Notably, almost all of these 10 programs, uniquely verified by ImplCheck, are also verified by Eldarica and exhibit the same characteristic: their postconditions involve conditional guards with precise numeric constants. This further supports our hypothesis that LLM-based methods like PAL2Inv struggle with such precise numeric reasoning, whereas symbolic approaches—whether constraint-based (Eldarica) or data-driven with symbolic components (ImplCheck)—are better equipped to handle them.

Table 2. Number of successfully verified programs on multi-phase benchmarks using different LLMs.

| LLM | PAL2Inv | Clause2Inv | LaM4Inv |
|---|---|---|---|
| QWen2.5:3b | **13** | 2 | 1 |
| QWen2.5:7b | 14 | **17** | 3 |
| QWen2.5:14b | **17** | 14 | 12 |
| QWen2.5:32b | **20** | 13 | 13 |
| GPT-4o-mini | 14 | **17** | 8 |
| GPT-4 | **29** | 19 | 12 |
| **Total** | **107** | 82 | 49 |

Table 3. Results of ablation study for linear and multiphase benchmarks.

| RQ | Method | #Solved | Avg. Time (s) |
|---|---|---|---|
| 1 | Clam-int | 56 | 0.22 |
| | Clam-pk | 132 | 0.23 |
| | Full | 260 | 34.62 |
| 2 | Clam-int | 0 | 0.00 |
| | Clam-pk | 1 | 0.27 |
| | No V&M | 24 | 149.03 |
| | LLM only | 3 | 29.79 |
| | Full | 29 | 141.04 |

**Runtime Efficiency.** As for efficiency, the runtime analysis shown in Figure 9a reveals that PAL2Inv incurs a higher average solving time (141.04 seconds) on the instances it verifies. However, this overhead is strongly correlated with benchmarks that no other method can solve, suggesting that the cost stems primarily from the intrinsic difficulty of these cases. Moreover, nearly all tools exhibit substantially longer average runtimes on this benchmark than on the linear invariant tasks in RQ1, further highlighting the greater challenge of multi-phase verification.

Notably, ImplCheck, which resolves the same number of instances, consumes significantly less time, averaging 38.77 seconds per verified program. This efficiency stems from its use of model-based projection to compute program phases and a data-driven approach that infers relationships among variables using techniques such as data matrix computation. In contrast, PAL2Inv incurs overhead due to network latency and LLM inference costs and additionally relies on CEGAR-based clause combination and refinement. Furthermore, CPAchecker reports a lower average runtime (9.75 seconds) but solves only 11 instances; this fast average reflects quick termination on simpler cases rather than superior efficiency on complex multi-phase benchmarks.

**Verification Throughput over Time.** The time-to-solution profile shown in Figure 9b reveals deeper insights into each tool's behavior. PAL2Inv addresses a critical gap: it enables invariant inference in domains where existing techniques fail to produce any solution. Notably, PAL2Inv maintains a nearly steady slope as it verifies up to 25 programs, and even beyond that point, it continues to solve additional instances at a steeper but sustained rate, ultimately reaching 29 verified benchmarks.

In contrast, Clause2Inv matches or slightly outperforms PAL2Inv on the first 19 solved instances but times out on all remaining benchmarks. LaM4Inv consistently incurs higher runtime than PAL2Inv across the entire range, consistent with the limitations discussed in RQ1. Traditional symbolic tools such as ImplCheck and Eldarica exhibit a characteristic pattern: they verify many programs quickly, but their runtime grows sharply once program complexity exceeds a threshold. Moreover, the slowest program verified by PAL2Inv takes less time than the slowest verified by Eldarica, reflecting stronger worst-case efficiency on solvable instances.

**Performance across Language Models.** To further assess robustness, we evaluate PAL2Inv, Clause2Inv, and LaM4Inv under different LLM backends. As shown in Table 2, PAL2Inv consistently outperforms LaM4Inv across all models, and surpasses Clause2Inv except when using weaker models such as GPT-4o mini or QWen2.5:7B, whose limited reasoning capabilities hinder PAL2Inv's full potential. Notably, even with small models like QWen2.5:3B, PAL2Inv maintains strong performance, while Clause2Inv and LaM4Inv solve fewer than five instances. Moreover, PAL2Inv's effectiveness


(a) Linear invariant inference. (b) Multi-phase invariant inference.

Fig. 10. Performance of not employing abstract interpretation or using different abstract domains.

scales with model capacity, suggesting that advances in LLM architecture are likely to yield further improvements in verification power.

Overall, PAL2Inv demonstrates a clear advantage in generating multi-phase invariants. This advantage is particularly important because multi-phase reasoning is essential for proving correctness in practical software systems, where loops often encode state-dependent behavior.

### 5.3 Evaluation of the Design Choices (RQ3)

We conduct an ablation study to assess the contribution of key components. First, we evaluate how many programs Clam can verify across different abstract domains to isolate the impact of pure abstract interpretation as the pre-analysis phase in PAL2Inv. To ensure a fair comparison, we select two representative domains: the strong-relational polyhedra domain (pk) and the non-relational interval domain (int). All other settings of Clam remain at their default values, including the widening threshold and other parameters.

As shown in Table 3, this pre-analysis alone solves 132 benchmarks in **RQ1** with the polyhedra domain and 56 with the interval domain—a predictable outcome, as polyhedra provide strictly greater precision than intervals for linear invariants. However, because the polyhedra domain can only represent convex properties, it fails to capture the disjunctive invariants required by multi-phase programs. Consequently, Clam verifies only a single benchmark in **RQ2**. In contrast, by integrating LLM-guided clause generation with path-sensitive composition, PAL2Inv successfully solves nearly all benchmarks in both RQ1 and RQ2, as the "Full" strategy shows.

Meanwhile, we evaluate the influence of the V&M agent. Without it, PAL2Inv can still verify 24 multi-phase programs in Table 3, more than any other existing method except Eldarica (25). This highlights the effectiveness of *CEGCC*, achieved even with the implementation of just the greedy refinement algorithm. Additionally, there is an increase in the total number of solved programs and a decrease in the average time spent, thanks to the second agent's parallel operation.

We also evaluate the ability of LLMs to infer multi-phase loop invariants by directly prompting GPT-4 to return full invariants for the input programs, as [72] did. As shown in Table 3, GPT-4 can derive correct invariants for only 3 programs. However, based on this powerful model, PAL2Inv outperforms all existing methods.

Figure 10 further shows the effect of model scaling with (or without, labeled as "none") the help of employing abstract interpretation based on the interval domain or polyhedra domain. Larger QWen2.5 models improve performance, while pre-analysis based on a more precise abstract domain also increases the solved instances. For both the linear benchmark (Figure 10a) and the multi-phase benchmark (Figure 10b), the application of the polyhedra domain outperforms the interval domain, and the case without any pre-analysis. In Figure 10b, performance in the interval domain degrades because the 32B model takes longer to respond.

### 5.4 Discussions

**Threats to Validity.** There are mainly four validity threats. First, *CEGCC* generates candidate invariants in disjunctive normal form (DNF), where each conjunct corresponds to a counterexample. Since we do not simplify these candidates before encoding them into SMT formulas, the resulting encodings often contain many disjunctions. As noted in prior work [3], a high degree of disjunctiveness can significantly degrade SMT solver performance. Second, we evaluate our method on limited benchmarks, while our current implementation still relies on syntactic analysis and needs improvement to support real-world programs. Third, due to the opaque nature of LLM training, it remains unknown whether our benchmarks were included in the model's training data. However, because loop invariants are difficult to derive using existing methods, this problem is less prominent. Fourth, the non-determinism inherent in LLM-based inference—stemming from stochastic decoding under default temperature settings—is not quantified in our evaluation. Although we used consistent prompting during experiments, variations across runs could still affect invariant generation.

**Theoretical Limitations.** Our work primarily demonstrates PAL2Inv's empirical effectiveness through extensive experiments. However, we acknowledge that it currently lacks a formal theoretical foundation that provides guarantees regarding its convergence, completeness, or computational complexity bounds. This is a recognized limitation of our current approach, like other existing LLM-based methods [7, 72].

- *Convergence*: The CEGCC loop is guaranteed to terminate if the SMT solver validates the candidate invariant. However, due to the undecidability of the general loop invariant inference problem, termination cannot be guaranteed for all programs. In practice, our method converges when the LLM generates a sufficient set of atomic clauses and the counterexample-guided refinement successfully assembles them into a valid invariant.
- *Completeness*: The completeness of PAL2Inv hinges on the LLM's ability to generate all atomic predicates necessary to construct the target invariant. Since LLMs are inherently stochastic and may hallucinate or omit critical clauses, our method is heuristic and not complete in the formal sense. The path-sensitive generation and abstract interpretation guidance aim to increase the probability of generating relevant clauses, but do not eliminate this fundamental incompleteness.
- *Complexity*: The worst-case time complexity is dominated by the interplay between LLM invocations, the combinatorial space of clause combinations, and SMT solving. While the counterexample-guided strategy prunes the search space effectively in practice, the lack of a formal bound means that resource consumption (time, memory) can vary significantly across benchmarks. Our experimental evaluation shows that the approach is feasible for the considered benchmarks, but a rigorous complexity analysis remains an important direction for future work.

**Better Neuro-symbolic Integration.** Currently, abstract interpretation is primarily used to obtain initial sound loop invariants for given programs, and is therefore integrated with LLM-based invariant inference in a somewhat loosely coupled manner. However, abstract interpretation can play additional roles, such as determining invariants of so-called path programs [4, 27] to optimize invariant inference for each loop path, and simplifying candidate invariants [17] to accelerate SMT solving. Furthermore, we can integrate symbolic execution [42] into our framework to improve path invariant inference. Finally, and most importantly, the current *CEGCC* iterations operate largely in isolation: each round starts with an initial candidate combination and does not leverage knowledge from prior iterations. This lack of inter-iteration feedback limits both the speed of invariant inference and the total number of programs that can be successfully verified. Future

work will explore mechanisms to propagate and refine invariant hypotheses across *CEGCC* cycles, thereby accelerating inference and increasing the number of verification instances.

## 6 Related Work

**Loop Invariant Inference.** Invariant inference techniques differ along two main axes: inference strategy and syntactic expressiveness. One axis of classification distinguishes between eager and lazy approaches. Eager methods, such as abstract interpretation [12, 13] aim to infer all invariants within a fixed framework, independent of their relevance to the proof. Lazy methods, in contrast, generate invariants on demand, guided by verification conditions. Our approach follows the lazy paradigm, refining invariants only as needed, as in CEGAR [11, 80]. A second axis concerns the syntactic expressiveness of inferred invariants. Abstract interpretation is constrained by the chosen domain (e.g., intervals [14], octagons [50]), and constraint-based methods [8, 62, 77] restrict inference to fixed templates. Our method imposes no such restrictions and can infer linear invariants with complex Boolean structures. Learning-based approaches typically follow a *guess-and-check* paradigm, using decision trees [25, 26, 75], SVMs [43, 66], PAC learning [65], neural networks [60, 76], or reinforcement learning [67, 78]. While expressive in principle, these methods often struggle with disjunctive invariants. ICE-based techniques [25], such as ICE-DT [26], Code2Inv [67], LIPus [78], rely on fixed templates, which limit their applicability to multi-path programs. The other Neural methods, such as CLN2INV [60] and G-CLN [76], capture only simple logical structures.

**LLM for Program Verification.** In the specific domain of loop invariant inference, recent studies have diversified the integration strategies between LLMs and formal methods. LaM4Inv [72] uses model checking to filter invalid predicates before combining preserved ones; Loopy [38] applies the Houdini algorithm to converge to a correct inductive subset from LLM-generated guesses; and Clause2Inv [7] adapts the typical *guess-and-check* framework into a *generate-combine-check* pipeline tailored to LLM response characteristics. Beyond these predicate-centric approaches, King et al. [41] target weakest preconditions and precise array invariants, Bharti et al. [6] combine reasoning-optimized LLMs with SMT solvers for synthesis, and Liu et al. [45] enhance generation for complex programs via LLM augmentation. While these advances significantly broaden the applicability of LLMs in invariant discovery, they primarily focus on monolithic or path-insensitive invariant structures. PAL2Inv differs by explicitly decomposing loops along control-flow paths and composing path-specific clauses into disjunctive invariants, directly addressing the gap in handling path sensitivity and disjunction that remains underexplored in prior LLM-based approaches. Our method builds upon the *generate-combine-check* framework introduced by Clause2Inv, but substantially improves both the *generate* and *combine* steps through path-aware decomposition.

A few attempts have explored LLMs' capabilities in program verification, despite the loop-invariant inference discussed above. LEMUR [73] employs LLMs to generate properties that assist ESBMC in program verification. PropertyGPT [46] applies LLM-based property generation to derive comprehensive formal verification for smart contracts. Additionally, some work leverages LLMs to synthesize program specifications, including AutoSpec [71] and SpecGen [48]. Laurel [51] uses LLMs to automatically generate intermediate assertions for automated verifiers like Dafny, thereby reducing the burden of manual proof engineering. Complementing these efforts, recent work [47] investigates the use of LLMs for formal proof automation in interactive theorem provers, proposing a framework that combines LLM-generated proof sketches with symbolic repair to significantly improve proof success rates.

## 7 Conclusion

We present PAL2Inv, a path-sensitive loop invariant inference method that integrates LLMs with abstract interpretation. PAL2Inv enhances the *generate–combine–check* framework by improving both the *generate* and *combine* processes, enabling more effective use of LLMs and inference of disjunctive invariants. The core contribution is a counterexample-guided clause combination strategy that constructs and refines candidate invariants from counterexamples. Experimental results show that PAL2Inv matches the performance of existing LLM-based approaches on linear benchmarks and substantially outperforms them on multi-phase programs.

## 8 Data Availability

All source code of PAL2Inv and the compared tools, experimental results, and benchmark programs are publicly available at https://zenodo.org/records/17700961.

## Acknowledgments

We thank the TOSEM reviewers and editors for their constructive feedback. This work is supported by the National Key R&D Program of China (No.2022YFA1005101) and the National Natural Science Foundation of China (Nos.62032024, 62302434 and U2341212).

## References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*. Springer, 270–288.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 305–343.

[4] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Path invariants. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 300–309.

[5] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *International conference on computer aided verification*. Springer, 184–190.

[6] Varun Bharti, Shashwat Jha, Dhruv Kumar, and Pankaj Jalote. 2025. Combining Reasoning Optimized LLMs and SMT Solvers for Automated Loop Invariant Synthesis. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. IEEE, 192–196.

[7] Weining Cao, Guangyuan Wu, Tangzhi Xu, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2025. Clause2Inv: A Generate-Combine-Check Framework for Loop Invariant Inference. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1009–1030.

[8] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 672–687.

[9] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design* 19, 1 (2001), 7–34.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

[12] Patrick Cousot. 2021. *Principles of abstract interpretation*. MIT Press.

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[14] Patrick Cousot and Radhia Cousot. 1977. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*. 77–94.

[15] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. 84–96.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[17] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *International Static Analysis Symposium*. Springer, 236–252.
[18] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. *Acm Sigplan Notices* 48, 10 (2013), 443–456.
[19] Bruno Dutertre and Leonardo De Moura. 2006. A fast linear-arithmetic solver for DPLL (T). In *International Conference on Computer Aided Verification*. Springer, 81–94.
[20] Guangsheng Fan, Dengping Wei, and Banghu Yin. 2026. Spurious or Genuine? Evaluating Large Language Models in Validating Counterexamples for Loop Invariant Inference. *Electronics* 15, 6 (2026), 1148.
[21] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 57–64.
[22] Grigory Fedyukovich. 2020. Multi-phase CHC Bechmarks used in ImplCheck. https://github.com/chc-comp/aeval-benchmarks/blob/main/multi-phase/.
[23] Grigory Fedyukovich. 2020. Original CHC program used in ImplCheck. https://github.com/grigoryfedyukovich/aeval/blob/rnd/bench_horn/s_split_27.smt2.
[24] Florian Frohn and Jürgen Giesl. 2023. ADCL: Acceleration driven clause learning for constrained Horn clauses. In *International Static Analysis Symposium*. Springer, 259–285.
[25] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.
[26] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.
[27] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. 2017. Loop invariants from counterexamples. In *International Static Analysis Symposium*. Springer, 128–147.
[28] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A symbolic abstract domain of boxes. In *International Static Analysis Symposium*. Springer, 287–303.
[29] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.
[30] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMTInterpol: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 641–643.
[31] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 69–85. doi:10.1007/978-3-642-03237-0_7
[32] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.
[33] Hossein Hojjat, Radu Iosif, Filip Konečný, Viktor Kuncak, and Philipp Rümmer. 2012. Accelerating interpolants. In *Proceedings of the 10th International Conference on Automated Technology for Verification and Analysis* (Thiruvananthapuram, India) *(ATVA'12)*. Springer-Verlag, Berlin, Heidelberg, 187–202. doi:10.1007/978-3-642-33386-6_16
[34] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–7.
[35] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017. Invariant generation for multi-path loops with polynomial assignments. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 226–246.
[36] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. 2014. Abstract acceleration of general linear loops. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 529–540.
[37] Ranjit Jhala and Kenneth L McMillan. 2006. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 459–473.
[38] Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2024. Leveraging llms for program verification. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 107–118.
[39] Jingyu Ke, Hongfei Fu, Hongming Liu, Zhouyue Sun, Liqian Chen, and Guoqiang Li. 2025. Affine disjunctive invariant generation with farkas' lemma. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 187–213.
[40] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

[41] Daragh King, Vasileios Koutavas, and Laura Kovács. 2025. Llm-Based Generation of Weakest Preconditions and Precise Array Invariants. In *2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 1–5.

[42] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[43] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation anc refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 782–792.

[44] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. 2017. Fib: Squeezing loop invariants by interpolation between forward/backward predicate transformers. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 793–803.

[45] Ruibang Liu, Minyu Chen, Ling-I Wu, Jingyu Ke, and Guoqiang Li. 2026. Enhancing automated loop invariant generation for complex programs with large language models. *Science of Computer Programming* 248 (2026), 103387. doi:10.1016/j.scico.2025.103387

[46] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580* (2024).

[47] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1509–1520.

[48] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807* (2024).

[49] Kenneth L McMillan. 2010. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*. Springer, 104–118.

[50] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19 (2006), 31–100.

[51] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1519–1545.

[52] Jorge Navas. 2025. Clam. https://github.com/seahorn/clam.

[53] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 605–615.

[54] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[55] Daniel Riley and Grigory Fedyukovich. 2022. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 607–619.

[56] Daniel Riley and Grigory Fedyukovich. 2023. ImplCheck Source Code. GitHub, latest commit. https://github.com/grigoryfedyukovich/aeval/commit/2b40266c9d9d0c13cf6c26d2a9b4c6884be7f599

[57] Philipp Rümmer. 2021. Original CHC program used in the CHC-COMP. https://github.com/chc-comp/aeval-benchmarks/blob/main/multi-phase/s_split_27.smt2.

[58] Philipp Rümmer. 2025. aeval-benchmarks. https://github.com/chc-comp/aeval-benchmarks.

[59] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive interpolants for Horn-clause verification. In *International Conference on Computer Aided Verification*. Springer, 347–363.

[60] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2019. CLN2INV: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542* (2019).

[61] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 318–329.

[62] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2005. Scalable analysis of linear systems using mathematical programming. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–41.

[63] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design* 48, 3 (2016), 235–256.

[64] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying loop invariant generation using splitter predicates. In *International Conference on Computer Aided Verification*. Springer, 703–719.

[65] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. 2013. Verification as learning geometric concepts. In *International Static Analysis Symposium*. Springer, 388–411.

[66] Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *International Conference on Computer Aided Verification*. Springer, 71–87.

[67] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems* 31 (2018).
[68] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2inv: A deep learning framework for program verification. In *International Conference on Computer Aided Verification*. Springer, 151–164.
[69] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2025. Clang-fe. https://github.com/PL-ML/code2inv/tree/master/clang-fe.
[70] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2025. ELINA. https://elina.ethz.ch/.
[71] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328.
[72] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417.
[73] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating large language models in automated program verification (2023). *arXiv preprint arXiv:2310.04870* (2024).
[74] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 61–72.
[75] Rongchen Xu, Fei He, and Bow-Yaw Wang. 2020. Interval counterexamples for loop invariant learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 111–122.
[76] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.
[77] Peisen Yao, Jingyu Ke, Jiahui Sun, Hongfei Fu, Rongxin Wu, and Kui Ren. 2023. Demystifying Template-Based Invariant Generation for Bit-Vector Programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 673–685.
[78] Shiwen Yu, Ting Wang, and Ji Wang. 2023. Loop invariant inference through SMT solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 175–187.
[79] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *ACM SIGPLAN Notices* 53, 4 (2018), 707–721.
[80] He Zhu, Aditya V Nori, and Suresh Jagannathan. 2015. Learning refinement types. *ACM SIGPLAN Notices* 50, 9 (2015), 400–411.