Clause2Inv: A Generate-Combine-Check Framework for Loop Invariant Inference

WEINING CAO, Nanjing University, China
GUANGYUAN WU, Nanjing University, China
TANGZHI XU, Nanjing University, China
YUAN YAO, Nanjing University, China
HENGFENG WEI, Nanjing University, China
TAOLUE CHEN, Birkbeck University of London, United Kingdom
XIAOXING MA, Nanjing University, China

Loop invariant inference is a fundamental, yet challenging, problem in program verification. Recent work adopts the *guess-and-check* framework, where candidate loop invariants are iteratively generated in the *guess* step and verified in the *check* step. A major challenge of this general framework is to produce high-quality candidate invariants in each iteration so that the inference procedure can converge quickly. Empirically, we observe that existing approaches may struggle with guessing the complete invariant due to the complexity of logical connectives, but usually, all the *clauses* of the correct loop invariant have already appeared in the previous guesses. This motivates us to refine the guess-and-check framework, resulting in a *generate-combine-check* framework, where the loop invariant inference task is divided into clause generation and clause combination. Specifically, we propose a novel loop invariant inference approach Clause2Inv under the new framework, which consists of an LLM-based clause generator and a counterexample-driven clause combinator. As the clause generator, Clause2Inv leverages LLMs to generate a multitude of clauses; as the clause combinator, Clause2Inv leverages counterexamples from the previous rounds to convert generated clauses into invariants. Our experiments show that Clause2Inv significantly outperforms existing loop invariant inference approaches. For example, Clause2Inv solved 312 (out of 316) linear invariant inference tasks and 44 (out of 50) nonlinear invariant inference tasks, which is at least 93 and 16 more than the existing baselines, respectively. By design, the *generate-combine-check* framework is flexible to accommodate various existing approaches which are currently under the *guess-and-check* framework by splitting the guessed candidate invariants into clauses. The evaluation shows that our approach can, with minor adaptation, improve existing loop invariant inference approaches in both effectiveness and efficiency. For example, Code2Inv which solved 210 linear problems with an average solving time of 137.6 seconds can be improved to solve 252 problems with an average solving time of 17.8 seconds.

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

Additional Key Words and Phrases: Program verification, loop invariant, historical predictions, clause combination

Authors' Contact Information: Weining Cao, Nanjing University, Nanjing, China, weiningcao@smail.nju.edu.cn; Guangyuan Wu, Nanjing University, Nanjing, China, guangyuanwu@smail.nju.edu.cn; Tangzhi Xu, Nanjing University, Nanjing, China, xutz@smail.nju.edu.cn; Yuan Yao, Nanjing University, Nanjing, China, y.yao@nju.edu.cn; Hengfeng Wei, Nanjing University, Nanjing, China, hfwei@nju.edu.cn; Taolue Chen, Birkbeck University of London, London, United Kingdom, taolue.chen@gmail.com; Xiaoxing Ma, Nanjing University, Nanjing, China, xxm@nju.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA045
https://doi.org/10.1145/3728920

ACM Reference Format:
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. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA045 (July 2025), 21 pages. https://doi.org/10.1145/3728920

1 Introduction

Bugs and security vulnerabilities pose significant threats to our daily life with software's ever-increasing ubiquity and complexity. Program verification, which aims to prove the absence of these problems at compile-time, is an indispensable tool to mitigate these risks [17]. Loop invariant plays a central role in program verification. Specifically, a loop invariant is a proposition that holds before and after each iteration of a loop, the inference of which is a central task in deductive verification based on, e.g., Hoare logic and its variants. Loop invariant inference is, however, undecidable in general, and turns out to be challenging in practice, making it a subject of active research for over 40 years.

Traditionally, loop invariant inference methods mainly leverage deductive reasoning, such as constraint solving [9, 21, 22, 42], counterexample guided abstraction refinement [8, 48], and Craig interpolation [24, 31]. However, this class of approaches is often not scalable; some of them rely highly on manual intervention.

Another class of approaches leverages inductive reasoning, which is of a data-driven nature [19, 20, 32, 37, 38, 49]. At an earlier stage, some approaches cast the invariant inference problem as a supervised learning problem [19, 37] where various classic machine learning algorithms can be applied. One of the serious shortcomings is that they require handcrafted features, which turns out to be challenging for complex, unstructured programs [20]. Moreover, some of the standard linear classification methods, such as SVM [47] and Perceptron [18], may overfit to existing data, limiting their ability to infer a correct loop invariant.

More recent works along this line have adopted the *guess-and-check* framework [22, 33, 36, 44–46], where candidate invariants are iteratively generated in the *guess* step and verified in the *check* step. While these methods lack completeness guarantees, in practice they usually outperform the earlier approaches. Additionally, they can instantiate cutting-edge (data-driven) approaches during the *guess* step, significantly improving their effectiveness and scalability. Indeed, various machine learning techniques, such as decision tree [20], reinforcement learning [41, 46] and continuous logic networks [36, 45], have been applied to generate candidate invariants.

Our work. One drawback of the guess-and-check methods lies in that they are not effective at utilizing previously failed guesses. Namely, different rounds of guesses are largely independent and the insight of the (albeit failed) previous ones was not sufficiently harnessed later on.

In this paper, we propose to refine the “guess" step into two separated but closely related steps, i.e., generating clauses and then combining clauses to derive a candidate loop invariant, giving rise to a new *generate-combine-check* framework. To instantiate this new framework, we propose to leverage large language models (LLMs) as the backbone clause generator and a counterexample-driven clause combinator, leading to a tool Clause2Inv. In a nutshell, for the clause generator, we craft prompts specifically aimed at eliciting assertions from LLMs. These assertions are free of logical connectives but are with emphasis on specific parts of the program under consideration. For the clause combinator, we maintain all the previously generated assertions and devise an algorithm to systematically assemble them, thereby deriving the candidate loop invariant.

The underpinning rationale of our design is that the state-of-the-art LLMs excel in “brainstorming”, but not in logical reasoning. As a result, fully-fledged loop invariants (with logic connectives) generated directly from LLMs may not be of high quality, which is also observed by our experiments. Instead, the clauses/assertions produced by LLMs turn out to be much more useful. In particular, we can devise suitable prompts to incorporate the lesson learned from the previous mistakes so that LLMs can return more targeted clauses. These clauses are assembled by a symbolic combinator, which can, again, make use of failed candidate loop invariants (from previous rounds) via counterexamples. As such, we design a new form of neuro-symbolic system (i.e., LLM as the neural component and the clause combinator as the symbolic component) for loop invariant inference.

Evaluation. To thoroughly evaluate Clause2Inv, we conduct experiments on an extended linear loop invariant benchmark comprising 316 problems, including 133 problems from Code2Inv [41], 84 additional problems from the 2019 SyGuS competition [1], and 99 new problems from the 2024 SV-COMP benchmarks [5]. We also extend the existing nonlinear loop invariant benchmark from 30 problems [46] to 50 problems, with 20 additional problems mainly from the 2024 SV-COMP benchmarks. The performance of Clause2Inv is compared against seven typical methods, i.e., LoopInvGen [34], CVC5 [3], Code2Inv [41], LIPuS [46], CLN2INV [36], G-CLN [45], and SymInfer [33]. The experimental results show that Clause2Inv outperforms the state-of-the-art methods. In particular, the number of linear problems solved is increased from 219 (previously by G-CLN) to 312 by Clause2Inv, and the number of nonlinear problems solved is increased from 28 (previously by SymInfer) to 44. We refer the readers to Section 4.2 for details.

In addition, for tools which are currently under the *guess-and-check* framework, we can modify them as per the proposed *generate-combine-check* framework. Specifically, historical guesses from these base tools are gathered and split into clauses, and then our clause combinator is applied to derive the loop invariant. As such, we obtain variants of respective base tools. In our experiments, Code2Inv, LIPuS, CLN2INV and G-CLN are selected for evaluation. The result shows that their performance is improved by 20% at least (cf. Section 4.3 for details), which demonstrates the efficacy and wide applicability of the *generate-combine-check* framework.

Our contributions can be summarized as follows.

- We propose a novel *generate-combine-check* framework for loop invariant inference, which clearly divides the problem into clause generation and clause combination, so that the discarded clause information can be better leveraged to speed up the convergence of the traditional guess-and-check process.
- We instantiate the framework by an LLM-based clause generator and a counterexample-driven clause combinator, implementing a new tool Clause2Inv. This establishes a new paradigm of neuro-symbolic synergy for challenging program verification tasks.
- We curate new benchmarks for loop invariant inference and conduct extensive experiments on them. The results confirm that Clause2Inv sets up a new baseline for loop invariant inference, and moderate adaptations of various existing methods as per our new framework can enhance their respective performance.

Roadmap. The rest of the paper is organized as follows. Section 2 introduces the background knowledge and presents a motivating example. Section 3 describes the proposed approach. Section 4 shows the evaluation results. Section 5 discusses threats to validity. Section 6 covers the related work. Section 7 concludes the paper.

2 Preliminary and Motivation

In this section, we first introduce the loop invariant inference problem and then present a motivating example.

2.1 Problem Statement

The purpose of loop invariant inference is to identify a loop invariant that can be used to demonstrate that a program exhibits the expected properties. For a loop of the form `while` B `do` S, the classical

Hoare logic stipulates that a loop invariant I must satisfy

$$\frac{P \implies I \quad \{I \land B\}\, S\, \{I\} \quad (I \land \neg B) \implies Q}{\{P\} \text{ while } B \text{ do } S\ \{Q\}}.$$

Here, P is the *pre-condition*, Q is the *post-condition*, and S is the loop body (consisting of a sequence of statements) with B as the loop condition (boolean expression). Essentially, the above rule indicates that the loop invariant I satisfies the following three conditions.

- *Prev.* The loop invariant must hold before entering the loop. This ensures that the invariant is true at the start, before the execution of the loop body.
- *Inv.* If the loop invariant is true before an iteration of the loop and the loop condition holds, then the invariant must still be true after executing the loop body. This ensures that the invariant remains true throughout the loop's execution.
- *Post.* If the loop terminates, then the loop invariant combined with the negation of the loop condition should imply the post-condition. This ensures that the loop invariant is strong enough to guarantee that the desired property (represented by the post-condition) will always hold once the loop terminates.

In practice, SMT solvers (e.g., Z3 [12]) are typically used to verify the correctness of three (logical) conditions. If any of them fails, the SMT solver usually provides a concrete counterexample. It includes the assignments to the variables within the program and indicates which specific constraint has been violated.

2.2 Motivating Example

Listing 1 presents a simple program with two variables and a single loop. A valid loop invariant for this program is $(x \leq n) \lor (x = 0)$. While this invariant may appear to be straightforward for human beings, deducing it automatically requires considerable efforts. We summarize two common limitations of existing learning-based methods.

Listing 1. A motivating example

```
int main() {
    int n;
    int x;
    // pre-conditions
    x = 0;
    // loop body
    while (x < n) {
        x = x + 1;
    }
    // post-condition
    if (n >= 0) {
        assert(x == n);
    }
}
```

Limitation 1: previous guesses are not effectively utilized. Existing loop invariant inference methods that are within the guess-and-check framework iteratively generate a candidate invariant and check it; if the check fails, the candidate invariant is discarded and the next candidate invariant will be generated. In practice, we observe that although the current invariant may not be valid, at least some of its clauses are likely parts of the correct (to-be-found) invariant. Unfortunately, such clauses are not well-utilized by the existing methods.

To illustrate this limitation, we examine the inference process of Code2Inv [41] for the program in Listing 1. Code2Inv is a learning-based method for loop invariant inference. It follows the general guess-and-check framework where reinforcement learning (RL) is adopted to guess the invariants. Specifically, Code2Inv generates candidate invariants based on BNF expansion rules chosen by a deep RL model, and then verifies these invariants against conditions using SMT solvers. The RL model enables Code2Inv to adapt its strategies for different programs, allowing it to learn from its inference attempts and improve over time.

For the given example in Listing 1, Code2Inv generates numerous expressions, and finally obtains the correct loop invariant

$$((n \geq (x+0) \ \vee \ x=(0-0))),$$

which is the 225th expression generated during the inference process. Interestingly, clauses such as $(x \leq n)$ and $(x=0)$, which are equivalent to the two clauses of the finally accepted loop invariant, appeared much earlier in this process, e.g., in the 12th expression

$$((n=1) \ \wedge \ (x \geq n \ \vee \ n > (1-n)) \ \wedge \ (x \leq n)),$$

and the 13th expression

$$((x=(x+0)) \ \wedge \ (x=0)).$$

This observation indicates that Code2Inv generates all the crucial components of the desired loop invariant early on, but unfortunately does not use them effectively. Similar limitations also exist for other methods. For example, CLN2INV [36] and G-CLN [45] generate candidate invariants and repeatedly test them, and the generated invariant is discarded and no longer contributes to the inference process, once it fails the SMT solver check.

Limitation 2: complex symbolic knowledge is not effectively handled. Another limitation of existing learning-based methods lies in their handling of complex expressions with logical connectives [29]. Although some efforts have been made in this direction, insofar the results have not been satisfactory. For example, existing methods for extracting logical rules from general neural architectures lack sufficient precision [2], and inductive logic learning lacks sufficient expressiveness for use in verification [14]. Moreover, while CLN2INV [36] has shown impressive results, its ability to express complex logical structures is limited. Specifically, when faced with programs requiring intricate logical relations or nonlinear patterns, the model struggles to capture these complexities effectively. It has been reported that CLN2INV cannot solve any problems on the nonlinear benchmark of LIPuS [46]. Some other learning-based approaches (e.g., Code2Inv) are even worse in this regard. Similarly, approaches based on reinforcement learning cannot handle logical connectives directly, but infer logical connectives through trial-and-error, which is of low efficiency.

To illustrate this limitation, we show some generated expressions from Code2Inv for the program in Listing 1 as follows,

$$((x=0) \ \wedge \ (x=(1-x)) \ \wedge \ (n>0)),$$
$$((n=1) \ \wedge \ (x=(n+n)) \ \wedge \ (n \leq (0-n))),$$

which are all unsatisfiable given the logic conjunction is introduced.

LLMs unfortunately exhibit similar issues. For instance, we directly query GPT-4o mini whether $(n \geq x \wedge x=0)$ is the loop invariant of the program. However, it incorrectly denies $(x=0)$ as a part of the loop invariant, and returns “the loop invariant should be $(n \geq x)$”. This shows that LLMs do not fully grasp the concept of loop invariants and cannot strictly enforce the three constraints of loop invariants. In this case, while the LLM recognizes part of the invariant, it fails to account for the complete logical structure, highlighting its limitations in handling precise logical reasoning.

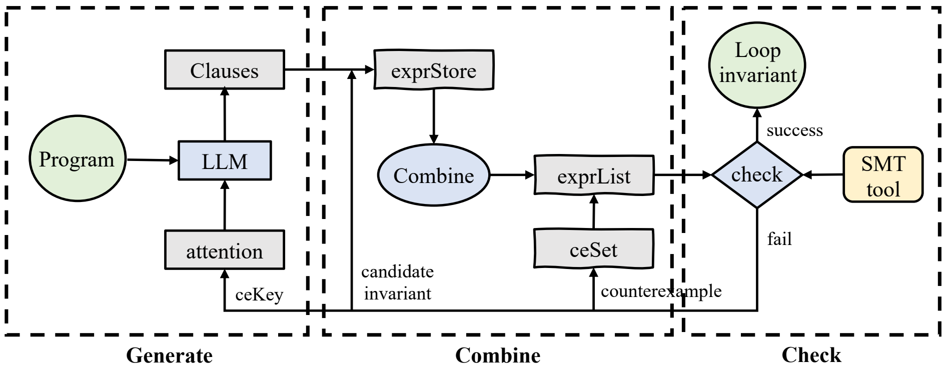

Fig. 1. An overview of Clause2Inv.

Our framework. The above limitations motivate us to explore the question: *Can we use learning-based methods to generate clauses first, and then combine them with logic connectives to derive a valid loop invariant?*

To this end, we propose to refine the *guess-and-check* framework into *generate-combine-check*, where the learning-based methods are devoted to the easier task of generating clauses instead of the complete invariants, and manually designed rules are used to combine the clauses into loop invariants. This design effectively addresses the two previously mentioned limitations.

- The previous guesses from the generating step can be maintained and reused. Such guesses can be further combined to form the future loop invariants.
- It does not require learning-based methods to handle complex concepts such as logical connectives or loop invariants, which are a better fit for symbolic tools such as manually defined rules or SMT solvers.

Another advantage of the *generate-combine-check* framework is that it can be easily adapted to improve the performance of existing learning-based loop invariant inference methods by splitting their previous guesses into clauses and maintaining them for later use.

3 The Proposed Approach

In this section, we present Clause2Inv, a loop invariant inference approach under the *generate-combine-check* framework. We shall first present an overview, then describe the details of its key components, followed by how it can be integrated with the existing methods.

3.1 Overview

Figure 1 presents an overview of our approach, which consists of the following two components.

(1) *Clause generation.* As mentioned before, the key of our *generate-combine-check* framework is to generate the clauses rather than the loop invariants directly. Specifically, we leverage LLMs and cast the clause generation task into an assertion generation task. In the prompt to LLMs, we deliberately ask LLMs not to use logical connectives in the generated expressions. We also design a feedback mechanism that allows LLMs to pay more attention to certain parts of the program (e.g., pre-conditions, loop body, and post-conditions), according to which constraint has been violated by the previous candidate loop invariant. For example, if the counterexample associated with the previous candidate loop invariant is returned due to a violation of the *prev* constraint (i.e., pre-conditions should imply the candidate invariant), we

will ask LLM to pay more attention to the pre-conditions during the next clause generation process.

(2) *Clause combination.* After preparing the clauses, we proceed to generate candidate invariants by combining the clauses with logical connectives $\wedge$ and $\vee$. Specifically, we maintain three data structures in this stage: a set `exprStore` storing historical clauses, a set `ceSet` storing all the previous counterexamples, and a list `exprList` keeping expressions that can pass all the counterexamples in `ceSet`. The clauses in `exprStore` are combined to find expressions that can pass all the current counterexamples. To increase the success rate of generated expressions, we also store the failed candidate invariants in `exprStore` as they have high potential to be parts of the correct loop invariant. For the expressions in `exprList`, we select the most concise one as the candidate invariant and check it by an SMT solver. If the check passes, a correct loop invariant is found. Otherwise, a new counterexample will be returned and `exprList` will be updated against this new counterexample. The specific constraint that has been violated (i.e., 'ceKey' in the figure) will also be returned to the clause generation component.

The above process continues until the desired loop invariant is derived or a timeout occurs.

3.2 LLM-based Clause Generation

We now elaborate on the LLM-based clause generator utilized in Clause2Inv. Note that, we do not prompt LLMs to directly generate the complete loop invariant; instead, we convert the task into an assertion generation task and intentionally ask LLMs not to contain logical connectives such as $\wedge$ and $\vee$ in the returned expressions. All the generated clauses will be maintained in a set for clause combination.

Prompt 1: Clause Generation Prompt.

```
### Your Task ###
I will provide you with a program.
Your task is to generate assertions based on the program's execution.
Please split the generated assertions and return them in a list, ordered from the most
generalizable to the least generalizable for {attention}.

### Notes ###
1. Try to find more complex assertions involving multiple variables with longer expressions.
2. Prioritize finding generalizable assertions that hold true across a wider range of situations.
3. Limit operators in the assertions to "==", "!=", "<", ">", "<=", ">=".
4. Avoid using logical operators like "&&", "||", "==>", "->", "if", "=>", "or", and "and".

### Program ###
{program}
```

The prompt template is shown in Prompt 1. We first describe the task, asking LLMs to generate assertions, with special "attention" paid to the three conditions mentioned in Section 2.1. Initially, we set "attention" to "pre-conditions, loop body, and post-conditions". When a counterexample is returned from the check step, this "attention" is set to one of the three conditions (i.e., "pre-conditions", "loop body", and "post-conditions") depending on which condition is violated.

Using "attention" is essential here as it encourages LLMs to generate all the required clauses in a correct loop invariant. If we simply ask LLMs to generate assertions for the program, they tend to generate ones for the loop body only. For instance, for the program shown in Listing 1, LLMs tend to generate clauses like $(n \geq x)$, $(n > x)$, and $(x \geq 0)$. However, they are less likely to generate $(x = 0)$ which is the clause we need for both pre-conditions and post-conditions. In contrast, if we ask LLMs to pay more attention to specific parts such as pre-conditions, clauses like $(x = 0)$ will have a greater chance of being generated because these clauses are more generalizable for those corresponding parts.

After describing the task, we list a few notes for LLMs. Specifically, the first note asks LLMs to generate more complex assertions involving multiple variables with longer expressions, in case LLMs may generate trivial expressions. The second note asks LLMs to generate generalizable assertions that hold true across a wide range of situations. Assertions akin to this have better potential to be the final clause of the loop invariant. The third note limits the comparison operators that LLMs can use and the fourth note asks LLMs not to use logical connectives. The "program" in this prompt will be replaced with the concrete program.

Example. We revisit the example in Listing 1. If we directly prompt the LLM to infer the loop invariant, it will not include $(x = 0)$ as a part of the final loop invariant and thus cannot infer the loop invariant within 10 minutes. In fact, even though we give LLM the correct loop invariant and ask whether it is the correct one, interestingly the LLM tends to deny its correctness.

In contrast, with our clause generator, the LLM initially returns a clause list that includes $(x \geq 0)$, $(x \leq n), \cdots, (x \neq 0)$. We then identify $(x \leq n)$ as the best candidate invariant, but it is rejected by the SMT solver as the pre-condition is violated. At this point, we ask the LLM to focus more on the pre-conditions in the prompt, and the clause list including $(x = 0)$ is returned in the *third* iteration, which can be further combined with $(x \leq n)$ to form the final loop invariant. The whole process for this example takes only 10.8 seconds.

3.3 Counterexample-driven Clause Combination

We next explain how we combine the clauses in `exprStore` into expressions in `exprList` mentioned in the overview of the approach in Section 3.1. Recall that `exprList` maintains the expressions that can pass all the existing counterexamples in `ceSet`.

3.3.1 Combination in exprStore. Each time we query LLMs, they will return a set of new clauses. For these new clauses, we first directly append them into `exprList`. The intuition is that a single clause itself may be the correct loop invariant. Next, we will combine each new clause with the old clauses in `exprStore`. Specifically, for a new clause c_{new} and an old clause c_{old} in `exprStore`, we combine two more clauses as $c_{old} \land c_{new}$ and $c_{old} \lor c_{new}$, which are then appended to `exprList`.

The rationale behind this combination is as follows. We combine new clauses with old ones as their attention may be concentrated on different program parts. More importantly, note that the previously failed candidate loop invariant is also stored in `exprStore` as mentioned in Section 3.1. Therefore, we can directly refine it by exploring the potential of the new clauses, especially since we have deliberately asked LLMs to generate clauses with the given attention (i.e., the reason why the previous candidate invariant failed). For example, if the previous candidate invariant fails due to the pre-condition is not satisfied, we will ask LLMs to generate new clauses with special attention paid to the pre-condition. Therefore, combining these new clauses with the failed candidate invariant will have a higher chance of obtaining the correct loop invariant.

Algorithm 1: The Clause2Inv Algorithm

```
Input: Program information program
Output: Loop invariant inv
Function Clause2Inv(program):
    exprStore ← ∅;
    ceSet ← ∅;
    inv ← None;
    attention ← "pre-conditions, loop body, and post-conditions";
    while inv is None do
        clauses ← LLMClauseGenerate(program, attention);
        inv, attention ← combineClauses(exprStore, clauses, ceSet);
    end
    return inv
end
```

In addition to combining failed candidate invariants with new clauses, we also *refine* them immediately after their failed check. That is, we further combine each failed candidate invariant with existing expressions in exprStore. Specifically, if the candidate invariant fails the pre-condition, meaning that it is too strict to be implied by the pre-condition, it will be relaxed through a combination with expressions using $\lor$. Similarly, $\land$ is used if the post-condition is violated. For the inductiveness constraint, we use both $\land$ and $\lor$. Such a refinement step is essential. For example, for loops with multiple phases such as “if-else” control structures, their loop invariants are difficult to infer due to the complexity of using disjunction $\lor$ to combine multiple expressions corresponding to different paths [25, 34, 45]. In contrast, our refinement step explicitly handles such cases by deliberately combining existing expressions.

3.3.2 exprList Update. Whenever a candidate loop invariant fails the SMT check, exprList is updated based on the newly returned counterexample. Specifically, we check each expression in exprList against this new counterexample and discard the ones that cannot pass it. For the remaining expressions, we sort them according to their conciseness (i.e., the expression length), where more concise ones are prioritized because they tend to be more generalizable.

While exprList is not empty meaning that expressions in it can still pass all the previous counterexamples, we iterate the above process by selecting the most concise expression as the candidate invariant and checking it in SMT solvers. The iteration ends when this candidate invariant passes the check, i.e., a valid loop invariant is found. Otherwise, a new counterexample is returned and the exprList is updated. If the exprList is empty, we will turn to the clause generator and ask the LLMs for new clauses.

3.4 The Overall Algorithm

The overall algorithm of Clause2Inv is summarized in Alg. 1. Given the input program, Clause2Inv first initializes the data structures. During the iterations, Clause2Inv iteratively asks LLMs to generate clauses with the current attention, which is initialized with three conditions and later updated based on the returned results of procedure combineClauses(). Then, Clause2Inv passes the current exprStore set, the new clauses generated by LLMs and the current ceSet to combineClauses(). This iteration terminates when a correct loop invariant has been found or a timeout occurs.

Algorithm 2: Procedure `combineClauses`

```
   Input: Expression set exprStore, newly generated clauses clauses, and counterexample
          set ceSet
   Output: Candidate invariant inv
1  Function combineClauses(exprStore, clauses, ceSet):
2      exprList ← ∅;
3      foreach clause in clauses do
4          exprList.append(clause);
5          foreach expr in exprStore do
6              exprList.append(andCombine(expr, clause));
7              exprList.append(orCombine(expr, clause));
8          end
9          exprStore.add(clause);
10     end
11     exprListUpdate(exprList, ceSet);
12     while not exprList.empty() do
13         candidateInv ← exprList[0];
14         exprStore.add(candidateInv);
15         result, ceKey, ce ← SMTSolver.check(candidateInv);
16         ceSet.add(ce);
17         if result is True then
18             return candidateInv
19         end
20         else
21             foreach expr in exprStore do
22                 if ceKey != "pre" then
23                     exprList.append(andCombine(expr, candidateInv));
24                 end
25                 if ceKey != "post" then
26                     exprList.append(orCombine(expr, candidateInv));
27                 end
28             end
29         end
30         exprListUpdate(exprList, ceSet);
31     end
32     return None, ceKey;
33 end
```

The detailed algorithm of procedure `combineclauses()` is summarized in Alg. 2. We first combine new clauses with old clauses in `exprStore` (Lines 3-10), and update `exprList` (Line 11). Then, the first expression in `exprList` is selected as the candidate invariant and checked by an SMT solver (Lines 13-16). If the check passes, a valid loop invariant is found (Lines 17-19); otherwise, we immediately refine the candidate invariant based on which constraint has been violated (Lines 20-29). We then update `exprList` again and continue the iteration as long as `exprList` is not

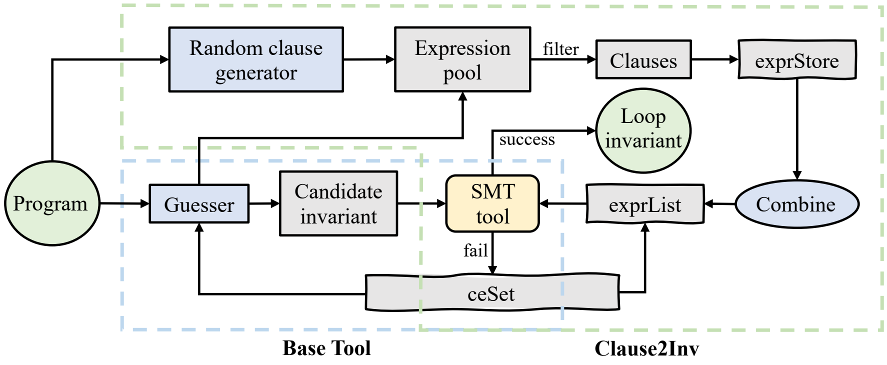

Fig. 2. Integration of Clause2Inv with other tools.

empty. If `exprList` becomes empty, procedure `combineclauses()` terminates and new clauses with be obtained through the clause generator.

3.5 Integration with Other Tools

One advantage of Clause2Inv is that it can also be integrated with various existing tools built upon the *guess-and-check* framework. That is, we can replace the LLM-based clause generator by (1) gathering guessed invariants from existing tools and (2) splitting them into clauses, as illustrated in Figure 2. This compatibility extends to several prior loop invariant inference tools, such as Code2Inv [41], LIPuS [46], CLN2INV [36], and G-CLN [45], among others. Such integration is non-invasive, and could further improve the performance of existing tools by better leveraging the previous guesses.

As shown in Figure 2, for a given base tool, we gather all the guessed expressions, decompose them into clauses and filter out useless ones (e.g., tautologies and contradictions). The remaining clauses are then combined based on our `combineClauses` procedure (cf. Algorithm 2). The base tool and Clause2Inv share the same counterexample set. Note that, more technically, we also use a white-box random clause generator which addresses the issue when the base tool generates insufficient clauses for combination. That is, while waiting for the guess step of the base tool, we invoke the random clause generator to produce clauses, ensuring the combination procedure proceeds uninterrupted. This random clause generator is based on the template shown in LIPuS [46], where the constants and variables are randomly chosen from the input program.

4 Evaluation

In this section, we present the experimental results, which are designed to answer the following three research questions (RQs).

RQ1. How does Clause2Inv perform compared with the existing baselines?
RQ2. To what extent can the proposed *generate-combine-check* framework enhance existing tools?
RQ3. How do different design choices affect the performance of Clause2Inv?

4.1 Setup

Benchmark. To thoroughly evaluate different loop invariant inference methods, we curate a dataset of 316 linear benchmark problems and 50 nonlinear benchmark problems. The linear dataset includes 133 problems collected by Code2Inv [41], commonly used in previous evaluations.[1] Additionally, we manually craft 84 problems from the 2019 SyGuS competition and 99 problems from the 2024 SV-COMP benchmarks. For the nonlinear benchmark, we collect 30 problems from LIPuS [46], and additionally craft 20 problems from the 2024 SV-COMP benchmarks. Each problem consists of a C code snippet containing a loop, potentially with nested if-then-else structures, and the corresponding SMT-LIB2 files.

Baselines. To evaluate our approach, we compare it with the following baselines.

- LoopInvGen [34] searches a conjunctive normal form over predicates synthesized by an underlying engine, and it is a heuristic search-based system.
- CVC5 [3] is an efficient open-source SMT solver, which can be used to prove the satisfiability of first-order formulas with respect to a variety of useful background theories. It provides a Syntax-Guided Synthesis (SyGuS) engine to synthesize invariants.
- Code2Inv [41] is a standard *guess-and-check* approach, which generates candidate invariants according to BNF (Backus-Naur form) expansion rules selected by a deep RL model and checks the invariants against the verification conditions with an SMT solver.
- LIPuS [46] is an extension of Code2Inv, introducing a two-dimensional reward mechanism to guide reinforcement learning and integrating it with a template iteration method. For both linear and nonlinear loop invariants its performance is good as reported.
- CLN2INV [36] is a deep learning method that builds a "bijection" between logic expressions and neural networks. It infers the loop invariant by training a neural network (i.e., logic expression) to fit in the program traces.
- G-CLN [45] builds upon CLN2INV by incorporating gating mechanisms and other enhancements, enabling it to learn more complicated loop invariants. It is good at inferring nonlinear loop invariants.
- SymInfer [33], or DIG, infers loop invariants by utilizing symbolic states generated by symbolic executions. Different from other approaches, it aims at inferring non-trivial invariants for any location labeled at the source code. SymInfer is good at inferring nonlinear loop invariants, but its performance is poor for linear ones [46].

Among these baselines, LoopInvGen, CVC5, and SymInfer are traditional symbolic methods, and the rest are learning-based methods. For the linear benchmark, we compare all the baselines except SymInfer. For nonlinear benchmark, we compare Clause2Inv with LIPuS, G-CLN, and SymInfer, which are all specially designed for inferring nonlinear loop invariants.

Evaluation metrics. To evaluate the efficacy of different methods in inferring loop invariants, we adopt the following performance indicators: 1) the number of successfully generated loop invariants; 2) the number of SMT solver queries during the inference process; and 3) the time used to infer the loop invariant. The first metric relates to effectiveness, while the latter two metrics pertain to efficiency.[2]

Implementations. Our experiments evaluate all methods under identical settings using the same hardware and benchmark. For the LLM used in Clause2Inv, we adopt GPT-4o mini (gpt-4o-mini) by default. Each method is subject to a timeout of 600 seconds per problem. The experiments are conducted on a GPU server equipped with an Intel Core i9-13900K@3.00GHz CPU, 48GB RAM, and a GeForce RTX 4090 GPU.

[1]Ryan et al. [36] identified that 9 out of them were unsolvable. During our review of the benchmarks, we discovered an additional 6 unsolvable cases. We have corrected these problematic cases based on their original SMT-LIB2 files.
[2]Note that we compute the latter two metrics based on the successfully solved problems. In other words, these two metrics are somewhat unfriendly to methods that can solve more challenging problems.

Table 1. Performance comparison of different loop invariant inference methods on linear invariant generation. The proposed Clause2Inv can solve 312 out of 316 problems, which is at least 93 more than the competitors.

Methods	# Solved Benchmarks	# Avg. Queries	Avg. Time (s)
	Total 316 (133/84/99)		
LoopInvGen	218 (107/49/62)	5.6	17.8
CVC5	207 (107/46/54)	13.9	5.5
Code2Inv	210 (110/47/53)	10.9	137.6
LIPuS	159 (124/18/17)	3.7	48.4
CLN2INV	211 (124/35/52)	23.9	0.6
G-CLN	219 (116/45/58)	20.1	2.6
Clause2Inv	**312 (132/82/98)**	5.8	35.2

4.2 RQ1. Performance Comparison

Benchmark of linear loop invariant problems. We first compare different methods on the problems where only linear invariants are required. The results are presented in Table 1 (the detailed performance on the three data sources (i.e., Code2Inv, SyGuS 2019 and SV-COMP 2024) is also shown in parentheses). The best performance is highlighted in boldface.

It can be observed that Clause2Inv significantly outperforms the competitors in terms of the number of solved benchmark problems. With GPT-4o mini used in the clause generator, Clause2Inv solves 93 more problems than the best competitor, G-CLN. As for the efficiency metrics, the number of SMT solver queries of Clause2Inv is relatively low compared to the other approaches. The average solving time of Clause2Inv is higher than several baselines (but is still lower than the two RL-based methods). The reasons are two-fold: First, the queries to the LLM require time. Second, the additional solved problems of Clause2Inv are usually more challenging, thus increasing the average time.

We next compare the performance of different methods on the three data sources. It can be observed that all the competitors perform relatively well on the original 133 benchmark problems (solving 107—124 problems), but not as effectively on the additional two benchmark sources (solving at most 58.3% and 62.6% of the problems respectively). As tools using more traditional methods (such as LoopInvGen and CVC5) also perform relatively worse on the new problems, it is reasonable to conclude that the new problems may indeed be more difficult. In contrast, we observe that Clause2Inv performs relatively stable (missing one, two, one solution across the three data sources, respectively), showing its strong generalizability. The reasons may be two-fold. First, Clause2Inv adopts off-the-shelf LLMs, which exhibit sufficient generalization capabilities and do not overfit to the previous benchmarks. Second, our clause combinator can better handle the previous guesses as well as the logic connectives, both of which are critical for loop invariant inference.

Benchmark of nonlinear invariant problems. For this benchmark, we compare Clause2Inv with three state-of-the-art approaches that demonstrate impressive performance on nonlinear problems, i.e., LIPuS [46], G-CLN [45], and SymInfer (also known as DIG) [33]. The results are shown in Table 2. Note that, different from other tools, SymInfer aims at inferring non-trivial invariants for any location labeled at the source code with concrete program data traces, and does not need to resort to SMT solvers to check whether its result satisfies the three constraints of loop invariants. Therefore, we do not compare SMT solver queries here.

Table 2. Performance comparison of different loop invariant inference methods on the nonlinear benchmark. Clause2Inv can solve at least 16 more problems.

Methods	# Solved Benchmarks	Avg. Time (s)
	Total 50 (30/20)	
G-CLN	21 (15/ 6)	10.3
SymInfer	28 (15/13)	16.7
LIPuS	25 (25/ 0)	183.5
Clause2Inv	**44 (28/16)**	64.9

Our approach Clause2Inv demonstrates better performance not only on the initial 30 problems, but also on the additional 20 problems. For SymInfer, its performance is consistent on the additional 20 problems. For learning-based methods G-CLN and LIPuS, their performance significantly decreases on the additional problems. One possible explanation for this phenomenon is the overfitting issue in learning-based methods.

The performance improvement of Clause2Inv in nonlinear cases is attributed to several reasons. For traditional template-based approaches, inferring a nonlinear loop invariant involves a significantly larger solution space than the linear cases, causing significant or even unaffordable burdens to such methods. Similar issues also exist for learning-based methods that utilize an invariant template, such as LIPuS [46], Code2Inv [41], and ICE-DT [20]. However, our approach does not rely on the invariant template, but instead leverages LLMs to generate clauses, which effectively narrows the solution space for the clause combinator. Furthermore, the clause combinator is designed to better handle the historical clauses and logic connectives, thus gradually converging to the correct loop invariant.

4.3 RQ2. Integration with Other Tools

Basic integration. We evaluate the integration capability of Clause2Inv. The results with tools such as Code2Inv [41], LIPuS [46], CLN2INV [36], and G-CLN [45] as the base tools are presented in Table 3. For simplicity, we focus on the linear benchmark in this experiment. For a fair comparison, we also present the average solving time of Clause2Inv on the initially solved problems by the base tool (indicated in parentheses in the last column). Note that the solved problem set of Clause2Inv is a superset of that of the base tool.

We can observe that Clause2Inv, when integrated with the existing base tools, can significantly improve the base tools' performance. For example, the classic Code2Inv solved 210 problems with an average solving time of 137.6 seconds. When integrated with Clause2Inv, it solves 252 problems with an average solving time of 17.8 seconds. Similarly, it also solves 82, 53, and 49 more problems for LIPuS, CLN2INV, and G-CLN, respectively. Meanwhile, when considering the average solving time on the commonly-solved problems, Clause2Inv is also faster in all cases.

Comparing different methods, we observe that 'CLN2INV + Clause2Inv' and 'G-CLN + Clause2Inv' are relatively better in both the number of solved problems and the average solving time. We attribute this improvement to the higher quality of their generated clauses, which makes the clause combination of Clause2Inv more effective. Specifically, CLN2INV and G-CLN generate invariants based on the program execution traces instead of static analysis as adopted by Code2Inv and LIPuS.

Integrating multiple tools. Clause2Inv is also flexible for integrating multiple existing loop invariant inference tools, by simply aggregating all the generated loop invariant expressions into the expression pool, as shown in Figure 2. To illustrate this, we combine CLN2INV and G-CLN

Table 3. The integration capability of Clause2Inv. It can further enhance the performance of various existing tools built upon the guess-and-check framework.

Methods	# Solved Benchmarks	# Queries	Time (both solved)
Code2Inv	210 (110/47/53)	10.9	137.6
Code2Inv + Clause2Inv	252 (121/58/73)	8.0	17.8 (13.0)
LIPuS	159 (124/18/17)	3.7	48.4
LIPuS + Clause2Inv	241 (124/48/69)	6.4	6.0 (5.0)
CLN2INV	211 (124/35/52)	23.9	0.6
CLN2INV + Clause2Inv	264 (131/60/73)	14.4	1.1 (0.5)
G-CLN	219 (116/45/58)	20.1	2.6
G-CLN + Clause2Inv	268 (131/61/76)	6.6	2.2 (1.9)
CLN2INV+G-CLN	227 (124/45/58)	22.9	1.5
CLN2INV+G-CLN + Clause2Inv	287 (131/72/84)	13.4	9.5 (1.2)

Table 4. Results of ablation study. The performance of Clause2Inv is stable with different LLMs, and both designs of our clause generator and clause combinator are essential.

Models	# Solved Benchmarks	# Avg. Queries	Avg. Time (s)
	Total (133/84/99)		
GPT-4o mini	312 (132/82/98)	5.8	35.2
GPT-3.5-Turbo	308 (131/80/97)	6.3	51.1
Deepseek	301 (132/75/94)	5.8	54.1
Random generator	227 (124/47/56)	11.4	67.4
LLM only	225 (92/53/80)	2.3	61.0
Without refinement	285 (122/74/89)	5.0	34.8

together as the base tool to output loop invariant expressions, and further integrate them with Clause2Inv. The result is shown in the last two rows of Table 3. Here, "CLN2INV+G-CLN" means that we run the two tools separately, with the better result being reported. After integrating the two tools with Clause2Inv, 60 more problems are solved (from 227 to 287), whose improvement is even greater than the improvement of either CLN2INV or G-CLN integrated with Clause2Inv.

4.4 RQ3. Ablation Study

Performance under different LLMs. For the ablation study, we first test different LLMs. Specifically, we select an alternative model from OpenAI and another open-source model from DeepSeek. The results on the linear benchmark are given in Table 4. It can be observed that Clause2Inv's performance is stable across different LLMs. This result may come as a surprise as there are obvious gaps between GPT-3.5-Turbo and GPT-4 as well as DeepSeek and GPT-4 [7, 25]. We attribute this stability to the *generate-combine-check* framework, where LLMs do not need to infer the complete loop invariant which is, we posit, beyond their current capability. Instead, they only need to generate a set of clauses describing the properties of the program, which is the strength of LLMs.

Random clause generator. We next evaluate the case when LLMs are not used in the clause generator. Specifically, we substitute the LLMs with the random clause generator discussed in Figure 2. The result, denoted by "random generator", is shown in Table 4. It can be observed that, even with a random white-box clause generator, Clause2Inv can solve 227 problems, which is even slightly better than the existing tools, as shown in Table 1. This result suggests the effectiveness of our clause combinator and, more broadly, the strength of the proposed *generate-combine-check* framework for loop invariant inference.

LLM only. To verify the usefulness of our clause combinator and the ability of inferring loop invariants directly for LLMs, we craft a prompt asking the LLM to directly generate loop invariants (with logical connectives), and disable the clause combination part. The result is also shown in Table 4. Using GPT-4o mini, the number of solved problems decreases from 312 to 225. This shows that LLMs alone cannot sufficiently handle complex symbolic knowledge, necessitating a symbolic combinator to achieve better performance.

In Table 4, we also observe that "LLM only" performs closely to "random generator", but exhibits discrepancies across different data sources. Specifically, random generator outperforms LLM only on the original 133 problems, but underperforms on the new 183 problems. The reasons are as follows. The loop invariants of the original 133 problems tend to be complex combinations of simple clauses, while the clauses of the 183 problems are more complex with more operators (e.g., mod) and variables involved. Moreover, our clause combinator plays an important role in the 133 problems and LLMs may perform better in the latter case.

Without the refinement step. We also conduct an experiment to validate the effectiveness of the refinement step (i.e., Lines 20-29 in Alg. 2) in the clause combinator. We remove this part from the algorithm and run the same experiment again. The result is in the last row of Table 4, where one can see that the number of solved problems decreases from 312 to 285. Although this decrease is not very substantial, it significantly weakens our approach especially for complex loops with multiple phases, as will be shown in a case study in Section 5.1.

5 Discussion

5.1 Case Studies

Case study 1: a linear loop invariant. We first showcase a complex linear benchmark problem that can only be solved by our Clause2Inv in Listing 2.

Listing 2. A problem with linear loop invariant

```
int main(){
    int i;
    int j;
    int n;
    int k;
    int b;

    n = 0;
    b = 0;
    assume(k > 0);
    assume(k < 20000001);
    assume(i == j);

    while(n < (2 * k)){
        n = n + 1;
        if(b == 1){
            b = 0;
            i = i + 1;
        }
        else{
            b = 1;
            j = j + 1;
        }
    }

    if(n >= (2 * k))
        assert(i == j);
}
```

Clause2Inv successfully produces a valid loop invariant for the program as

$$((n \ge 0 \wedge (\underline{(b = (n \bmod 2) \wedge (i = j \wedge b = 0))} \vee \underline{(b = (n \bmod 2) \wedge (i \ge j - 1 \wedge (k > 0 \wedge (n < 2 \cdot k \wedge n \bmod 2 = (i + j) \bmod 2)))))})) \wedge j \ge i),$$

which contains 10 clauses connected by 9 logic connectives, and takes 482.9 seconds and 19 SMT solver queries. In this invariant, $n \ge 0$ and $j \ge i$ rigorously capture the behavior of the program. The two underlined sub-expressions correspond to the two paths of the "if-else" control structure, respectively. By observing the inference process of this loop invariant, we find that Clause2Inv first infers the sub-expression corresponding to the $b = 0$ condition, which is relatively simpler, and then infers another sub-expression corresponding to the $b = 1$ condition, which is more complicated. After preparing the two sub-expressions, the combinator merges them with $\vee$ in the refinement step. We note that this problem cannot be solved if the refinement step is excluded.

Case study 2: a nonlinear loop invariant. We also show a complex nonlinear case in Listing 3, which is the standard Bresenham's line drawing algorithm.

Listing 3. A problem with nonlinear loop invariant

```
int main() {
    int X, Y;
    int v, x, y;

    assume(X > 0);
    assume(Y > 0);
    assume(X >= Y);
    v = 2 * Y - X;
    y = 0;
    x = 0;

    while (x <= X) {
        if (v < 0) {
            v = v + 2 * Y;
        } else {
            v = v + 2 * (Y - X);
            y++;
        }
        x++;

    }

    assert(2*Y*x - 2*x*y - X \
    + 2*Y - v + 2*y == 0);
}
```

For this program, Clause2Inv generates a valid loop invariant, i.e.,

$$y \ge 0 \wedge (2Y \cdot x \ge 2x \cdot y + X - 2Y + v - 2y \wedge (X > 0 \wedge Y > 0 \wedge v = 2Y - X + 2Y \cdot x - 2X \cdot y)).$$

This is very challenging even for human beings. For example, the $v = 2Y - X + 2Y \cdot x - 2X \cdot y$ sub-expression is hard to infer even for someone familiar with Bresenham's line drawing algorithm, let alone the combination of the multiple sub-expressions to form a valid loop invariant.

5.2 Comparison with LaM4Inv

A parallel piece of work is LaM4Inv [43]. Our Clause2Inv differs from LaM4Inv in two aspects. First, LaM4Inv asks LLMs to generate complete invariants and employs bounded model checking to eliminate unnecessary clauses. In contrast, Clause2Inv directly constructs a pool of clauses and infers the loop invariant from scratch based on the clauses. As a result, while LaM4Inv may be hindered due to the scalability of bounded model checking, Clause2Inv avoids this issue by using a clause combination algorithm.

Second, LaM4Inv directly combines the existing invariants with conjunctions to form the final invariant, while Clause2Inv uses a clause combination algorithm, which can better handle disjunctive normal forms. As illustrated in LaM4Inv [43], it fails to generate the invariant for the case in Listing 4. In contrast, Clause2Inv solves this problem in 29.1 seconds, which can be attributed to the clause combination algorithm. The inferred loop invariant is:

$$(x \bmod 2 = y \bmod 2 \ \vee \ (x \leq 99 \ \wedge \ x \bmod 2 = 0)).$$

Listing 4. Case 2 in LaM4Inv

```
int main() {
    int x;
    int y;

    x = 0;

    while (x < 99) {
        if(y % 2 == 0){
            x = x + 2;
        }
        else{
            x = x + 1;
        }
    }

    assert((x % 2) == (y % 2));
}
}
```

5.3 Threats to Validity

We have identified three primary threats to the validity of our work. The first relates to the capabilities of SMT solvers. While SMT solvers are essential tools, they struggle with operations such as multiplication, division, and exponentiation. This limitation creates a bottleneck when solving nonlinear benchmarks. Existing research often restricts the number of multiplicative layers in nonlinear invariant inference and avoids using power operations where both the base and exponent are variables. Although we could generate some invariants that include expressions like "power(a, b)", SMT solvers face difficulties verifying such invariants. Furthermore, recent studies have shown that SMT solvers can sometimes produce incorrect results, which could potentially affect the verification results.

The second threat concerns the benchmark we used, which is limited in scope. Despite the expansion, the benchmark problems still cannot fully represent real-world scenarios. In future work, we aim to develop more robust benchmarks to better demonstrate the performance of our method with greater clarity and accuracy.

The third threat pertains to the potential data leakage associated with large language models. There remains uncertainty regarding whether LLMs have encountered the prior 133 benchmark problems, as these are open-sourced and may have been previously exposed to them. Furthermore, the sources of our new benchmark problems, the 2019 SyGuS competition and 2024 SV-COMP benchmarks, may also have been exposed to LLMs. Nonetheless, the new benchmark problems are manually crafted by us, effectively mitigating the data leakage concern. Additionally, the correct loop invariants are not included in the datasets.

6 Related Work

In this section, we provide a brief review of related work, categorized into traditional approaches and learning-based approaches.

Traditional approaches. Various loop invariant inference methods have been proposed in the history, which can be divided into several categories: constraint solving [9, 21], abstract interpretation [10, 11, 26], logical abduction [6, 13], model checking [22, 42], craig interpolation [24, 31], syntax-guided synthesis [3, 4], counterexample guided abstraction refinement [8, 48], recurrence analysis [16, 23, 27], etc. For example, Eldarica [22] is a model-checking-based Horn clauses solver. CVC4/CVC5 [3, 4] is an efficient open-source SMT solver, which provides a syntax-guided synthesis engine to synthesize invariants. These approaches tend to be stable for different application situations, but their performance is not as effective and scalable as learning-based methods.

Learning-based approaches. Recently, a variety of approaches follow the *guess-and-check* framework and utilize machine learning techniques to generate candidate invariants. Exemplar machine learning techniques utilized include decision tree [15, 19, 20, 35, 44], support vector machine [28, 40], PAC (probably approximately correct) learning [39], reinforcement learning [41, 46], continuous logic networks [36, 45]. For example, ICE-DT [20] uses decision trees to handle implication counterexamples, but requires experts to annotate program properties. Code2Inv [41] and LIPuS [46] utilize reinforcement learning to expand the grammar tree of an invariant template. CLN2INV [36] and G-CLN [45] design continuous logic networks and aim to build a "bijection" between logic expressions and neural networks. A few recent attempts have also explored the capability of LLMs in loop invariant inference. Kamath et al. [25] directly employ LLMs to find inductive loop invariants; Chakraborty et al. [7] further rank the outputs of LLMs; Liu et al. [30] propose a self-supervised learning paradigm to fine-tune LLMs for loop invariant inference. Very recently, Wu et al. [43] investigate a synergy of LLMs and bounded model checking for loop invariant inference.

These learning-based approaches all aim to directly infer the loop invariants, which usually include logical connectives that are difficult to handle for machine learning techniques. In contrast, we propose a *generate-combine-check* framework, leveraging learning-based approaches to generate clauses and manually designed rules to systematically combine clauses into candidate invariants.

7 Conclusion

In this paper, we propose a new loop invariant inference approach Clause2Inv, which is based on a new *generate-combine-check* framework. The key idea is to divide loop invariant inference into clause generation and clause combination, where the former is a much easier task than directly generating the loop invariant, and the latter can better handle the historical guesses and logic connectives. Evaluations on 316 linear benchmark problems and 50 nonlinear benchmark problems demonstrate the superior performance of Clause2Inv, surpassing the state-of-the-art methods by solving at least 93 and 16 more problems, respectively. Moreover, due to the flexibility of our *generate-combine-check* framework, Clause2Inv can be integrated with many previous loop invariant inference tools to further improve their performance.

8 Data Availability

To ensure transparency and facilitate the reproduction of our results, we make the augmented datasets as well as the experiment scripts and results publicly available via the link https://github.com/SoftWiser-group/Clause2Inv.

Acknowledgment

We would like to thank the authors of LIPuS for helping reproducing the results. This work is supported by the National Natural Science Foundation of China (Grants #62025202), the Frontier Technologies R&D Program of Jiangsu (BF2024059), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by oversea grants from the State Key Laboratory of Novel Software Technology, Nanjing University under Grant #KFKT2022A03 and #KFKT2023A04. Yuan Yao is the corresponding author.

References

[1] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. CoRR abs/1904.07146 (2019). *arXiv preprint arXiv:1904.07146* (2019).

[2] M Gethsiyal Augasta and T Kathirvalavakumar. 2012. Rule extraction from neural networks—a comparative study. In *International Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME-2012)*. IEEE, 404–408.

[3] 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.

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *International Conference on Computer Aided Verification (CAV)*. Springer, 171–177.

[5] Dirk Beyer. 2024. State of the art in software verification and witness validation: SV-COMP 2024. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 299–329.

[6] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. 2009. Bi-abductive resource invariant synthesis. In *Asian Symposium on Programming Languages and Systems*. Springer, 259–274.

[7] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP*. 9164–9175.

[8] 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.

[9] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. 2003. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification (CAV)*. Springer, 420–432.

[10] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. 269–282.

[11] 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)*. 84–96.

[12] 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.

[13] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. *Acm Sigplan Notices* 48, 10 (2013), 443–456.

[14] Richard Evans and Edward Grefenstette. 2018. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research* 61 (2018), 1–64.

[15] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[16] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 57–64.

[17] James H Fetzer. 1988. Program verification: The very idea. *Commun. ACM* 31, 9 (1988), 1048–1063.

[18] Yoav Freund and Robert E Schapire. 1998. Large margin classification using the perceptron algorithm. In *Proceedings of the eleventh annual conference on Computational learning theory*. 209–217.

[19] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification (CAV)*. Springer, 69–87.

[20] 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.

[21] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. 2009. From tests to proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 262–276.

[22] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–7.

[23] 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.

[24] 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.

[25] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *arXiv preprint arXiv:2311.07948* (2023).

[26] Michael Karr. 1976. Affine relationships among variables of a program. *Acta Informatica* 6, 2 (1976), 133–151.

[27] 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.

[28] 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.

[29] Zenan Li, Zehua Liu, Yuan Yao, Jingwei Xu, Taolue Chen, Xiaoxing Ma, and Jian Lü. 2023. Learning with Logical Constraints but without Shortcut Satisfaction. In *The Eleventh International Conference on Learning Representations*

(ICLR).

[30] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. 2023. Towards General Loop Invariant Generation via Coordinating Symbolic Execution and Large Language Models. *arXiv preprint arXiv:2311.10483* (2023).

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

[32] 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.

[33] ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. 2017. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 804–814.

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

[35] 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 (ESEC/FSE)*. 607–619.

[36] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *International Conference on Learning Representations (ICLR)*.

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

[38] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 574–592.

[39] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. 2013. Verification as learning geometric concepts. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer, 388–411.

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

[41] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems (NeurIPS)* 31 (2018).

[42] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. 2023. Global guidance for local generalization in model checking. *Formal Methods in System Design* (2023), 1–29.

[43] 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.

[44] 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 (ESEC/FSE)*. 111–122.

[45] 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 (PLDI)*. 106–120.

[46] 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 (ISSTA)*. 175–187.

[47] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *ACM SIGPLAN Notices* 53, 4 (2018), 707–721.

[48] He Zhu, Aditya V Nori, and Suresh Jagannathan. 2015. Learning refinement types. *ACM SIGPLAN Notices* 50, 9 (2015), 400–411.

[49] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 491–507.

Received 2024-10-31; accepted 2025-03-31