LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference

Guangyuan Wu
guangyuanwu@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Weining Cao
weiningcao@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Yuan Yao
y.yao@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Hengfeng Wei
hfwei@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

Taolue Chen
t.chen@bbk.ac.uk
School of Computing and Mathematical Sciences, Birkbeck, University of London
UK

Xiaoxing Ma
xxm@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University
China

ABSTRACT

Loop invariant inference, a key component in program verification, is a challenging task due to the inherent undecidability and complex loop behaviors in practice. Recently, machine learning based techniques have demonstrated impressive performance in generating loop invariants automatically. However, these methods highly rely on the labeled training data, and are intrinsically random and uncertain, leading to unstable performance. In this paper, we investigate a synergy of large language models (LLMs) and bounded model checking (BMC) to address these issues. The key observation is that, although LLMs may not be able to return the correct loop invariant in one response, they usually can provide all individual predicates of the correct loop invariant in multiple responses. To this end, we propose a "query-filter-reassemble" strategy, namely, we first leverage the language generation power of LLMs to produce a set of candidate invariants, where training data is not needed. Then, we employ BMC to identify valid predicates from these candidate invariants, which are assembled to produce new candidate invariants and checked by off-the-shelf SMT solvers. The feedback is incorporated into the prompt for the next round of LLM querying. We expand the existing benchmark of 133 programs to 316 programs, providing a more comprehensive testing ground. Experimental results demonstrate that our approach significantly outperforms the state-of-the-art techniques, successfully generating 309 loop invariants out of 316 cases, whereas the existing baseline methods are only able to tackle 219 programs at best. The code is publicly available at https://github.com/SoftWiser-group/LaM4Inv.git.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695014

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

loop invariant, program verification, large language model

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695014

1 INTRODUCTION

Loop invariant inference plays a central role in program verification. In a nutshell, a loop invariant is a proposition that holds before and after each iteration of the loop, serving as a cornerstone for deductive verification of loop programs. However, being undecidable in general, inferring loop invariants is a notoriously difficult problem which has been the subject of active research for over 40 years.

Historically, direct inference of loop invariants using pure logical methods has proven to be hard; such methods are normally based on fixpoint computation and/or abstract interpretation, which are not scalable in practice. Later on, several studies adopt template-based constraint-solving approaches, where typically polynomial invariants are considered in a pre-defined parametric form (e.g., polynomials in a fixed set of variables up to certain degrees) and then computer algebra tools (e.g., Gröbner basis, Nullstellensatz/-Positivstellensatz) are applied to synthesize the coefficients of the polynomial. This class of methods relies on a suitable form of templates which cannot be too simple (otherwise not producing useful invariants) or too complicated (otherwise cannot be handled by constraint solvers).

Recent work adopts the "*guess-and-check*" framework, where candidate invariants are iteratively generated (*guessing*) and verified (*checking*). This class of methods is heuristic in nature (hence no completeness guarantees) but is highly versatile. In particular, they

can accommodate data-driven approaches in the "guessing" step. Indeed, various machine learning techniques have been applied to generate candidate invariants including decision tree [20, 23, 51], reinforcement learning [46, 54], continuous logic networks [42, 52] and large language models (LLMs, [7, 28, 33, 50]).

Our work. In this paper, we propose a novel neuro-symbolic approach LaM4Inv (**L**LM **a**nd **M**odel checking for loop **Inv**ariant inference) to instantiate the aforementioned guessing step, leveraging the cutting-edge breakthrough in generative AI and traditional, established symbolic approaches from formal verification for loop invariant inference. Specifically, our work is motivated by two observations. First, recent progress in (Transformer-based) LLMs has demonstrated their remarkable capabilities in "understanding" programs; there is evidence that LLMs may generate loop invariants for programs that traditional methods failed to handle. Meanwhile, LLMs are, compared to previous machine learning methods which may require building up a neural network model from scratch and a tremendous volume of training data, considerably more cost-effective owing to their in-context learning capability. Second, our empirical studies show that, although LLMs may not return the correct loop invariant in one response, all the individual predicates/clauses of the correct loop invariants are usually included in multiple responses. Based on the these observations, we propose a "query-filter-reassemble" procedure to instantiate the "guessing" step. That is, we first query LLMs to obtain a set of candidate loop invariants, and then distill appropriate atomic predicates which can be further "reassembled" to produce new candidate loop invariants. As to the "checking" step, the candidate loop invariants are checked by off-the-shelf SMT solvers.

Two challenges to implement such a "query-filter-reassemble" procedure are: 1) to effectively extract knowledge from LLMs under the context loop invariant inference, and 2) to effectively identify the appropriate predicates from the LLM outputs. For the first challenge, we design prompts (cf. Section 3.2 for details) to query LLMs which include, among others, the feedback (e.g., counterexamples and the specific properties that the current candidate invariant violates) from the "checking" step. In this way, LLMs would have richer information to produce more accurate candidate loop invariants, thereby accelerating the entire inference process. For the second challenge, a technical question is to determine whether a predicate should be filtered out or remain to be reassembled later, by which we can maximize the chance of generating a valid loop invariant. To this end, we leverage a (symbolic) approach from formal verification, i.e., bounded model checking (BMC, [9]). In a nutshell, BMC is a symbolic model-checking technique which leverages SAT solvers. In its plain form, the loop is unrolled for a (pre-defined) fixed number of times producing a finite execution on which properties (encoded in predicates in this paper) are checked. In practice, BMC is mainly used for falsification, i.e., violations of properties, and we leverage its strong falsification capabilities to filter out incorrect predicates.

Essentially, the interaction between LLM and BMC gives rise to a closed-loop neuro-symbolic system. That is, the neural component (LLM) takes into account the feedback, which can be regarded as an implicit inductive bias when generating candidate loop invariants; the symbolic component (BMC) processes the output from the neural component, and reassembles a new candidate loop invariant; the new candidate is further verified with feedback further used in the next round of LLM querying.

Evaluation. To evaluate the proposed approach LaM4Inv in a more thorough way, we first expand the existing benchmark which consists of 133 loop invariant inference problems [46]. We include 84 problems from the 2019 SyGuS competition [1] and 99 problems from the 2024 SV-COMP benchmarks [5], giving rise to a new benchmark of 316 problems. Each problem consists of a C code snippet containing a loop with possibly nested if-then-else structures, and the corresponding SMT-LIB2 files. We compare LaM4Inv with 8 existing methods including LoopInvGen [38], CVC5 [3], Code2Inv [46], LIPUS [54], CLN2INV [42], G-CLN [52], ESBMC [21], and LEMUR [50]. The experimental results on the new benchmark show that LaM4Inv successfully generates 309 loop invariants out of 316 cases, which is at least 90 more than the existing competitors.

Additionally, we conduct a series of ablation studies, the results of which show that: 1) the carefully designed prompt that includes feedback information from SMT solvers, 2) the adoption of BMC as a filter mechanism to identify potentially correct predicates, and 3) the closed-loop synergy between LLM and BMC, all contribute to the performance enhancement of LaM4Inv.

In summary, the main contributions of this paper include:

- *Approach.* We propose a neuro-symbolic loop invariant inference approach LaM4Inv that synergizes large language models and bounded model checking. The experiments confirm that LaM4Inv advances the state-of-the-art baselines.
- *Dataset and Evaluation.* We expand the existing benchmark of loop invariant inference problems, curating a dataset double of the size of the existing benchmark. The proposed approach is thoroughly evaluated based on the new dataset.

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

2 PRELIMINARY AND MOTIVATION

In this section, we provide the background knowledge on loop invariant inference and present a motivating example.

2.1 Loop Invariant Inference

By well-known Hoare logic [25], for a given loop `while` B `do` S, the loop invariant inference problem aims to identify a loop invariant I that satisfies

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

where P is the *pre-condition*, Q is the *post-condition*, and S is the loop body with B as the loop condition. Essentially, Eq. (1) includes the following three requirements. Recall that a (program) state is a valuation of all variables of the program and an execution can be viewed as a sequence of transitions of program states.

- *Reachability.* The set of program states represented by the loop invariant should include all states that can be reached by the code

executed before the loop. This ensures that the loop invariant encompasses the pre-condition of the loop.
- *Inductiveness.* For each program state within the loop invariant, after each loop execution which leads to a new program state, this new state must also remain within the loop invariant. This requirement ensures that the invariant is maintained throughout the loop's execution, regardless of the number of iterations.
- *Provability.* The states within the loop invariant, upon meeting the exit conditions of the loop, must satisfy the property that we aim to verify. This condition ensures that the loop correctly achieves the intended outcomes and that the properties of interest are maintained once the loop terminates.

Once a candidate invariant is obtained, the next step is to verify its correctness. To this end, we instruct an SMT solver to check the satisfiability of the following formula

$$\neg(P \Rightarrow I) \vee \neg\,(I \wedge B \wedge S \Rightarrow I) \vee \neg(I \wedge \neg B \Rightarrow Q). \qquad (2)$$

If the SMT solver returns "sat" (satisfiable), we will obtain a counterexample that demonstrates a violation of one of the three conditions. Otherwise, i.e., the solver returns "unsat" (unsatisfiable), we get a valid loop invariant.

Automatically generating a loop invariant is challenging, even generally impossible due to the undecidability of the problem. So incomplete/heuristic approaches are necessary in general. However, with a candidate invariant, SMT solvers such as Z3 [15] normally can determine whether it meets the three requirements efficiently. In light of this, most practical methods adopt a *guess-and-check* strategy. In a nutshell, program information (e.g., results from symbolic execution, syntactic features of the program, etc.) is gathered to hypothesize a candidate invariant, which is further checked by an SMT solver. If the violation of the aforementioned three conditions is detected, the solver provides a counterexample, which can be utilized in the "guessing" procedure to generate a new candidate invariant. This process iterates multiple times until the correct result is found or the resource budget is exhausted.

2.2 Motivating Example

Figure 1 presents a code snippet in C from our benchmark. The pre-condition of this program is $P(lo, mid, hi) := mid > 0 \wedge lo = 0 \wedge hi = mid * 2$ and the post-condition is $Q(lo, mid, hi) := lo = hi$. The loop condition is $B(lo, mid, hi) := mid > 0$. In this example, a valid loop invariant might be $(hi - lo = 2 * mid) \wedge (mid >= 0)$.

Previous tools (e.g., LoopInvGen [38], Code2Inv [46], CLN2INV [42], G-CLN [52], and LIPUS [54]) have struggled to identify the correct solution in this example. Simply using LLMs cannot generate the valid invariant either, as shown in Figure 1. However, although none of LLM's outputs are a valid loop invariant, the *correct predicates* (e.g., $hi - lo = 2 * mid$ and $mid >= 0$ in the example) have appeared in different responses (highlighted in red in Figure 1). Note that correct predicates are those which hold true before the loop starts and at the beginning and end of each iteration. This observation motivates us to identify the correct predicates from the candidates and reassemble them (e.g., with a conjunctive normal form) to produce a valid loop invariant.

```
1.  int main() {
2.    //variable declarations
3.    int lo, mid, hi;
4.    //pre-condition
5.    assume(mid > 0 && lo == 0 && hi == 2 * mid);
6.    //loop-body
7.    while(mid > 0) {
8.      lo = lo + 1;
9.      hi = hi - 1;
10.     mid = mid - 1;
11.   }
12.   //post-condition
13.   assert(lo == hi);
14.}
```

```
Valid loop invariant:
  assert((hi − lo == 2 * mid) && (mid >= 0));
GPT's answer:
  assert((lo + hi == 2 * mid + lo) && (mid >= 0));
  assert((lo + hi == 2 * mid));
  assert((lo + mid == hi) && (mid >= 0));
  assert((hi − lo == 2 * mid));
The red part is the correct predicate.
```

Figure 1: An example from our benchmark. Although LLM may not return the valid loop invariant, the correct predicates are already included in different responses.

3 OUR APPROACH

In this section, we present a neuro-symbolic approach for loop invariant inference, which combines LLMs with traditional bounded model checking tools to generate loop invariants automatically. Figure 2 outlines the framework of our approach, which consists of the following major steps.

(1) Initially, we prepare a prompt containing program information, task description and suggestions for LLMs, asking them to generate loop invariants. The output is a set of candidate invariants in the form of "`assert(. . . );`".
(2) Each returned candidate invariant from the LLM is verified using an SMT solver. If the invariant satisfies the three conditions as per Eq. (1), it is returned and the procedure terminates; otherwise, the SMT solver returns a counterexample for the candidate invariant.
(3) We split each failed candidate invariant into individual predicates, and use bounded model checking tools to check whether each predicate holds within the input program. Verified predicates form the correct predicate set.
(4) We combine the verified predicates to form a new loop invariant, and submit it to the SMT solver for verification. Similarly, the procedure terminates if the combined invariant passes SMT checking, and a counterexample is returned otherwise.
(5) We incorporate the counterexample generated in either Step 2 or Step 4 to come up with a new prompt, query LLMs to obtain a new loop invariant, and go back to Step 2.

The automatic generation process is repeated until the desired loop invariant is obtained or the resource budget is exhausted. In the sequel, we present the details of two key steps, i.e., predicate filtering and LLM querying.

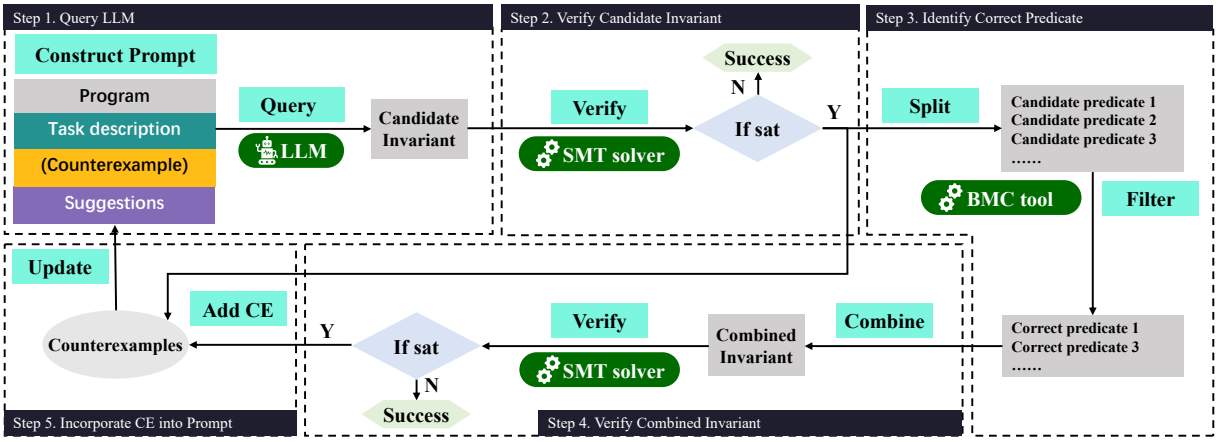

Figure 2: An overview of LaM4Inv.

3.1 Predicate Filtering

The main purpose of predicate filtering is to identify those (atomic) predicates which are most likely to be assembled to give rise to a valid loop invariant. As mentioned in Section 1, we leverage bounded model checking. One might be wondering why we cannot simply use SMT solvers to determine whether a predicate should be included. To see this, consider the example in Figure 1 again. Assume we want to check the predicate $hi - lo = 2 * mid$. An SMT solver might return a counterexample, e.g., the program state $\{lo = 0, mid = -1, hi = -2\}$ and thus filtered this predicate out. However, a careful examination would reveal that this is a spurious counterexample because this state can never be reached in a program execution (i.e., it violates provability), the consequence of which is that a valid loop invariant would be missed. This means that one should take all the executions of the program into account, and should consider an under-approximation of the program states when ruling out predicates, for which BMC fits squarely.

We next describe the proposed predicate filtering mechanism, which processes candidate loop invariants in both conjunctive normal form and disjunctive normal form.

Conjunctive normal form. If the candidate invariant is in the conjunctive normal form, we first decompose it into individual predicates by splitting it at the outermost conjunction. Each predicate is then transformed into an "assert" statement. These assertions are then inserted into the source code at points, for instance, before the loop commences, at the start of each loop iteration, and at the end of each iteration. These assert statements are subsequently validated using a bounded model checking tool. If the tool returns a failure, the predicate does not correspond to any property contained within the program. Such a result implies that the predicate narrows the state space represented by the candidate invariant, thereby diminishing its reachability and inductiveness. Consequently, we discard this predicate. Conversely, if the verification does not fail, indicating that the predicate holds true across the specified points in the loop, we retain this predicate and add it to the correct predicate set.

Disjunctive normal form. For a candidate loop invariant in the disjunctive normal form, we directly input it into a bounded model checker. If this tool returns "false", the candidate invariant is immediately discarded; otherwise, we proceed to split the invariant into individual predicates at the outermost disjunction. Each of these predicates is negated and transformed into an "assert" statement. These assertions are inserted at three critical points in the source code, similar to the previous case, and are re-evaluated using the bounded model checker. If the tool returns "true", we can conclude that the negated predicate consistently holds in all three locations within the program, which implies that the original predicate has never held along any execution of the program, suggesting that the predicate is redundant in the original disjunctive normal form, and should be removed. We evaluate all the predicates and conjunct the remaining ones as a disjunctive form which is then added to the collection of candidate predicates.

Remarks. The above predicate filtering mechanism allows us to identify the predicates that align with properties during program execution. In particular, BMC ensures that these predicates hold

Algorithm 1: Predicate Filtering Algorithm

Input: A program p, current candidate invariant set CIs, correct predicate set CPs.
Output: Updated correct predicate set CPs.

```
1  Function BMCFilter(p, CIs, CPs):
2      foreach can_I in CIs do
3          if can_I is in conjunctive normal form then
4              foreach can_P in can_I.split() do
5                  if BMC.verify(p, can_P) is not Failure then
6                      CPs.add(can_P);
7          else
8              if BMC.verify(p, can_I) is not Failure then
9                  foreach can_P in can_I.split() do
10                     if BMC.verify(p, ¬(can_P)) is not
                           Failure then
11                         can_I.remove(can_P);
12                 CPs.add(can_I);
13     return CPs
```

Prompt 1: Initial prompt.

[p]

Print loop invariants as valid C assertions that help prove the assertion. In order to get a correct answer, you may want to consider both the situation of not entering the loop and the situation of jumping out of the loop. If some of the preconditions are also loop invariant, you need to add them to your answer as well. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...);'

before entering the loop and before and after each loop iteration. Consequently, they naturally satisfy the reachability and inductiveness conditions of the invariant. Regarding provability, as our candidate predicate set grows, the program state space it constrains becomes increasingly tighter. Eventually, this constrained program state space $S' = (I \wedge \neg B)$ will satisfy $S' \subseteq Q$, indicating that the correct loop invariant has been found. The entire algorithm of our predicate filtering mechanism is summarized in Alg. 1.

Our filtering mechanism can theoretically be applied with other loop invariant inference methods as long as they are built on the "guess-and-check" framework. We choose LLMs as our candidate invariant generation agent primarily because our empirical observations show that they often can capture correct predicates, but struggle to combine them effectively using logical conjunctions ($\wedge$) or disjunctions ($\vee$) to form valid loop invariants.

3.2 LLM Querying

In this work, to effectively extract knowledge from LLMs, we design two types of prompts, i.e., *initial prompt* and *intermediate prompt*.

Initial prompt. The initial prompt, as presented in Prompt 1, is used at the beginning of our loop invariant inference procedure, when only the program information is available. Specifically, we combine the program source code (denoted as [p]) with a task description and generation suggestions. In the suggestions, we encourage LLMs to pay attention to the pre-condition P and loop condition B, as both have a significant impact on reachability and inductiveness. We also specify that the output format must be in the form of "`assert(...);`", facilitating subsequent automatic processing.

Intermediate prompt. In our approach, each generated candidate loop invariant (either directly obtained from LLMs or recombined from verified predicates) is verified using an SMT solver. If the verification fails, SMT solvers may provide critical feedback, and our intermediate prompt is designed to handle such cases.

To maximize the effect of feedback, we consider both the provided counterexample and its specific type (i.e., which condition in Eq. (2) is violated). To obtain this type, we use an SMT solver to solve the three clauses in Eq. (2), and record the type if any clause yields a solution. According to the three conditions (i.e., reachability, inductiveness, and provability) that the current candidate invariant fails to satisfy, we design different intermediate prompts. Take reachability as an example. The corresponding prompt is presented in Prompt 2.

Prompt 2: Intermediate prompt with reachability violation.

[p]

Print loop invariants as valid C assertions that help prove the assertion. Your previous answer [CI] is too strict and not reachable. The reachability of the loop invariant means that the loop invariant I can be derived based on the pre-condition P, i.e. $P \Rightarrow I$.

The following is a counterexample given by z3: [CE]

In order to get a correct answer, you may want to consider the initial situation where the program won't enter the loop. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...);'

The intermediate prompt is designed to guide the LLM to modify the candidate invariant (denoted as [CI]) based on the feedback from the counterexample (denoted as [CE]). It generally consists of the following parts: the current program p, the task description, the previous answer, the failure reasons for the previous answer, the counterexample provided by the SMT solver, and the generation suggestions. Specifically, when the candidate invariant fails to satisfy reachability, we inform the LLM that the previous answer was too stringent, making it impossible to derive the candidate invariant from the pre-condition.

Details of the other two types of intermediate prompts can be found in Appendix A.3. Essentially, if the provability is not satisfied, we indicate that the previous answer was too loose, and the derived invariant is insufficient to deduce the post-condition; if inductiveness is not satisfied, we explain that the previous answer does not hold for each loop iteration and may need to account for special cases during loop execution.

3.3 Overall Algorithm

The overall algorithm is summarized in Alg. 2. "`LaM4Inv`" refers to our main function, whose input includes the current program to be solved, and the three loop invariant conditions in Eq. (2). After initializations, `LaM4Inv` utilizes the initial prompt to drive LLMs in inferring candidate invariants (denoted as *cur_CIs*), and adds them to the set of candidate invariants (denoted as *CIs*). It then uses BMC tools to update the correct predicate set (denoted as *CPs*). The function "`BMCFilter()`" refers to invoking Alg. 1. Subsequently, it iteratively invokes "`verifyAndRefine()`" to verify the candidate invariant from *CIs* and reassemble a new invariant formed by all verified predicates from *CPs*, prompting LLMs with counterexamples and their types given by the SMT solver to infer new candidates. Throughout the loop, we keep track of the number of proposals (denoted as *count*), and terminate the algorithm if it exceeds the predefined limit. In practice, we also add a timeout

Algorithm 2: The LaM4Inv Algorithm

Input: A program p, the three loop invariant conditions VC, maximum number of iterations N.
Output: the correct loop invariant I

```
Function LaM4Inv(p, VC):
    solved ← False;
    count ← 0;
    I ← None;
    CIs, CPs ← ∅, ∅;
    prompt ← promptGen(p, ⟨∅, ∅⟩);          ▷ Initial prompt
    cur_CIs ← LLM.query(prompt);
    CIs.push(cur_CIs);
    CPs ← BMCFilter(p, cur_CIs, CPs);        ▷ Call Alg. 1
    while solved is False and count < N do
        count ← count + 1;
        can_I ← CIs.pop();
        I ← verifyAndRefine(p, VC, can_I, CIs, CPs);
        if I is not None then
            solved ← True;
        else
            count ← count + 1;
            can_I ← conjunction(CPs);
                                  ▷ Reassemble new invariant
            I ← verifyAndRefine(p, VC, can_I, CIs, CPs);
            if I is not None then
                solved ← True;
    return I;
Function verifyAndRefine(p, VC, can_I, CIs, CPs):
    ce ← SMT-solver.verify(can_I, VC);
    if ce is not None then
        I ← None;
        prompt ← promptGen(p, ⟨can_I, ce⟩);
                                  ▷ Intermediate prompt
        cur_CIs ← LLM.query(prompt);
        CIs.push(cur_CIs);
        CPs ← BMCFilter(p, cur_CIs, CPs);    ▷ Call Alg. 1
    else
        I ← can_I;
    return I;
```

constraint. Each LLM query begins a new session without providing the context from previous dialogues for saving costs.

4 EVALUATION

In this section, we present the experimental results. Our experiments are designed to answer the following research questions.

RQ1. How effective is LaM4Inv in inferring loop invariants compared with the state-of-the-art methods?
RQ2. What impacts do different LLMs, prompt design and predicate filtering mechanisms have on LaM4Inv's performance?

4.1 Setup

Benchmark dataset. To more comprehensively evaluate different loop invariant inference methods, we curated a dataset of 316 benchmark problems. Our benchmark contains the 133 problems collected by Code2Inv [46], which are commonly evaluated by previous work. Additionally, we have manually crafted 84 problems from the 2019 SyGuS competition [1] and 99 problems from the 2024 SV-COMP benchmarks [5]. More details of the benchmark construction are included in Appendix A.1.

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

- LoopInvGen [38] uses an enumerative synthesis technique which repeatedly adds consistent clauses to strengthen the post-condition until it becomes inductive.
- CVC5 [3] is an advanced SMT solver equipped with a syntax-guided synthesis engine for loop invariant inference.
- Code2Inv [46] uses reinforcement learning to infer loop invariants, along with recurrent and graph neural networks to capture program's features.
- LIPUS [54] is built upon Code2Inv. It improves reinforcement learning with a two-dimensional reward, and combines it with a template iteration method.
- CLN2INV [42] infers loop invariants from the program execution traces. It designs new neural network models to fit logical expressions.
- G-CLN [52] is also based on program execution traces and further extends CLN2INV by using gating units and dropout.
- ESBMC [21] is a context-bounded model checker for verifying programs, combining BMC, k-induction, abstract interpretation, SMT and constraint programming solvers.
- LEMUR [50] casts program verification tasks into a series of deductive steps suggested by LLMs and validated by automated reasoners, assisting ESBMC in verifying programs.

Among these baselines, LoopInvGen, CVC5 and ESBMC are traditional symbolic methods, and the rest are learning-based methods. The last two methods are based on k-induction verifier [17], which aligns with the verification procedure used in our approach.

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 candidate invariants proposed during the inference process; and 3) the time consumed to infer the invariants. The first metric is related to effectiveness, and the latter two metrics are related to efficiency. Note that the latter two metrics are computed based on the successfully generated loop invariants of each method.

Implementation details. In our experimental setup, all methods were evaluated under identical settings, using the same hardware and benchmark problems. Each method was subjected to a timeout of 600 seconds per problem, with a maximum of 50 loop invariant proposals allowed. For the experiments involving LLMs, we utilized four different models: Llama3-8B, GPT-3.5-Turbo (gpt-3.5-turbo-0125) [37], GPT-4 (gpt-4-0613) [36], and GPT-4-Turbo (gpt-4-turbo-2024-04-09). For LEMUR [50], we used their default LLM, GPT-4. When invoking these LLMs, we adopt the default settings (e.g., parameters *temperature* and *penalty* are both set to zero). In our

Table 1: The performance comparison of different loop invariant inference methods. The proposed LaM4Inv generates 309 correct loop invariants out of 316 benchmark problems.

Methods	# Solved Benchmarks				# Avg. Proposals	Avg. Time (s)
	All	None	Only	Total (133/84/99)		
LoopInvGen [38]			0	218 (107/49/62)	5.6	17.8
CVC5 [3]			1	207 (107/46/54)	13.9	5.5
Code2Inv [46]			0	210 (110/47/53)	10.9	137.6
LIPUS [54]			0	159 (124/18/17)	3.7	48.4
CLN2INV [42]	43	4	0	211 (124/35/52)	23.9	0.6
G-CLN [52]			0	219 (116/45/58)	20.1	2.6
ESBMC [21]			0	126 (70/23/33)	0	0.2
LEMUR [50]			0	177 (81/43/53)	1.5	9.8
LaM4Inv			20	309 (133/81/95)	3.7	35.7

study, the default bounded model checker tool employed is ESBMC [21], because it is an efficient and state-of-the-art BMC-based program verification tool. Our framework is flexible as other BMC tools can be adopted as well. The bound (i.e., the number of times the loop is unrolled) of ESBMC is set to 10. The experiments are conducted on a GPU server with an Intel Core i9-13900k@3.00GHz CPU, 48GB RAM, and a GeForce RTX 4090 GPU.

4.2 RQ1. Effectiveness and Efficiency

We first compare the overall performance of different methods in loop invariant inference. The experimental results across 316 benchmark problems are shown in Table 1. In the table, "All" indicates the number of problems solved by all methods; "None" denotes the problems that no method could solve; "Only" means the number of problems only solved by the specific method; "Total" represents the total number of benchmark problems solved by the method. The three numbers in the parentheses are the respective numbers of solved problems from three different sources (i.e., Code2Inv, SyGuS 2019, and SV-COMP 2024). The average number of proposals and average time required by each method on the solved problems are also reported.

As shown in the table, LaM4Inv successfully infers loop invariants for 309 out of 316 benchmark problems, which is at least 90 more than the competitors. Additionally, our LaM4Inv generates the correct loop invariant for 20 problems that none of the existing competitors could solve. Such results demonstrate the superior performance of the proposed approach.

Among the competitors, LIPUS performs relatively well on the original 133 benchmark problems but struggles on the expanded problems. This is probably due to the overfitting problem which is common in machine learning based methods. For example, the two-dimensional reward design in LIPUS relies on manually defined heuristics, and may not generalize well across different datasets. CLN2INV and G-CLN face similar issues,[1] i.e., the performance on the expanded problems is also significantly worse than that on the original problems. This is primarily due to their reliance on generating training data for the continuous logic networks. We observe 36 out of the 183 new problems result in timeout due to excessively long data generation process.

In contrast to the above three methods, LoopInvGen, CVC5, and Code2Inv perform relatively stable across the three dataset sources. LoopInvGen and CVC5 are traditional symbolic methods, and thus are expected to be more generalizable. The result of Code2Inv, as a reinforcement learning based method, is relatively surprising. We conjecture that its relative better generalization is due to the simple but effective reward design, i.e., the number of counterexamples the candidate invariant can pass.

Among the competitors, although efficient, ESBMC solves the least problems. This is because when the branching of loop execution traces become complex, verifying all reachable program states becomes challenging for ESBMC. This also shows the importance of the synergy between LLMs and model checking. LEMUR is also efficient. This is due to the fact that LEMUR first uses ESBMC to verify the program, and queries LLMs to generate loop invariants when ESBMC fails. With this straightforward combination of ESBMC and LLM, LEMUR generates the correct loop invariants for 51 more benchmark problems. In contrast, our LaM4Inv solves 183 more problems compared to ESBMC. This result shows the superiority of the proposed closed-loop combination, i.e., LLMs' outputs are checked by BMC to assemble a new loop invariant, which is further verified with feedback leveraged for LLM querying.

For efficiency, LaM4Inv generates fewer proposals except for ESBMC and LEMUR. For LEMUR, on the 51 problems that require LLMs to solve, the average time is 33.6 seconds and the average number of proposals is 5.2. In contrast, LaM4Inv's average time is 35.7 seconds and average number of proposals is 3.7. LaM4Inv takes relatively longer runtime compared with several competitors. This can be attributed to two main factors: 1) the inference of LLMs itself takes a significant amount of time, and 2) we need to decompose each generated candidate invariant and verify each predicate using ESBMC.

4.3 RQ2. Ablation Study

To thoroughly understand the impact of different components in our approach, we conduct a series of ablation studies. These studies cover the specific LLMs used, the content of the prompts, and the

[1]Using the original code provided by the authors, we can solve 191 (123/31/37) and 184 (123/30/31) problems for CLN2INV and G-CLN, respectively. However, we found a bug in the code, fixing which could further improve their performance. Hence, we report the results of the fixed version in the table.

Table 2: The ablation results of different components in LaM4Inv. Both our LLM prompt design and BMC based predicate filtering play important roles.

LLMs	# Solved Benchmarks			# Avg. Proposals			# Avg. Time (s)		
	Baseline Prompt	No BMC	Full	Baseline Prompt	No BMC	Full	Baseline Prompt	No BMC	Full
Llama-3-8B	231	214	233	8.1	10.8	7.0	20.7	10.4	18.8
GPT-3.5-Turbo	265	248	271	6.1	8.9	5.4	22.0	21.2	28.9
GPT-4	241	275	306	3.8	6.3	5.0	27.1	26.5	46.3
GPT-4-Turbo	246	275	309	3.5	5.5	3.7	28.2	27.0	35.7

predicate filtering mechanism. Specifically, we consider four LLMs including Llama-3-8B, GPT-3.5-Turbo, GPT-4, and GPT-4-Turbo. For the prompt, we consider a baseline prompt adopted by LEMUR [50], which is detailed in Appendix A.2. In a nutshell, the baseline prompt does not contain the feedback information from SMT solvers and the related modification suggestions. We also consider the impact of predicate filtering by removing it from LaM4Inv (denoted as "No BMC"). The ablation results are listed in Table 2, where "Full" refers to the complete LaM4Inv.

Impact of different LLMs. Table 2 demonstrates performance variability across the different LLMs. GPT-4 and GPT-4-Turbo generally show better performance under both the "No BMC" and the "Full" columns, and GPT-3.5-Turbo outperforms the others using the baseline prompt. Since both "No BMC" and "Full" incorporate feedback information into the prompt, this result indicates that GPT-4 series are more capable of generating the correct loop invariants when feedback information is provided. In contrast, we observe that GPT-3.5-Turbo may output more diverse results when the same baseline prompt is provided, thus has a higher chance to obtain the correct invariant with our predicate filtering mechanism. Nevertheless, even the least performing LLMs demonstrated comparable effectiveness to the state-of-the-art methods mentioned before. For example, using Llama-3-8B could generate 15 more correct loop invariants than the best existing competitor LoopInvGen.

Impact of feedback information. Next, we compare the results of the "Baseline Prompt" columns with those of the "Full" columns. In terms of the number of solved benchmark problems, our full method LaM4Inv is better in all cases, especially for GPT-4 series. For example, with GPT-4-Turbo, LaM4Inv solves 63 more problems. This, again, shows the importance of feedback information, whose absence hampers LLMs' ability to iteratively refine and improve their candidate invariants.

Impact of predicate filtering. In this ablation experiment, we compare the results in the "No BMC" columns with those in the "Full" columns. We observe improvements in all cases w.r.t. the number of solved benchmark problems and the average number of proposals, clearly indicating our predicate filtering mechanism's usefulness in enhancing the efficacy of loop invariant inference. For instance, when using GPT-4-Turbo, our full LaM4Inv solves 309 benchmark problems with 3.7 proposals in average; when the predicate filtering mechanism is excluded, it solves 34 less benchmark problems and meanwhile requires 1.8 more proposals in average. The trade-off for this improvement is the additional time required when using the BMC tool to filter predicates.

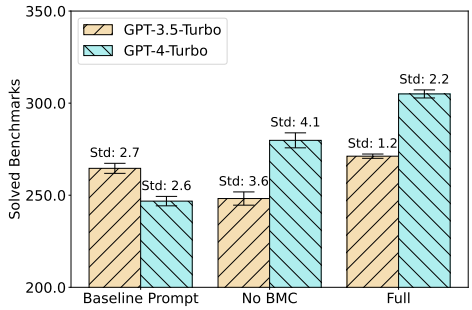

Figure 3: The robustness results of LaM4Inv against the randomness of LLM outcomes. LaM4Inv performs stable and the predicate filtering mechanism improves stability.

Robustness to randomness. We next evaluate the robustness of LaM4Inv against the randomness of LLM outcomes. Specifically, we still consider the three ablation methods mentioned above, and apply GPT-3.5-Turbo and GPT-4-Turbo for inferring loop invariants. Five runs of the experiments are conducted for each ablation method, and the results are shown in Figure 3, where both mean and standard deviation results are plotted. It can be observed that the standard deviation of our full LaM4Inv is relatively low (2.2 for GTP-4-Turbo and 1.2 for GPT-3-Turbo), indicating that LaM4Inv's performance is relatively stable. Additionally, comparing the results in the "No BMC" columns and the "Full" columns, the standard deviation is reduced when BMC is incorporated. This result demonstrates that our BMC based predicate filtering mechanism mitigates the randomness in the inference of LLMs.

5 DISCUSSION

5.1 Threats to Validity

There are three main validity threats. The first concerns the limited SMT solver capabilities. SMT solvers often struggle with handling multiplication, division, and power operations. This limitation becomes a bottleneck for some methods [49, 54]. Existing works on non-linear invariant inference also restrict the number of multiplicative layers (typically 4-5) [52, 54] and avoid dealing with power operations where both the base and exponent are variables. While LLMs can generate candidate invariants of the form "`pow(a, b)`", we are unable to verify these invariants due to the performance

```
1. int main() {
2.     unsigned int x, y;
3.     //pre-condition
4.     assume(x == 0 && y == 0);
5.     //loop-body
6.     while (x < 1000000) {
7.         if (x < 500000) {
8.             y++;
9.         }
10.        else {
11.            y--;
12.        }
13.        x++;
14.    }
15.    //post-condition
16.    assert(y == 0);
17.}
```

(a) Case 1. A successfully-solved but time-consuming case due to BMC's performance limitation.

```
1. int main() {
2.     int x, y;
3.     //pre-condition
4.     assume(x == 0);
5.     //loop-body
6.     while (x < 99) {
7.         if (y % 2 == 0) {
8.             x = x + 2;
9.         }
10.        else {
11.            x = x + 1;
12.        }
13.    }
14.    //post-condition
15.    assert((x % 2) == (y % 2));
16.}
```

(b) Case 2. A failed case due to LaM4Inv's weak DNF handling.

Figure 4: Problems that LaM4Inv struggles to solve.

constraints of SMT solvers. This limitation is the primary reason we did not include non-linear benchmarks in our test suite.

The second concerns limited benchmarks. Although we expanded our benchmark set to 316 problems, these benchmarks are still quite basic and resemble toy problems when compared to real-world applications. The limited size of the dataset also poses a challenge for fine-tuning LLMs, as it does not provide enough data to capture the complexities of real-world programs.

The third concerns data leakage. Despite the risk of overfitting due to a small dataset being observed in previous work [42, 52, 54], we cannot guarantee that the benchmarks used in our study are not part of the pre-training data for LLMs. However, we believe this issue is less prominent in loop invariant inference because the loop invariants themselves are unlikely to be included (note that only the requirements/assertions to verify loop invariants are present in existing benchmarks).

5.2 Limitations

There are limitations of the proposed LaM4Inv. First, considering the efficiency aspect, it relies on BMC tools which are sometimes inefficient in terms of identifying property violations. Second, considering the solving capability, there are still cases in which LLMs cannot produce the correct predicates.

Case 1: a successfully-solved but time-consuming case. As noted in Table 1, our LaM4Inv takes relatively more time when generating loop invariants. One main reason is due to the inefficiency of BMC tools under certain circumstances. Figure 4a presents such an example. Consider a candidate invariant given by the LLM: "$((x >= 500000 \;\&\&\; y == x - 500000) \;||\; (x < 500000 \;\&\&\; y == x))$". The BMC tool will not find a program execution path that violates this candidate invariant within 500,000 loop iterations, as it requires the loop to be unfolded 500,000 to 1,000,000 times. Given the computational speed constraints of BMC, such extensive unfolding is often impractical. Consequently, we have to rely only on LLMs to generate the correct invariants, substantially prolonging the inference process. Enhancing the predicate filtering through appropriate symbolic execution or generating suitable test cases automatically could potentially address this issue.

Case 2: a failed case due to LaM4Inv's weak DNF handling capacity. Figure 4b demonstrates an example where LaM4Inv's weak DNF (disjunctive normal form) handling capacity results in unsuccessful invariant inference. The correct loop invariant includes a predicate "$(x <= 99 \;\&\&\; y \,\%\, 2 == 1) || (x <= 100 \;\&\&\; y \,\%\, 2 == 0 \;\&\&\; x \,\%\, 2 == 0)$". However, when attempting to infer the invariant, LLMs frequently focus on the parity of variable x (odd or even) rather than considering the range of values x can take under different conditions ($x <= 99$ or $x <= 100$). Consider a candidate predicate $(x <= 99 \;\&\&\; y \,\%\, 2 == 1 \;\&\&\; x \,\%\, 2 == 1) || (x <= 100 \;\&\&\; y \,\%\, 2 == 0 \;\&\&\; x \,\%\, 2 == 0)$. This predicate would be discarded by our filtering mechanism because the entire expression is incorrect (i.e., x being odd when y is odd is not always true). However, if we further analyze its conjunctive clause, we may find that simply removing $x\%2 == 1$ would make the entire predicate correct. Unfortunately, further refining this process would require evaluating all possible combinations of predicates within each conjunctive clause of the DNF, which has an exponential time complexity and is impractical to implement.

6 RELATED WORK

In this section, we briefly review the related work, which is divided into traditional approaches and learning-based approaches.

Traditional approaches. Traditional loop invariant inference methods mainly rely on symbolic methods such as abduction analysis [6, 16], dynamic analysis [18, 19, 31], model checking [26, 47], Craig interpolation [27, 34], abstract interpretation [12–14, 29], random search [43], constraint solving [11, 24], syntax-guided synthesis [3, 4], and counterexample guided abstraction refinement [10, 55]. For example, Daikon [19] dynamically detects likely program invariants by running a program and observing the values it computes during execution. It then reports properties that consistently hold true over these observed executions. LoopInvGen [38] introduces a data-driven approach, which combines symbolic execution and constraint solving techniques to efficiently extract precise program invariants. Eldarica [26] is a model checker for Horn clauses over integer arithmetic, algebraic data types, and bit-vectors, using predicate abstraction combined with counterexample

guided abstraction refinement to check the satisfiability of Horn clauses. CVC4/CVC5 [3, 4] are efficient SMT solvers equipped with a syntax-guided synthesis engine, enabling the synthesis of functions in accordance with background theories and their combinations.

Learning-based approaches. Recently, machine learning and deep learning techniques have been employed in various software engineering tasks [8]. Specifically, for the loop invariant inference problem, existing methods mainly use decision tree [20, 22, 23, 41, 51], support vector machine [32, 45], PAC learning [44], reinforcement learning [46, 54], continuous logic networks [42, 52], and LLMs [7, 28, 33, 48, 50] under the guess-and-check framework. For example, Code2Inv [46] and LIPUS [54] employ reinforcement learning to search for the correct loop invariants. However, reinforcement learning highly relies on the quality of reward functions, and may require extensive trial and error. CLN2INV [42] and G-CLN [52] are both founded on fuzzy logic [35], and utilize deep learning models to model program execution paths. However, approximation errors introduced when fitting discrete logic expressions with continuous functions are inevitable, which may have adverse effects on loop invariant inference especially for complex expressions.

A few recent attempts have explored the capability of LLMs in loop invariant inference. Kamath et al. [28] directly empoly LLMs to find inductive loop invariants; Chakraborty et al. [7] further rank the outputs of LLMs; Liu et al. [33] propose a self-supervised learning paradigm to fine-tune LLMs for loop invariant inference; LEMUR [50] employs LLMs to generate properties that assist ESBMC in program verification; Wen et al. [48] consider the broader problem of automated specification synthesis. More generally, Pei et al. [39] focus on fine-tuning LLMs to generate program properties; Yao et al. [53] employ multi-turn prompts with feedback to leverage LLMs for automated proof synthesis in Rust. Different from the above work, LaM4Inv forms a closed-loop synergy between LLM and BMC. That is, LLMs' outputs are checked by BMC to filter out incorrect predicates; the remained predicates are reassembled and verified, and the verification feedback is incorporated into the next round LLM prompting.

7 CONCLUSION

In this paper, we have proposed a new approach LaM4Inv that combines LLMs with bounded model checking to generate loop invariants automatically. The key of LaM4Inv is a "query-filter-reassemble" procedure that forms a closed-loop synergy between the (neural) LLM for generating candidate variants and (symbolic) bounded model checking for filtering out incorrect predicates. Evaluations on an expanded dataset consisting of 316 problems confirm the superior performance of LaM4Inv over a range of state-of-the-art methods in loop invariant generation.

Our work showcases the power of a neuro-symbolic approach in solving traditionally challenging problems in the area of software engineering, which, we believe, can be used elsewhere, e.g., static analysis, code generation, and DevOps/MLOps.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grant #62025202), the Fundamental Research Funds for the Central Universities of China (Grant No.020214380102), 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.

A APPENDIX

A.1 Benchmark Details

In addition to the 133 benchmark problems from existing work, we have manually crafted 84 problems from the 2019 SyGuS competition and 99 problems from the 2024 SV-COMP benchmarks. Each problem consists of a C code snippet and the corresponding SMT-LIB2 files. Each C code snippet is meticulously annotated to outline pre-conditions, the body of the loop, and post-conditions. All the snippets contain a single loop, which may include complex nested if-then-else structures, reflecting typical control flow scenarios in programming. Moreover, to simulate real-world programming environments where external dependencies are common, the code snippets may include calls to unknown functions, denoted as "`unknown()`". This introduces additional complexity to the loop invariant inference problem. The SMT-LIB2 [40] files are provided in two distinct formats: SMT-LIB and SyGuS-IF. Each format is tailored to support different existing methods such as CVC5 [3] and LoopInvGen [38].

For the 133 benchmark problems from Code2Inv [46], which is collected from previous work [16, 23] and the 2017 SyGuS competition [2], Ryan et al. [42] identified that 9 out of them were unsolvable. During our own review of the benchmarks, we discovered additional 6 unsolvable cases. We undertook corrections to these problematic cases based on their original SMT-LIB2 files.

For the problems from the 2019 SyGuS competition, we follow Code2Inv [46] by manually converting SyGuS files into C programs and using Clang [30] to generate the required SMT-LIB2 files. After removing duplicates, we obtain 84 problems.

For the problems from the 2024 SV-comp, we exclude programs containing pointers, arrays, nested loops, or multiple loops. To accommodate the prompt length limitations and enhance code readability, we exclude programs exceeding 100 lines. We also exclude programs that were semantically identical. We then standardize the coding style for the remaining programs and enhance them with additional annotations to improve clarity and uniformity. Additionally, we implement pre-conditions for programs susceptible to numerical overflow to ensure their operational correctness. After these steps, we obtain 99 problems.

A.2 Baseline Prompt

The baseline prompt is from Lemur [50], shown in Prompt 3, which is based on the program information only.

A.3 Other Intermediate Prompts

If the inductiveness/provability is violated by the candidate invariant, the intermediate prompt is shown in Prompt 4/Prompt 5.

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).

Prompt 3: Baseline prompt.

$[p]$

Print loop invariants as valid C assertions that help prove the assertion. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...);'

Prompt 4: Intermediate prompt with inductiveness violation.

$[p]$

Print loop invariants as valid C assertions that help prove the assertion. Your previous answer $[CI]$ is not inductive. The Inductiveness of the loop invariant means that if the program state satisfies loop condition B, the new state obtained after the loop execution S still satisfies, i.e. $\{I \wedge B\}\, S\, \{I\}$.

The following is a counterexample given by z3: $[CE]$

In order to get a correct answer, you may want to consider the special case of the program executing to the end of the loop. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...);'

Prompt 5: Intermediate prompt with provability violation.

$[p]$

Print loop invariants as valid C assertions that help prove the assertion. Your previous answer $[CI]$ is too weak and not provable. The Provability of the loop invariant means that after unsatisfying loop condition B, we can prove the post-condition Q, i.e. $(I \wedge \neg B) \Rightarrow Q$.

The following is a counterexample given by z3: $[CE]$

In order to get a correct answer, you may want to consider the special case of the program executing to the end of the loop. If some of the preconditions are also loop invariant, you need to add them to your answer as well. Use '&&' or '||' if necessary. Don't explain. Your answer should be 'assert(...);'

arXiv preprint arXiv:1904.07146 (2019).

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

[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] Xiangping CHEN, Xing HU, Yuan HUANG, He JIANG, Weixing JI, Yanjie JIANG, Yanyan JIANG, Bo LIU, Hui LIU, Xiaochen LI, Xiaoli LIAN, Guozhu MENG, Xin PENG, Hailong SUN, Lin SHI, Bo WANG, Chong WANG, Jiayi WANG, Tiantian WANG, Jifeng XUAN, Xin XIA, Yibiao YANG, Yixin YANG, Li ZHANG, Yuming ZHOU, and Lu ZHANG. [n. d.]. Deep Learning-based Software Engineering: Progress, Challenges, and Opportunities. *SCIENCE CHINA Information Sciences* ([n. d.]). https://doi.org/10.1007/s11432-023-4127-5

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using k-induction. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer, 351–368.

[18] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. 2019. Ilinva: Using abduction to generate loop invariants. In *Frontiers of Combining Systems: 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings 12*. Springer, 77–93.

[19] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[20] 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.

[21] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. 2018. ESBMC 5.0: an industrial-strength C model checker. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 888–891.

[22] 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.

[23] 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.

[24] 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.

[25] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

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

[27] 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.

[28] 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).

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

[30] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[31] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 788–801.

[32] 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.

[33] 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).

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

[35] Vilém Novák, Irina Perfilieva, and Jiri Mockor. 2012. *Mathematical principles of fuzzy logic*. Vol. 517. Springer Science & Business Media.

[36] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[37] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems (NeurIPS)* 35 (2022), 27730–27744.

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

[39] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *International Conference on Machine Learning*. PMLR, 27496–27520.

[40] Mukund Raghothaman and Abhishek Udupa. 2014. Language to specify syntax-guided synthesis problems. *arXiv preprint arXiv:1405.5590* (2014).

[41] 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.

[42] 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)*.

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

[44] 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.

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

[46] 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).

[47] 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.

[48] 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 (CAV)*.

[49] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation (PLDI)*. 718–730.

[50] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. In *The Twelfth International Conference on Learning Representations (ICLR)*.

[51] 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.

[52] 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.

[53] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging large language models for automated proof synthesis in rust. *arXiv preprint arXiv:2311.03739* (2023).

[54] 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.

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