Guiding LLM-based Loop Invariant Synthesis via Feedback on Local Reasoning Errors

TIANCHI LI, Peking University, China
ZHENYU YAN*, Peking University, China
JUNHAO LIU*, Peking University, China
PENG DI, Ant Group, China
XIN ZHANG†, Peking University, China

We propose a novel framework that provides constructive feedback to an LLM in the "guess-and-check" paradigm by formally verifying its own thinking process and detecting local reasoning errors. We apply this framework to the loop invariant synthesis problem. We prompt the model to produce a step-by-step natural language proof justifying its thinking process for the failed verification condition of its generated loop invariants. Then, we use an LLM to translate the reasoning steps into first-order logic implications, which can be checked automatically. An invalid implication pinpoints the exact logical flaw in the LLM's thinking process, which we then use to construct targeted feedback for refinement. We have implemented our approach in a tool called LORIS and evaluated it on a main benchmark suite of 460 C programs and an additional benchmark suite of 50 C programs each of which involves non-linear properties. On the main benchmark suite, LORIS solved 445 of the programs, and achieved an overall success rate of 93.1%. LORIS also demonstrates robustness on the challenging non-linear benchmark suite.

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

Additional Key Words and Phrases: loop invariant synthesis; large language models; auto-formalization; guess-and-check paradigm

1 Introduction

The "guess-and-check" paradigm is a fundamental problem-solving strategy across various domains, such as mathematical theorem proving, program analysis, program verification, and so on. In this paradigm, a potential solution is first proposed (the "guess" step), and then the correctness of the solution is verified (the "check" step). Recently, Large Language Models (LLMs), such as GPT series [27] and DeepSeek series [11, 12], have emerged as promising approaches to solve the "guess" step. With their extensive training on huge amounts

*These authors contributed equally to this work.
†Corresponding author.

Authors' Contact Information: Tianchi Li, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, litianchi@pku.edu.cn; Zhenyu Yan, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China, zhenyuyan@stu.pku.edu.cn; Junhao Liu, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China, liujunhao@pku.edu.cn; Peng Di, Ant Group, Hangzhou, Zhejiang, China, dipeng.dp@antgroup.com; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1558-4593/2026/4-ART
https://doi.org/10.1145/3806652

Just Accepted

of data, LLMs can generate creative and high-quality candidate solutions, which often capture the high-level intuition required to solve complex problems. Moreover, while LLMs have no correctness guarantees and can generate erroneous solutions [6, 17], the "check" step in the paradigm can reject incorrect proposals and therefore address this shortcoming, making the "guess-and-check" paradigm particularly suitable for applying LLMs. However, one challenge is that when LLMs fail to generate correct solutions, current "checkers" cannot provide fine-grained feedback to guide LLMs to improve their outputs. This paper tries to address this problem by generating constructive feedback via formally checking the thinking process of an LLM. In particular, we focus on subtle logical errors as LLMs often think in a generally right direction but make mistakes in the details due to issues like *hallucination* [17].

Loop invariant synthesis, a cornerstone task in program analysis and verification, is a good example to explore this feedback-guided approach. A loop invariant is a property that holds before and after each iteration of a loop. Identifying correct and sufficiently strong invariants is essential for formally proving program correctness. Over the past decades, numerous techniques have been developed to automate loop invariant synthesis, most of which follow the "guess-and-check" paradigm [14, 15, 30, 34, 35]. In this paradigm, a candidate invariant is first generated, and then the correctness of the invariant and the program property is checked by a verification tool. If the invariant fails to pass the check, certain feedback, such as counterexamples, will be generated to guide the approach to find another invariant until the correct one is generated. Existing techniques for generating candidate loop invariants include symbolic and data-driven approaches. These approaches either rely on the pre-defined templates [14, 35], or require a large amount of training data [15, 30]. On the other hand, although pre-trained LLMs have shown their abilities in comprehending and reasoning about programs, they still struggle to generate correct and complete loop invariants directly [20, 29].

To address this, we propose a novel framework that leverages LLMs to synthesize loop invariants, guided by formal verification feedback on the model's own thinking process. In our approach, an LLM first proposes a candidate loop invariant for a given program and property. If a verifier confirms that the invariant is both correct and sufficient to prove the program property, the synthesis process terminates successfully. Otherwise, we prompt the LLM to produce a step-by-step thinking process to prove the correctness of its proposed invariant. We then formally analyze its thinking process to detect exact logical errors and provide them as precise feedback to the LLM for refinement. This process iterates until a correct invariant is generated.

The key challenges in our framework are how to formalize the thinking process of an LLM and how to provide constructive feedback based on the verification result of the thinking process. To address the challenges, we leverage the idea of autoformalization [41], where the LLM generates its thinking process in natural languages, and an LLM (potentially a different one) translates the thinking process into a formal proof. However, in contrast to existing approaches [19, 44] that produce fully checkable proof scripts in formal systems such as Coq or Isabelle, we intentionally avoid this strategy due to the following considerations. First, proofs in these formal languages are quite different from their natural language counterparts, making the translation potentially unfaithful. Second, LLMs often generate incorrect proofs due to low-level tactic issues, such as incorrect theorem applications [23]. Third, it is difficult to map the failure of the proof script back to the natural language thinking process, thereby preventing effective feedback.

To address the above issues, our key observation is that LLMs often think in generally right directions but fail in the details. For example, it is very common for LLMs to have off-by-one errors. Following this observation, instead of checking the full correctness of the thinking process, we focus on checking *local correctness* to provide effective feedback to LLMs on subtle errors. Specifically, we instruct the LLM to formalize the key steps in the natural language proof into a series of first-order logic implications. In this way, we do not check the correctness of the high-level strategy but focus on the correctness of each individual step. These implications are lightweight and can be efficiently checked for validity by an SMT solver. An invalid implication discovered by the SMT

Just Accepted

```
extern int unknown();

int main() {
    int a = 0;
    int j, m;
    if(m <= 0) return 0;

    for(j = 1; j <= m; j++) {
        if(unknown())
            a++;
        else
            a--;
    }

    assert(a >= -m && a <= m);
    return 0;
}
```

Fig. 1. Example program.

```
/*@
    loop invariant i1: a >= -(j - 1) && a
        <= (j - 1);
    loop invariant i2: j >= 1;
    loop invariant i3: m > 0;
*/
```

Fig. 2. Loop invariants proposed by the LLM.

```
/*@
    loop invariant i1: a >= -j + 1 && a <=
        j - 1;
    loop invariant i2: j >= 1;
    loop invariant i3: j <= m + 1;
    loop invariant i4: m > 0;
*/
```

Fig. 3. Loop invariants refined by the LLM.

Just Accepted

solver corresponds directly to a specific logical error in the LLM's thinking process, which can form constructive feedback to the LLM.

We have implemented our framework as a tool called LORIS (**LO**cal **R**easoning-guided **I**nvariant **S**ynthesizer), which synthesizes loop invariants to verify properties of C programs. We evaluated LORIS on a main benchmark suite of 460 C programs requiring loop invariants for verification, and an additional benchmark suite of 50 C programs each of which involves non-linear properties. Our experiment results show that by leveraging GPT-4.1, our tool can successfully solve 445 of the programs, reaching an overall success rate of 93.1% on the main benchmark suite. On the challenging non-linear benchmark suite, our tool solves 47 of the 50 programs. These results show that the feedback framework provided by our approach can guide the LLMs toward correct solutions more effectively.

In summary, our contributions are as follows:

(1) We propose a novel paradigm that guides an LLM to refine its solution within the "guess-and-check" paradigm, by providing formal verification feedback on the model's own thinking process.
(2) We propose a framework for the loop invariant synthesis using our proposed paradigm, where an LLM's natural language reasoning is formalized into first-order logic implications and checked by an SMT solver to generate precise feedback. We have implemented our framework as a tool called LORIS.
(3) We have demonstrated the effectiveness of our approach on a main benchmark suite of 460 C programs requiring loop invariants for verification, and an additional benchmark suite of 50 C programs each of which involves non-linear properties. The result shows that our approach effectively improves the number of correct solutions proposed by LLMs.

2 Motivating Example

In this section, we use a verification problem for a simple C program to illustrate our approach. Figure 1 shows a C program with a loop and an assertion. Inside the loop, the variable a is non-deterministically incremented or decremented by one, depending on the return value of the function unknown. The loop iterates m times, where

m is a positive integer. The verification task is to prove that after the loop, the final value of a is bounded by m, i.e., $-m \leq a \leq m$.

To prove the assertion, correct loop invariants must be discovered first. A loop invariant is a program property that holds before the loop begins and is preserved by each iteration. In this example, a sufficient set of loop invariants is: 1) $-(j-1) \leq a \leq (j-1)$ and 2) $1 \leq j \leq (m+1)$. The verification of the program's assertion then relies on checking the following three conditions based on these invariants:

(1) *Establishment.* The loop invariants hold before the loop starts ($Pre \Rightarrow Inv$). The program's pre-conditions must imply the invariants:

$$(a = 0 \ \wedge \ j = 1 \ \wedge \ m > 0) \implies (-(j-1) \leq a \leq (j-1) \ \wedge \ 1 \leq j \leq (m+1))$$

(2) *Preservation.* If the invariants hold at the beginning of an iteration, they must also hold at the end ($\{Inv \wedge Cond\}Prog\{Inv\}$). We use a Hoare triple to represent this condition, where S is the loop body:

$$\{ \ -(j-1) \leq a \leq (j-1) \ \wedge \ 1 \leq j \leq m+1 \ \wedge \ j \leq m \} \ S$$

$$\{ \ -(j-1) \leq a \leq (j-1) \ \wedge \ 1 \leq j \leq m+1 \}$$

(3) *Post-condition.* When the loop terminates, the invariants must imply the desired program assertion ($Inv \wedge \neg Cond \Rightarrow Post$):

$$(-(j-1) \leq a \leq (j-1) \ \wedge \ 1 \leq j \leq (m+1) \ \wedge \ \neg(j \leq m)) \implies (-m \leq a \leq m)$$

The overall verification process of our approach follows the "guess-and-check" paradigm. That is, candidate loop invariants are first generated in the "guess" step, and the three conditions above are then formally verified in the "check" step, which can be efficiently handled by modern verification tools.

For the "guess" step, previous works have developed various techniques to find loop invariants, including template-based and learning-based methods[1]. Recently, the advances in comprehending and reasoning about programs shown by Large Language Models (LLMs) suggest their potential for generating invariants directly from a zero-shot prompt. Motivated by these insights, we prompt ChatGPT-4o to generate loop invariants to verify the assertion for the C program in Figure 1. It responds with the candidates shown in Figure 2. As we can see, the loop invariant $i1$ correctly captures the relationship between a and the loop counter j. While the loop invariant $i2$ tries to identify the bound of j, it omits the upper bound, $j <= m + 1$. Although the proposed invariants are individually correct, they are insufficient to prove the assertion. This is because, without the upper bound, the exact value of j at the loop termination cannot be precisely determined from the loop exit condition $\neg(j < m)$ only.

This initial result highlights a key challenge: while the LLM grasps the program's main logic, it may fail on critical details required for a formal check. Our goal is to provide the LLM with targeted feedback on such details that help it correct its answer. Here, a trivial feedback strategy would be simply stating that the verification condition is wrong, such as "the invariants you proposed are correct but insufficient to prove the assertion". However, this naive strategy is ineffective: in our experiments, this led the LLM to substitute $-m \leq a \leq m$ for the loop invariant $i1$ as a new one. While this property is what we want to prove, it is not a valid invariant because its preservation condition cannot be proven. Therefore, more precise feedback is needed - one that identifies the specific flaw in the LLM's thinking process.

Getting Natural Language Proof. To pinpoint the error, we first prompt the LLM to generate a proof in natural language to demonstrate its thinking process. Since the establishment and preservation of the loop invariants have been checked valid, we ask the LLM to focus on proving the post-condition. That is, demonstrating that the proposed invariants imply the assertion after the loop terminates. Figure 4 shows part of the proof given by the LLM. In this proof, the LLM correctly deduces the loop exit condition is $j > m$. However, it then incorrectly

Just Accepted

```
- From invariant **i1**: `a >= -(j - 1) && a <= (j - 1)`
  - At the end of the loop, `not B` implies `j > m`.
  - Substituting `j = m + 1`:
    - At termination, `a >= -(m + 1 - 1) && a <= (m + 1 - 1)`
    - Simplifies to: `a >= -m && a <= m`.
```

Fig. 4. A local reasoning step of the natural language proof given by the LLM.

```
[Initial]
a >= -(j - 1) && a <= (j - 1)
j > m

[Proof]
(j > m) ==> (j == m + 1) // At loop termination, j is m + 1.
(j == m + 1) && (a >= -(j - 1) && a <= (j - 1)) ==> (a >= -m && a <= m) //
    Substituting j = m + 1 into i1, the range for a becomes -m to m.

[Conclusion]
a >= -m && a <= m
```

Fig. 5. The corresponding formalized proof of the local reasoning step.

Just Accepted

assumes $j = m + 1$. This step is a logical leap: without the upper bound on j, $j > m$ does not necessarily imply $j = m + 1$.

Formalizing the Proof. Next, to automatically and formally detect such reasoning errors, we formalize the LLM's natural language proof. One may consider having the LLM generate a proof script in a language such as Coq [3] or Isabelle [26]. However, there is a gap between such formal language proofs and the natural language thinking process, and it is a difficult task for LLMs to generate a fully correct proof script. The proof scripts generated by the LLM often fail due to non-logical reasons, such as wrong theorem application or incorrect rewrite [23]. This makes it difficult to isolate fundamental reasoning flaws from such errors.

Instead, we use a more lightweight formalization technique. The goal of our formalization is to detect the *local* errors in the LLM's thinking process, which can then be used to guide it to "guess" a correct answer. Specifically, we provide an LLM with the natural language proof and instruct it to translate the key reasoning steps into a sequence of first-order logic implications. As shown in Figure 5, the LLM identifies the initial conditions and expresses the reasoning process as two implications. Each implication is annotated with a comment linking it back to the original proof step. This process avoids the complexities of generating complete proof scripts while producing formal claims that can be validated. Using an SMT solver, we find that the first implication, $(j > m) \implies (j = m + 1)$, is invalid. This step successfully identifies the precise location of the logical flaw in the LLM's thinking process.

Feedback to the LLM. With the error identified, we construct precise feedback. We provide a new prompt to the LLM, stating: "In your reasoning, the step 'At loop termination, j is $m + 1$' is wrong. This is because the condition $(j > m)$ cannot derive $(j == m + 1)$". The feedback includes where the LLM makes mistakes and how it gets wrong. We then instruct it to reconsider its loop invariants in light of the flaw of its reasoning. Based on this feedback, the LLM produces the refined invariants in Figure 3. It correctly adds the missing upper bound

Algorithm 1 Iterative Invariant Refinement Framework.

Require: A single-loop program P, an assertion a, an invariant synthesizing LLM L_S, a formalizing LLM L_F, an invariant verification tool V.
Ensure: A set of inductive loop invariants inv.

```
inv := getInitialInvariants(L_S, P, a)
while true do
    invalidVC := checkInvariants(V, P, a, inv)
    if invalidVC is None then
        break
    naturalProof := getNaturalProof(L_S, P, a, invalidVC)
    formalizedProof := formalizeProof(L_F, naturalProof)
    errors := checkProof(formalizedProof)
    inv := getRefinedInvariants(L_S, P, a, inv, errors)
return inv
```

for j ($j \leq m+1$ as $i3$) without altering the other correct invariants. With this complete set of invariants, the verification task succeeds.

3 Preliminaries

The problem we focus on is loop invariant synthesis. Specifically, we focus on synthesizing loop invariants to verify the post-loop condition for programs with a single loop. Formally, consider a Hoare triple over a loop $\{P\}$ while B do $S\,\{Q\}$, which means that P is the pre-condition that holds before the loop, and Q is the post-condition that holds after the loop if the loop terminates. By the classical Hoare Logic [16], we have

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

This indicates that to prove the Hoare triple $\{P\}$ while B do $S\,\{Q\}$, we need to find an inductive loop invariant I, which satisfies the following three conditions:

(1) *Establishment.* The loop's pre-condition P implies the loop invariant I ($P \implies I$).
(2) *Preservation.* Given the loop invariant I holds before one iteration of the loop, and the loop executes for one more iteration, I must also hold at the end of the loop iteration ($\{I \wedge B\}\, S\, \{I\}$).
(3) *Post-condition.* If the loop exits, the loop invariant must imply the post-condition Q ($(I \wedge \neg B) \implies Q$).

If the above three conditions are satisfied, we say the loop invariant I is an *inductive loop invariant*, and the program property Q can then be proved using I.

4 Overall Framework

Algorithm 1 demonstrates the overall framework of our approach. It proposes loop invariants to verify the assertion a in a single-loop program P. An LLM L_S is used to synthesize the loop invariants, which we refer to as the *Synthesizer LLM*. And an LLM L_F is employed to formalize the natural language given by L_S, which we refer to as the *Formalizer LLM*. The loop invariants and the assertion can be formally checked by the verification tool V.

The algorithm first prompts the Synthesizer LLM L_S to propose invariants directly. The invariants are then checked by V. If the invariants are checked to be inductive, the verification task is completed. Otherwise, the verification tool will return a set of invalid verification conditions, and the algorithm goes into a feedback-refinement iteration. In each iteration, the Synthesizer LLM is first asked to give a step-by-step proof for the

Just Accepted

verification condition that fails. Then, the algorithm prompts the natural language proof to the Formalizer LLM L_F to let it formalize the proof into a sequence of first-order logic implications. The formalized proof can be checked for errors, which are then provided as targeted feedback to guide the Synthesizer LLM to correct the errors and propose refined loop invariants. The iteration proceeds until the inductive loop invariants are found.

5 Our Approach

In this section, we introduce the design of our framework in detail. Specifically, we introduce our approach in the following steps: the primitive "guess-and-check" paradigm and the formal feedback and refinement mechanism. Our approach iteratively applies these two steps until a set of inductive loop invariants is generated.

5.1 The Primitive Guess-and-Check Paradigm

In the primitive "guess-and-check" paradigm, we first ask the Synthesizer LLM to generate a set of candidate loop invariants. Subsequently, we leverage a formal verification tool V to check whether the generated invariants are correct and sufficient to prove the desired program assertion.

Specifically, the process begins by providing the Synthesizer LLM with the program source code P and a post-condition assertion a. We use a carefully crafted prompt, shown in Figure 6, to guide the LLM in proposing a set of loop invariants, $\mathbb{L}$. The model is expected to generate invariants that are not only valid but also strong enough to imply the assertion a.

For the "check" step, we use Frama-C [8] as the formal verification tool V to perform the verification. By building on Frama-C, the generated invariants are ensured valid for a C program. Frama-C verifies the loop invariants and the program assertion by decomposing the overall goal into a set of verification conditions (VCs). Specifically, for a program P with a single loop, a candidate invariant set $\mathbb{L}$ for the loop, and an assertion a after the loop, Frama-C generates VCs corresponding to the three standard proof obligations of Hoare Logic:

(1) **Establishment of Invariants:** For each loop invariant $l \in \mathbb{L}$, a VC is generated to verify that l holds true before the loop, given the program's pre-conditions.
(2) **Preservation of Invariants:** For each loop invariant $l \in \mathbb{L}$, a VC is generated to verify that if all invariants in $\mathbb{L}$ hold at the beginning of an arbitrary loop iteration, l will continue to hold after that iteration completes.
(3) **Implication of Assertion:** For the program assertion a, a VC is generated to verify that upon loop termination, the conjunction of all invariants in $\mathbb{L}$ is sufficient to imply the program assertion a.

Note that Frama-C treats each verification condition as an independent proof obligation. During the verification of a given VC, its premises are assumed to hold, regardless of the proof status of other VCs. This modularity is crucial for pinpointing the exact sources of failure later in our feedback process.

For each VC, Frama-C returns a success or failure result. If all VCs pass the validation, the verification task is considered successful, which confirms that the loop invariants are both correct and sufficient to prove the assertion. Otherwise, we will get a set of failed VCs (denoted as $\mathbb{VC}_{fail}$), which indicates that either some invariants are incorrect (i.e., they fail the establishment or preservation checks) or the current invariants are not sufficient to prove the final assertion.

5.2 Formal Feedback and Refinement

While the primitive "guess-and-check" paradigm can succeed on simple problems, it often fails when the invariants are complex. A key capability of modern LLMs is their ability to refine their outputs based on feedback [21]. However, the nature of the feedback is critical. The prior work [20] has explored using binary (success/failure) feedback to guide the LLM to refine its loop invariants. This approach offers a weak signal, informing the LLM only *that* its proposal is incorrect, but not *why*. It fails to target the root cause of the failure, which is often a

Just Accepted

Prompt

```
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 follow
 the following format:

```c
    assert(...);
    assert(...);
    ...
```
```

Fig. 6. Prompt for the Synthesizer LLM to generate initial invariants.

Just Accepted

subtle logical error or hallucination rather than a complete misunderstanding of the program. As a result, the binary feedback leads to a limited improvement on the LLM.

To overcome this limitation, we propose a formal feedback and refinement mechanism to automatically diagnose the LLM's thinking process and provide targeted feedback. This mechanism forms the core technical contribution of our work.

Our formal feedback and refinement framework proceeds in the following steps:

(1) **Structured Natural Language Proof Generation:** We prompt the Synthesizer LLM to generate its thinking process by generating a step-by-step natural language proof for a failed VC.
(2) **Formalization and Checking:** We translate the Synthesizer LLM's thinking process generated in the above step into a series of formal logical expressions using the Formalizer LLM, and use an SMT solver to automatically verify each reasoning step.
(3) **Feedback and Refinement:** We synthesize the results of the formal check into a report that pinpoints the exact logical errors, which is then provided to the Synthesizer LLM to guide the generation of a refined set of invariants.

5.2.1 Structured Natural Language Proof Generation. The goal of this step is to obtain a natural language proof from the Synthesizer LLM that is both *detailed* enough to expose its underlying thinking process and structured in a way that is *ready to be formalized*.

However, our initial experiments revealed that prompting the LLM to justify the entire verification task at once often results in a coarse-grained proof, which tends to merely re-states the high-level VCs. To obtain more detailed reasoning, our method focuses the LLM on a single failed VC from $\mathbb{VC}_{fail}$ at a time. The specific VC is selected following a natural deductive order: *establishment of invariants, preservation of invariants, and then implication of assertion*. This order is the same as the standard methodology of proof in Hoare Logic. If there are multiple invalid VCs of the same type, a random one is picked.

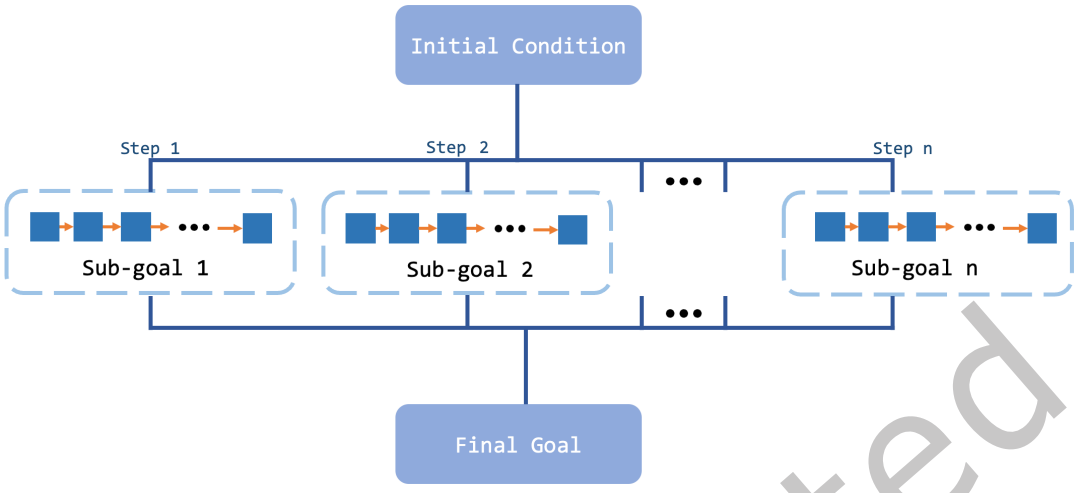

Fig. 7. The structure of the natural language proof.

To ensure the proof is ready to be formalized, we constrain its structure using the prompt shown in Figure 8. The LLM is instructed to first list all initial conditions from the VC, and then decompose its reasoning into a sequence of numbered steps, as depicted in Figure 7. Each step is expected to represent a single, sequential line of reasoning. In other words, if a sub-goal requires case analysis, the LLM is instructed to split the cases into separate steps. This structured format allows each step's core logic to be easily extracted and formalized. As an example, Figure 4 presents a single step of the natural language proof given by the LLM, which performs a sequential reasoning from the known conditions to the program assertion.

In the checking process, our method formalizes and verifies the reasoning within each step. It does not formally verify the high-level case-split structure of the proof itself. The primary goal of our work is to detect and correct the local, detailed logical errors in the LLM's thinking process. As our experiments in Section 6 show, this targeted focus on local reasoning errors is highly effective at guiding the LLM toward a correct solution.

5.2.2 Formalization and Checking. Given the structured natural proof from the Synthesizer LLM, the next task is to formally check it for logical errors. To maintain a clean context separation between loop invariant generation and formalization, we utilize a separate LLM session, the Formalizer LLM, for this translation task.

Previous auto-formalization works [19, 44] focused on targetting the LLM to generate a proof script in languages such as Coq or Isabelle, which can be checked for validity formally and completely. However, LLMs often struggle to generate code that is correct in both logic and syntax. It has been studied that the Coq code generated by an LLM often fails due to low-level issues like wrong theorem application or syntactic mistakes [23]. Besides, the failures in the proof script often mask the actual error in the natural language thinking process. For example, if the Coq code tries to apply the hypothesis H: $\mathrm{m}{=}\mathrm{n}$ to $\mathrm{n}{=}\mathrm{m}$, it will fail with the error message "Unable to unify 'm=n' with 'n=m'.", which does not provide any information for the error in the thinking process. Therefore, we cannot construct effective feedback based on the error of the proof script.

To address these issues, we take a more lightweight formalization strategy. Note that the goal of our approach is not for the LLM to write a completely verifiable proof script — the final "check" is already handled by Frama-C. Instead, we only need to detect local errors in the LLM's thinking process to guide its next "guess". As a result, we prompt the Formalizer LLM to transform the key inference within each proof step into a series of first-order logic implications. This approach has two main advantages. First, in practice, LLMs are significantly more

Just Accepted

Prompt

```
Here are some instructions on step-by-step Structure for your proof:
- The whole proof contains three parts: initial conditions, proof process, and
conclusion. Label them with `[Initial]`, `[Proof]`, and `[Conclusion]`,
respectively.
- In the `[Initial]` part, just list the initial conditions, do not perform any
 logical inference - leave them to the proof process.
- In the `[Proof]` part, label each step explicitly with the format:
 `[STEP N: <STEP_NAME>]` where `N` is the step number and `<STEP_NAME>`
 describes the step. Do not miss the left `[` and the right `]`.
- Under each step, provide detailed mathematical/logical reasoning.
- Each step should be a sequential logical reasoning. For example, if a step
contains subcases, please split them into different steps.
```

Fig. 8. Prompt for the Synthesizer LLM to generate structured natural language proof.

successful at generating syntactically correct expressions in this simple format. Second and more importantly, each implication maps directly back to a specific reasoning step in the natural language proof, which enables precise error attribution.

The formalization consists of three key parts:

- A list of all conditions assumed to be true at the beginning of the proof step.
- A sequence of first-order logic implications that represent the core reasoning. Each implication is annotated with a comment linking it to the original natural language step (see Figure 5).
- The final conclusions derived in the step.

To facilitate automated checking, all conditions and conclusions should be valid C logical expressions. Since the checking can be conducted automatically and rigidly by the checker, we ask the LLM to focus on *translating* instead of checking possible problems in those implications. The prompts used for the formalization task are shown in Figure 9 and Figure 10.

Figure 5 presents an example of this kind of formalization. In the example, two reasoning steps are transformed into two distinct logical implications. Each implication is mapped with the original natural language proof step using a comment.

The formalized proof is then checked automatically. If the focused verification condition is an establishment VC, we first check whether the initial conditions given by the LLM hold by integrating them as assertions in the original program and checking them by Frama-C. This is because sometimes the LLM may incorrectly identify the pre-conditions, such as wrongly assume the range of an undefined variable.

Next, we check the sequence of implications using the procedure in Algorithm 2. Specifically, we maintain a set of known fact *Conds* which is first initialized with the pre-conditions of the proof step given by the LLM. Then, we scan over the implications in the list of implications Imp. For each implication $p \Rightarrow q$, we first check if its premise p is entailed by the current set of known facts ($Conds \models p$). If not, it suggests the LLM uses an incorrect assumption in this specific step. Then, we check the validity of the implication $p \Rightarrow q$ itself using an SMT solver.

After checking an implication, we add its conclusion q to *Conds* for subsequent steps, regardless of whether the implication is valid. In other words, if a step applies a wrongly-derived conclusion of the previous steps, we

Just Accepted

Prompt

```
Please read the above proof for one certain goal. For each step marked with [
STEP N: ...] in the above proof, formalize its logical content into first-order
logic expressions. Follow these rules:

1. When formalizing a certain step, focus on it. Formalize **only** the certain
step, and ignore other steps.
2. List the known conditions before this step. Each condition should be a
logical expression represented in C grammar. The detailed known conditions are
listed as follows:
    - The initial conditions.
    - The conclusions that derived from previous big steps.
    - If this step needs declare a new variable, list it here using "==" symbol.
3. Formalization requirements:
    - Convert natural language claims into precise first-order logic formulas.
    - For each formula, represent it as an implication, using "==>" to represent
    .
    - The condition and the conclusion of the implication should be legal C
    logical expressions, i.e. it can only contain "&&", "||" and "!" as logic
    operators.
    - Each formula should be self-contained so that it can be checked.
    - You do not need to verify the correctness of the proof, just formalize it
    in logic formulas.
    - At last, summarize the conclusion of this step using a C logical
    expression. **Here, the conclusion means the one that are actually derived
    from this step, not the one that need to be derived.**

---
```

Fig. 9. The Prompt Used for Formalization (Part 1 / 2)

Just Accepted

do not treat this as an error. This allows our system to identify multiple, independent errors within a single proof attempt.

5.2.3 Feedback. The final step synthesizes the set of invalid expressions, *Invalid*, into a targeted, diagnostic feedback prompt for the Synthesizer LLM. The goal is to clearly explain the errors to guide the generation of a corrected set of invariants.

The feedback is structured to be maximally informative. For each error found, we provide the LLM with:

(1) The context: The specific verification condition (e.g., preservation of invariant 'i <= n') that was being analyzed.
(2) The location of the error: The specific step number from its natural language proof (e.g., '[STEP 3: ...]') and the reasoning location (given by the comment of the invalid implication).
(3) The specific logical flaw: The formal implication that was found to be invalid, or the initial conditions which are not satisfied.

Prompt

```
Output format:
    - For each step marked with [STEP N: ...], first output the step label in
    one line. It should be the same as the label in the natural proof. Then,
    output the formalization of this step in the following format.
    - First, list the known conditions. This part begins with a `[Initial]`
    label, with one condition per line. Each condition should be a legal C
    logical expression. Each condition follows by a comment starting with "//".
    Follow the following rules for comments:
        1. If the condition comes from initial conditions, comment "initial".
        2. If the condition comes from conclusions derived by previous big steps
        , comment "derived".
        3. If the condition is a declaration of a new variable, comment "
        declaration".
        4. Comment a single word for each condition.
    - Then, output the formalized proof. This part begins with a `[Proof]` label
    , with one implication per line.
    - Output one implication in each line, following by a comment starting with
    "//" which matches the claim in the natural language proof. The condition
    and conclusion must be legal C logical expressions. For example:
      ```
      (a <= 0) && (a == m) ==> (m <= 0) // From a is non-positive, and a is
      equal to m, we know that m is non-positive, too.
      ```
    - At last, output the conclusion of this step. This part begins with a `[
    Conclusion]` label. The conclusion stays within one line, with exactly one C
     logical expression. Similar to the above, represent the explanation to the
    conclusion in a comment starting with "//".
    - Only output the conditions, implications, conclusions, and comments, do
    not explain.
    - For all C logical expressions mentioned above, do not use functions in it.
```

Fig. 10. The Prompt Used for Formalization (Part 2 / 2)

(4) The nature of the flaw: A clear, templated explanation of whether the error was an unsupported assumption (the premise was not provable from prior facts) or a logical non-sequitur (the implication itself does not hold).

An example of this feedback prompt is shown in Figure 11. By pinpointing the exact location and nature of the error in the thinking process, we provide a constructive feedback, moving from simple binary feedback to actively guide the LLM's refinement process.

6 Evaluation

In this section, we evaluate the effectiveness of our approach on a main benchmark suite of 460 C programs and a non-linear benchmark suite of 50 programs. We aim to answer four research questions concerning the effectiveness, efficiency, and mechanisms of our proposed method.

Just Accepted

Algorithm 2 Iterative Checking of Implications.

Require: A set of pre-conditions *Pre*, a list of implications *Imp*.
Ensure: A set of invalid expressions *Invalid*.

```
1: Conds := Pre
2: Invalid := ∅
3: for p ⇒ q ∈ Imp do
4:    if Conds⊭p then
5:       Invalid := Invalid ∪ {p}
6:    if p⊭q then
7:       Invalid := Invalid ∪ {p ⇒ q}
8:    Conds := Conds ∪ {q}
9: return Invalid
```

6.1 Evaluation Setup

Baselines. We compare LORIS with LaM4Inv [39] and Clause2Inv [5], both of which represent state-of-the-art LLM-based approaches for loop invariant synthesis. Both baseline methods operate by prompting an LLM to generate candidate invariants (or invariant components) and subsequently employ symbolic methods to get a correct set of invariants from the LLM-generated answer. Specifically, LaM4Inv iteratively prompts an LLM for loop invariants, then validates these candidates and provides feedback to the LLM in the form of counterexamples generated by an SMT solver. Its key technical contribution is the use of Bounded Model Checking (BMC) to filter and combine valid conjuncts from multiple, partially correct LLM responses. Clause2Inv adopts a different strategy. Rather than generating full invariants, it prompts the LLM to produce a list of atomic clauses, which are then logically combined via conjunction or disjunction. Like LaM4Inv, it relies on counterexample-based feedback to guide the LLM to generate new clauses. Our approach diverges from these baselines in two fundamental ways. First, while both baselines rely on coarse-grained, counterexample-based feedback, our approach provides structured and detailed feedback from formally verifying the LLM's step-by-step thinking process. Second, while the baselines depend heavily on external symbolic methods to assemble the final solution from imperfect outputs, our approach focuses on enhancing the LLM's own reasoning capability to directly synthesize correct invariants. Furthermore, our feedback mechanism is complementary to these symbolic techniques. We evaluate a potential combination between BMC used in LaM4Inv and our approach in RQ4.

Benchmark. Our main benchmark suite consists of 460 C programs that require numerical invariants for verification, encompassing both linear and non-linear properties. This suite is an aggregation of problems from several established sources to ensure diversity and difficulty. The core of our benchmark consists of 313 programs evaluated for both LaM4Inv and Clause2Inv containing only linear properties. We excluded 3 of the original 316 programs due to their use of floating point arithmetic, which is unsupported by the version of Frama-C we use. To challenge our approach beyond the scope of the baselines, which have already achieved a near-perfect solve rate (309/316 for LaM4Inv, and 312/316 for Clause2Inv) on their own benchmarks, we incorporated more difficult problems. We collected additional programs from the Microsoft loop invariant generation experiments [20]. This dataset includes all the benchmarks from previous works - Code2Inv [32, 33], Accelerating Invariant Generation [24], and LinearArbitrary-SeaHorn [45]. Furthermore, we integrated benchmarks from all the sub-directories named with the prefix "loop" in the SV-COMP repository [36]. From these sources, we filtered for all programs containing a single loop and a single function, and removed duplicates to form our final suite of 460 programs. In addition to the main benchmark suite, we also evaluate LORIS on the non-linear benchmark suite

Just Accepted

Prompt

```
In your proof of {step}, "{comment}" is wrong. This is because the condition {
condition} cannot derive {conclusion}.

This indicates that the loop invariants you proposed are all correct, but they
are not sufficient to verify the assertion {assertion}.

Please fix your loop invariants in the following steps:
First, based on the error in your proof, analyze why the loop invariant are not
sufficient to verify the assertion {assertion}.
Then, from your analysis, carefully think about the verification problem again.
Every time you propose a loop invariant, note that do not make the same mistake
again.

In the end, 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. Your answer should follow the following format:

```c
    assert(...);
    assert(...);
    ...
```
```

Fig. 11. The Prompt Used for Feedback.

Just Accepted

from the Clause2Inv paper. This benchmark contains 50 more C programs each of which involves non-linear properties.

Evaluation metrics. To account for the stochastic nature of LLMs, we execute each tool on every benchmark for five independent runs. We consider the following evaluation metrics:

(1) **# Solved Benchmarks:** The total number of unique benchmarks solved in at least one of the five runs.
(2) **Success Rate:** The average success rate, calculated as the total number of successful runs divided by the total number of attempts (i.e., $460 \times 5 = 2300$ for the main benchmark suite).
(3) **Cost on Success:** The average number of tokens (input/output) and time (in seconds) consumed for successful runs only.

To specifically analyze our contribution, we also distinguish between successful runs where the LLM provided a correct invariant in the first attempt ("direct solve") versus those that required one or more rounds of our feedback mechanism ("feedback-driven solve"). We will explain them in detail in Section 6.2.3.

Implementation Details. All of our experiments were conducted on a Linux machine with 256GB memory and 2.6GHz processors. For fairness, we set a uniform time limit of 600 seconds and a token limit of 150,000 tokens per benchmark run for both our approach and the baselines. We used Frama-C version 27.1 to verify the inductiveness of the loop invariants, with a timeout value of 5 seconds per verification condition. The validity of logical implications in our formalized proofs was checked by the Z3 SMT solver [9]. We evaluated our approach on OpenAI's language models. We selected the most recent generative models with different parameter sizes, gpt-4.1 and gpt-4.1-mini, and the older gpt-4o-mini. We also considered one of the most recent and economical reasoning models gpt-o4-mini. To demonstrate the effectiveness of our approach on non-OpenAI models, we further considered Anthropic's claude-3.7-sonnet. For our approach, the same model was used as both the Synthesizer LLM and the Formalizer LLM.

6.2 Evaluation Results

We intend to answer the following research questions (RQs) with our experiments:

(1) *Effectiveness*: How does LORIS compare to the baselines in terms of the number of solved benchmarks and overall success rate?
(2) *Efficiency*: What is the time and token cost of our approach on successful runs?
(3) *Impact of Feedback*: What is the direct contribution of our formal-verification feedback mechanism to the overall success?
(4) *Combining with symbolic methods*: How does combining our approach with symbolic methods impact the verification performance?

6.2.1 RQ1: Effectiveness. Table 1 and Table 2 present the effectiveness of our approach compared to the baseline methods LaM4Inv and Clause2Inv on the main benchmark suite and the non-linear benchmark suite separately. The results show that **our approach consistently outperforms the baselines in the number of solved benchmarks across all the tested LLMs.** On the main benchmark suite, using the most capable model, GPT-4.1, our approach solves 445 of the total 460 programs. This is 31 more than LaM4Inv and 36 more than Clause2Inv, suggesting the effectiveness of our approach. On the non-linear benchmark suite, our approach can solve at most 47 of the 50 programs using Claude-3.7-Sonnet.

Furthermore, LORIS achieves the highest overall success rate on four of the five tested models on the main benchmark suite. With GPT-4.1, our success rate reaches 93.1%, significantly surpassing both LaM4Inv's (84.6%) and Clause2Inv's (84.9%). This advantage holds for other state-of-the-art models like Claude-3.7-Sonnet and the more economical GPT-4.1-mini. An exception is that with the weaker model, GPT-4o-mini, Clause2Inv achieves a slightly higher success rate (82.6%) than LORIS (81.9%). This is because Clause2Inv prompts the LLM to generate simple clauses, a task that is easier for weaker models to perform consistently, leading to a higher success rate on easier problems. However, our approach solves more unique benchmarks than Clause2Inv (431 vs. 408). This result shows that simple clauses fail to capture the complex invariants required for harder problems. In contrast, our feedback mechanism effectively enhances the model's capability, allowing it to eventually solve a larger number of unique, difficult benchmarks that are beyond the reach of the baselines. On the non-linear benchmark, LORIS achieves the highest success rate across all the tested LLMs, suggesting that our approach works better than the baselines on more difficult problems. These outcomes support that providing detailed constructive feedback on the LLM's thinking process is more effective than providing only a counterexample. By pinpointing the exact logical flaws in a proposed proof, our approach enables powerful models to correct their thinking process and converge on a correct solution more reliably.

An interesting observation is that the advantage appears to scale with model capability. Comparing the results of the three generative models of GPT series in Table 1, the performance gap between LORIS and the baselines on the success rate goes larger as the model becomes powerful. This suggests that as LLMs become more powerful,

Just Accepted

Table 1. Effectiveness and cost comparison on the main benchmark suite (460 programs). LORIS is compared against LaM4Inv and Clause2Inv. Costs are averaged over successful runs. Best results for each model are in bold.

Model	Method	# Solved	Success Rate (%)	Time (s)	Tokens (I/O)
GPT-4.1	LORIS	**445**	**93.1**	55.9	8296 / 2738
	LaM4Inv	414	84.6	40.5	4477 / 666
	Clause2Inv	409	84.9	**12.4**	**2344** / **406**
GPT-4.1-mini	LORIS	**439**	**91.1**	52.8	10203 / 3923
	LaM4Inv	412	84.2	52.0	3808 / **528**
	Clause2Inv	405	83.6	**18.4**	**3261** / 706
GPT-4o-mini	LORIS	**431**	81.9	104.3	14302 / 5161
	LaM4Inv	397	76.0	47.0	6572 / 857
	Clause2Inv	408	**82.6**	**20.8**	**3283** / **632**
GPT-o4-mini	LORIS	**435**	**89.2**	92.1	1557 / **4056**
	LaM4Inv	387	74.3	227.9	1890 / 12461
	Clause2Inv	414	85.0	**45.8**	**655** / 5026
Claude-3.7-Sonnet	LORIS	**443**	**89.9**	86.7	7155 / 2186
	LaM4Inv	413	81.4	92.7	**2425** / 1363
	Clause2Inv	409	83.8	**25.0**	3645 / **812**

Just Accepted

our approach is better equipped to leverage their enhanced abilities, hinting great potential for more advanced models in the future.

The results with the reasoning-specialized model, GPT-o4-mini, are particularly noteworthy when comparing with LaM4Inv on the main benchmark suite. While its absolute success rate (89.2%) is lower than that of GPT-4.1, the performance gap between LORIS and LaM4Inv is the largest, at 14.9 percentage points. The lower absolute performance is partly due to the model's higher latency and token usage per turn, which limits the number of possible iterations within the fixed resource limits. However, the significant relative improvement demonstrates that LORIS's structured feedback is highly compatible with the model's intended reasoning capabilities. On benchmarks solved via feedback, GPT-o4-mini requires an average of only 1.19 feedback rounds to find the solution, indicating that the feedback is precise and actionable.

6.2.2 RQ2: Efficiency. As shown in Table 1 and Table 2, the efficiency trade-offs between the approaches are clear. On the generative models (GPT-4.1, Claude, etc.), our approach typically consumes more tokens than the two baselines per successful run. This is a direct consequence of the design. Our method essentially substitutes the computational cost of the symbolic methods used in the baselines with the token cost of generating a detailed reasoning trace from the LLM. On simpler problems, where the baselines are sufficient to solve, this can result in a higher token cost for our method. However, the feedback mechanism of our approach is exactly what enables LORIS to solve more complex problems where the baseline approaches fall short, as shown in RQ1. Note that we set the same resource limit for all methods in the evaluation. The fact that LORIS can solve more difficult benchmarks within the same resource budget demonstrates that its precise feedback mechanism enables the LLM

Table 2. Effectiveness and cost comparison on the non-linear benchmark suite (50 Programs). LORIS is compared against LaM4Inv and Clause2Inv. Costs are averaged over successful runs. Best results for each model are in bold.

Model	Method	# Solved	Success Rate (%)	Time (s)	Tokens (I/O)
GPT-4.1	LORIS	**45**	**83.2**	80.4	17365 / 5430
	LaM4Inv	25	38.8	64.5	8334 / 1310
	Clause2Inv	42	76.0	**35.1**	**3915** / **741**
GPT-4.1-mini	LORIS	**46**	**80.8**	74.1	11878 / 4573
	LaM4Inv	29	44.8	73.2	8591 / 1383
	Clause2Inv	42	73.2	**24.3**	**2691** / **638**
GPT-4o-mini	LORIS	**44**	**77.6**	125.2	12007 / 4407
	LaM4Inv	27	36.0	60.9	8907 / 1188
	Clause2Inv	42	70.4	**40.3**	**4169** / **934**
GPT-o4-mini	LORIS	**47**	**87.6**	**61.8**	1548 / **4802**
	LaM4Inv	30	48.8	222.8	3241 / 20218
	Clause2Inv	46	83.6	61.9	**785** / 5515
Claude-3.7-Sonnet	LORIS	**47**	**79.2**	88.0	13784 / 4752
	LaM4Inv	29	44.4	95.7	11524 / 1965
	Clause2Inv	43	78.8	**37.9**	**4417** / **1125**

to discover non-trivial solutions that are inaccessible to the baselines. Furthermore, despite the higher token count, the monetary cost remains acceptable. For instance, a successful solve with GPT-4.1 costs approximately $0.038 on average.

Regarding time costs, Clause2Inv is generally the most efficient. This is because generating atomic clauses is a much simpler task than synthesizing complete invariants, resulting in lower time cost when solving easy problems. However, it fails to solve more complex problems within the same 600 seconds limit compared to our approach. For our approach and LaM4Inv, the time costs are generally comparable, with no approach having a consistent advantage across all models. The runtime of our approach is dominated by LLM inference time, while LaM4Inv's runtime is a mix of LLM inference and the computational cost of BMC.

It is interesting to note that when using the reasoning model GPT-o4-mini, LORIS is the most efficient in term of token cost among the three methods, and the time cost is much lower than LaM4Inv. For the reasoning model, it is much more costly for the model to generate one response. LaM4Inv's reliance on multiple iterations leads to an average time of 227.9s. Clause2Inv also requires multiple iterations to get a complete loop invariant, leading to a higher output token cost. In contrast, LORIS's ability to guide the model to a solution in fewer iterations (as noted in RQ1) results in a 2.5x time reduction than LaM4Inv and lower token usage, demonstrating the efficiency for models that excel at reasoning but may be slower to respond.

6.2.3 RQ3: Impact of Feedback. For some problems, the LLM can produce correct invariants directly from the initial prompt. Therefore, to isolate the specific contribution of our feedback mechanism, we analyze how many problems are actually solved by our feedback mechanism. Table 3 partitions the solved problems into those solved

Just Accepted

Table 3. Classification of solved problems on the main benchmark suite. Feedback-Enhanced is a subset of Directly Solvable where feedback further improved consistency across runs.

Model	Directly Solvable	Feedback-Enhanced	Feedback-Exclusive
GPT-4.1	305	112	140
GPT-4.1-mini	258	103	181
GPT-4o-mini	242	195	199
GPT-o4-mini	324	316	111
Claude-3.7-Sonnet	322	199	121

Table 4. Analysis of feedback contribution to successful runs on the main benchmark suite. A "Direct Success" occurs on the first attempt, while a "Feedback-Driven Success" requires one or more feedback iterations.

Model	Total Success	Direct Success	Feedback-Driven Success
GPT-4.1	2141	1259 (58.8%)	882 (41.2%)
GPT-4.1-mini	2095	1028 (49.1%)	1067 (50.9%)
GPT-4o-mini	1884	722 (38.3%)	1162 (61.7%)
GPT-o4-mini	2052	700 (34.1%)	1352 (65.9%)
Claude-3.7-Sonnet	2067	1089 (52.7%)	978 (47.3%)

Table 5. The refinement success rate results on the main benchmark suite. The refinement success rate means the percentage of feedback iterations that resulted in an invariant with higher Jaccard similarity to the correct answer.

Model	Refinement Success Rate
GPT-4.1	70.4%
GPT-4.1-mini	71.4%
GPT-4o-mini	62.9%
GPT-o4-mini	79.6%
Claude-3.7-Sonnet	80.2%
Average	**72.9%**

Just Accepted

on the initial attempt versus those that require feedback on the main benchmark suite. Specifically, we classify the problems into the following three categories:

(1) *Directly Solvable*: Problems where the LLM produces a correct loop invariant with the initial prompt in at least one run.
(2) *Feedback-Exclusive*: Problems where the LLM fails to produce a correct loop invariant with the initial prompt in all five runs, but are successfully solved via our feedback mechanism. Feedback is the only path to a solution for these problems.
(3) *Feedback-Enhanced*: A subset of the "Directly Solvable" category, where the LLM succeeds with the initial prompt in some runs but not all, and our feedback mechanism successfully repairs the initial failures in more runs. These problems can be solved occasionally without feedback but the success consistency can be greatly boosted using feedback.

The data in Table 3 demonstrates that our feedback mechanism significantly improves model performance. For GPT-4o-mini, 199 problems are solved exclusively by the feedback mechanism, reaching an 82.2% increase over the model's direct solving capability. For more powerful models like Claude-3.7-Sonnet, the feedback mechanism still yields a 37.6% improvement. Besides, our feedback mechanism narrows the gap between less capable models and the state-of-the-art ones. For instance, while the total number of problems solved by GPT-4.1-mini is only 6 fewer than GPT-4.1 using our approach (439 vs. 445), the number of problems solved *directly* by GPT-4.1-mini is 47 fewer than GPT-4.1. In summary, our approach makes strong models stronger to solve more complex problems, and brings the performance of weaker models much closer to the state-of-the-art ones.

Beyond expanding capability, the feedback mechanism significantly improves stability. Since LLM generation is stochastic, a model may not succeed in all five runs directly for a given problem. The "Feedback-Enhanced" column in Table 3 lists the number of problems that can be solved directly, but where the success rate is improved by our feedback mechanism. The results indicate that a large proportion of "Directly Solvable" problems benefit from refinement. For example, on GPT-o4-mini, 316 of the 324 direct-solve problems are improved by feedback in at least one run. Table 4 further breaks down all successful runs into direct successes and feedback-driven successes. For GPT-o4-mini, 65.9% of the total successful runs are achieved via feedback. These results confirm that our approach stabilizes LLM performance, ensuring reliability on problems the model can theoretically solve on its own.

Finally, we inspect the quality of the refinement steps to determine how often the feedback leads to a better invariant. We measure the refinement success rate in Table 5, defined as the percentage of feedback iterations where the refined invariant is closer to the correct answer than the previous proposal. Specifically, we treat a loop invariant as the conjunction of a set of atomic clauses and calculate the Jaccard similarity ($J(A, B) = |A \cap B|/|A \cup B|$ for two sets A and B) between the proposed invariant and the correct answer. Clause matching is performed syntactically: two clauses are considered identical if they are identical after normalization (e.g., whitespace and operator canonicalization). The higher the Jaccard similarity is, the closer the proposed loop invariant is to the correct answer.

As shown in the table, our feedback is highly constructive. The average success rate reaches 72.9%. For GPT-4.1, 70.4% of the refinement iterations result in an invariant closer to the correct answer. This rate is even higher for Claude-3.7-Sonnet and o4-mini. This high success rate confirms that the formal verification feedback provides constructive signals for the LLM toward the correct solution efficiently.

6.2.4 RQ4: Combining with Symbolic Methods. The motivation of our approach is to guide the LLM to generate a complete correct answer in the "guess-and-check" paradigm using constructive feedback. However, in the loop invariant synthesis problem, LLMs often produce partially correct conjuncts across several attempts. LaM4Inv's BMC-based combination strategy is designed to exploit this behavior. Since this technique is orthogonal to our feedback loop, it is natural to combine them. Specifically, after each feedback round, we use BMC to collect all valid conjuncts from the LLM's new response and add them to a growing set of answer set, which can serve as a new candidate invariant.

Table 6 shows the results of this experiment on the main benchmark suite. The combination yields a marginal improvement in the overall success rate (e.g., +0.5% for GPT-4.1). The more significant benefit is in efficiency. For GPT-4.1-mini, the hybrid approach reduces the input and output tokens by approximately 26% and 25%, respectively.

This outcome indicates that while our targeted feedback is powerful enough to guide the LLM to the full solution eventually, BMC can accelerate convergence by assembling the final invariant from partial pieces generated in earlier, less-successful iterations. This reduces the number of feedback loops required for some problems, thereby saving time and tokens. However, the small increase in total solved benchmarks suggests that for the

Just Accepted

Table 6. Effect of combining LORIS with Bounded Model Checking (BMC). The solved runs by BMC are included in those by feedback.

Method	Success Rate (%)	# Solved (Feedback)	# Solved (BMC)	Tokens (I/O)	Time (s)
GPT-4.1	93.1	882	0	8296 / 2738	55.9
GPT-4.1 + BMC	**93.6**	921	162	**7929 / 2643**	50.7
GPT-4.1-mini	91.1	1067	0	10203 / 3923	52.8
GPT-4.1-mini + BMC	**91.6**	1059	283	**7525 / 2930**	55.3

most difficult problems, simply combining conjuncts is insufficient. The corrective guidance of our feedback mechanism remains essential for finding the missing, non-trivial parts of the invariant.

7 Discussion

In this section, we analyze the limitations and reliability of our proposed framework, and how our approach can generalize to more complex scenarios. Following the experimental results in Section 6, we first conduct a detailed failure analysis to identify the root causes preventing the successful verification of the remaining benchmarks (Section 7.1). Next, we discuss the correctness of the auto-formalization component through a manual validation (Section 7.2). Then, we provide guidelines for generalizing our approach to more complex programs with nested or multiple loops and memory manipulation (Section 7.3). Finally, we discuss the broader threats to validity regarding our experimental design and benchmark selection (Section 7.4).

7.1 Failure Analysis

For the benchmarks that our method failed to solve in the evaluation of Section 6, we conducted a manual inspection to better understand the limitations of our approach and get clues for the future work. We found that a small subset of failures resulted from tool-specific limitations, such as Frama-C's handling of specific C features (e.g., `unsigned int` casting), or the benchmark code itself cannot be verified. For the remaining failures, the root causes fall into two primary categories: (1) the inability of the feedback to guide the LLM's global reasoning, and (2) ineffective refinement strategies where the LLM weakens invariants rather than strengthening them to satisfy verification conditions. We illustrate these two failure modes via the following case studies.

7.1.1 Failure on Global Reasoning. The first failure category occurs when the problem requires understanding the "big picture", for example, identifying dead code or global constraints that local reasoning steps miss. Our feedback mechanism focuses on local logical errors of the LLM's thinking process, which helps reduce hallucination, but it is less effective when the LLM's fundamental understanding of the control flow is incorrect.

Figure 12 presents a program that our method failed to solve using GPT-4.1, and Figure 13 shows the correct loop invariants needed to verify the assertion. The loop contains three branches, and proving the assertion requires realizing the third branch (containing $y = -y$) is unreachable. This is deduced from the global upper bound $y \leq 100 * x$ and $z = 10 * w$. The branch condition requires $y > 10 * w$, which implies $y > z$. However, since $y \leq 100 * x$ and the branch also requires $z \geq 100 * x$, this leads to a contradiction. Because this branch is unreachable, the variable y is never negated and remains non-negative, making the assertion provable.

However, the LLM failed to identify this unreachability. As shown in Figure 14, it proposed invariants *i2* and *i5* based on the assumption that y could be negative. Once this incorrect assumption is made, the preservation must be verified across all three branches, which fails in this case. After feedback, the LLM attempted to fix the logic for negative y rather than realizing y cannot be negative. This highlights a limitation of our method: while our method effectively repairs local reasoning errors, it cannot currently prompt the LLM to re-think the global

Just Accepted

```
//Pre-condition: the values of
    variable x, y, w, z are all 0.
while(unknown()) {
    if(unknown()) {
        x++; y += 100;
    } else if(unknown()) {
        if(x >= 4) {
            x++; y++;
        }
    } else if (y > 10 * w) {
        if(z >= 100 * x)
            y = -y;
    }
    w++;
    z += 10;
}

assert((x < 4) || (y > 2));
```

Fig. 12. Example program where our approach failed on global reasoning.

```
/*@
loop invariant i1: z == 10 * w;
loop invariant i2: y <= 100 * x;
loop invariant i3: (x < 4) ==> (y == 100 *
    x);
loop invariant i4: (x >= 4) ==> (y >= 400);
*/
```

Fig. 13. Correct loop invariants for the program.

```
/*@
loop invariant i1: x >= 0;
loop invariant i2: y >= 0 || ( y < 0 && z
    >= 100 * x);
loop invariant i3: z == 10 * w;
loop invariant i4: w >= 0;
loop invariant i5: (x < 4) || (y > -100 * x
    );
*/
```

Fig. 14. Loop invariants proposed by the LLM.

Just Accepted

reasoning of the problem. However, as LLMs become more and more powerful, our method will be more helpful to leverage their abilities, as shown in our evaluation.

7.1.2 Ineffective Refinement. The second failure category involves the LLM's inability to repair invariants when preservation fails. Instead of strengthening the invariant to complete the inductive proof, the LLM sometimes attempts to weaken the invariant or delete the invariant entirely.

Figure 15 illustrates this scenario. The program execution follows two distinct traces determined by *flag*: if *flag* is set, k remains 0; otherwise, j and k maintain a linear relationship. The LLM correctly identified the two states but combined them into a single disjunctive invariant: $(j = 2 * k + 2) \vee (k = 0)$ (Figure 17). While this property holds, it is not inductive on its own. To prove preservation, the verifier must know *which* branch of the disjunction holds to determine the valid transition. For example, if $j = 2 * k + 2$, executing the *if(flag)* branch breaks the relationship unless we know *flag* prevents the linear relation trace. The correct solution requires conditioning the invariant on *flag* (see Figure 16).

When our system provided feedback that preservation failed, the LLM did not introduce *flag* to split the cases. Instead, it attempted to weaken the invariant to a congruence modulo 4 (e.g., $(j - 2 * k - 2)\%4 = 0$), or simply deleted the invariant. While the weakened invariant is preserved, it is too weak to prove the final assertion. This suggests that future work should explore guiding the LLM to refine the loop invariants for inductive verification.

7.2 Correctness of Auto-Formalization

A core component of our framework is the auto-formalization of the Synthesizer LLM's natural language proof into first-order logic implications. A significant concern is the potential for the Formalizer LLM to hallucinate or misinterpret the natural language reasoning. If the translation is unfaithful, the resulting feedback could pinpoint non-existent errors or miss actual flaws.

To mitigate this risk, we have implemented several design strategies. First, instead of requesting a complete proof script (e.g. in Coq or Isabelle), which necessitates strict adherence to complex syntax and global context,

```
int main() {
    int j, k, flag;
    j = 2;
    k = 0;

    while(unknown()) {
        if(flag)
            j = j + 4;
        else {
            j = j + 2;
            k = k + 1;
        }
    }

    assert((k == 0) || j == 2 * k +
        2);
}
```

Fig. 15. Example program where our approach failed by ineffective refinement.

```
/*@
loop invariant i1: k >= 0;
loop invariant i2: j >= 2;
loop invariant i3: (flag == 0) ==> (j == 2
    * k + 2);
loop invariant i4: (flag != 0) ==> (k == 0)
    ;
*/
```

Fig. 16. Correct loop invariants for the program.

```
/*@
loop invariant i1: k >= 0;
loop invariant i2: j >= 2;
loop invariant i3: j == 2 * k + 2 || k ==
    0;
*/
```

Fig. 17. Loop invariants proposed by the LLM.

Just Accepted

we instruct the LLM to translate individual reasoning steps into first-order logic implications. This task is much simpler than generating a full proof script and is well-suited for our purpose of detecting local reasoning errors in the LLM's thinking process. Second, we prompt the Synthesizer LLM to generate the natural language proof step-by-step, allowing the Formalizer LLM to focus on one specific step at a time. This decomposition reduces the problem size, thereby enhancing the precision of the formalization procedure. We also carefully design the prompts to describe the task accurately.

However, this procedure cannot always be perfect, and the LLM can still make mistakes or hallucinate during formalization. To empirically validate the correctness of this procedure, we conduct a manual evaluation on a random sample of ten benchmark runs from our evaluation suite using GPT-4.1. For each run, we manually inspect every generated formalization step to verify whether the logical implications generated by the LLM accurately reflect the meaning of the corresponding natural language proof of the step. The results of this inspection are summarized in Table 7. Across the sampled runs, the framework generates a total of 108 formalization steps. Our manual review finds that 90 of these steps (83.3%) are correctly formalized.

We also analyze the incorrect formalizations to understand the causes of failure. We observe that the LLM performs well with pure mathematical reasoning but struggles with specific structural or semantic translations. The primary sources of error were:

(1) *Case Analysis:* Challenges arise during case analysis. In some instances, a natural language step merely states a case division (e.g., claiming the domain can be divided into $x > 4$ and $x \leq 4$). However, the LLM occasionally repeats the overall proof goal within these case definitions, rendering the implication invalid.
(2) *Program Semantics:* The LLM sometimes fails to distinguish between program state updates and logical equality. For example, it may formalize a variable update as $x = x + 1$. While valid in C syntax, this is a contradiction in standard mathematical logic. Although we explicitly address this in the prompt, the model occasionally remains confused.

Table 7. Manual evaluation of auto-formalization correctness on 108 sampled proof steps.

Benchmark	Total Steps	Correct Steps
cav/gulv_simp	7	5
pie/hola/10	15	12
loop-lit/cggmp2005	10	9
loop-zilu/benchmark15_conjunctive	11	11
dagger/cars	11	9
code2inv/29	14	9
llreve/barthe_merged_safe	7	7
loop-acceleration/diamond_2_2	7	6
SyGuS/153	17	15
loop-invariants/eq1-2	9	7
Total	**108**	**90 (83.3%)**

(3) *Context Omission:* The Formalizer LLM occasionally fails to include implicit pre-conditions mentioned in the natural language (e.g., omitting a variable bound previously established). This leads to implications that are logically invalid despite being semantically aligned with the text.

Errors by the Formalization LLM can introduce "noise" into the feedback mechanism. If the Formalizer LLM produces an invalid implication from a correct natural language step (a false positive error detection), the resulting feedback may pinpoint an error where the Synthesizer LLM is actually correct. On the contrast, if the Formalizer LLM incorrectly validates a flawed step (a false negative), the error in reasoning is missed. Additionally, our formalization does not verify structural errors in the proof. We employ specific strategies to mitigate the impact of this noise. In the case of false positives, we choose to report all errors found in the formalized proof. While this may include spurious feedback, it ensures that genuine reasoning errors are not suppressed. In the case of false negatives, even if a specific reasoning step is missed, the feedback mechanism can at least identify the failure at the level of the specific verification condition (e.g., preservation failure).

While mistakes in auto-formalization cannot be entirely eliminated, their impact on the overall framework is limited to efficiency rather than soundness. Since the final acceptance of loop invariants is strictly guarded by the formal verification tool (Frama-C), LORIS will never accept an incorrect solution based on faulty auto-formalization. The worst-case scenario is a delay in convergence, not a verification failure. The high accuracy rate of the sampled auto-formalization (83.3%) and the strong overall results presented in Section 6 suggest that for the majority of iterations, the feedback provided is accurate and constructive.

7.3 Generalizing to Complex Scenarios

While the current implementation of LORIS focuses on numerical problems with single loops to establish a fair comparison with state-of-the-art baselines, the underlying methodology can generalize to more complex programs. The core principle - guiding an LLM by formally verifying its local reasoning - is generic and applicable to more sophisticated verification scenarios. In this section, we discuss how our approach can generalize to real-world C programs. The generalization includes two main directions: nested or multiple loops and complex memory logic.

7.3.1 Nested or Multiple Loops. As standard verification tools like Frama-C have the ability to verify programs with nested or multiple loops through their loop invariants, our approach can scale to those programs seamlessly. Specifically, for a program with nested or multiple loops, we prompt the LLM to propose loop invariants for all

Just Accepted

loops simultaneously. Frama-C will generate the establishment and preservation verification conditions (VCs) for each loop invariant. Similarly to the approach in Section 5.2, we select a single failed VC and let the LLM express the idea to prove it in natural language. The order for selecting the specific failed VC still follows the natural deductive order: from prior loops to posterior loops, and from inner loops to outer loops. Then the natural language proof is formalized and checked, and the feedback is constructed. The refinement will require propagating the error to other loops, since the proofs for the invariants of multiple loops are dependent with each other. For example, proving the establishment of an inner loop invariant often requires strengthening the invariant of the enclosing outer loop.

To illustrate this, we provide a case study. Figure 18 presents a program with a nested loop. This code implements the Extended Euclidean Algorithm, which finds the greatest common divisor (GCD) of two numbers x and y. The assertion to prove is the linear combination $p * x + r * y = a$. This problem represents a challenge of nested loops because the correctness of the inner loop depends on the bounds maintained by the outer loop.

We chat with Google's Gemini-3-Flash, and the LLM produces the loop invariants for the outer loop and the inner loop in Figure 19 and Figure 20. The LLM correctly identifies the linear relations for both the outer loop and the inner loop. However, the verification fails on the establishment of the inner loop invariant $c \geq 0$. While semantically correct, Frama-C cannot verify this property because the necessary context that a (assigned to c at line 9) is non-negative is missing from the outer loop's invariant.

We then prompt the LLM to prove $c \geq 0$ holds at the start of the inner loop. In the natural language proof, the LLM erroneously relies on the initial program state ($c = 0$ at line 6) rather than the state at the start of the current outer loop iteration. This condition ($c = 0$) is then listed as the initial condition in the formalized proof, and can be verified invalid by Frama-C following the approach in Section 5.2. After the feedback, the LLM understands that it misuses the precondition, and strengthens the outer loop invariants to include $a \geq 0$ and $b \geq 0$, enabling the successful verification of the entire program. This case study demonstrates that our local reasoning feedback can scale to programs with nested loops.

7.3.2 Heap Manipulating. The other direction of generalization is scaling beyond numerical domains to programs involving complex data structures and memory manipulation. This requires extending the underlying logical framework to Separation Logic.

To adapt our framework for broader logical theories, the pipeline requires two modular updates. First, the Formalizer LLM must be prompted to support theories specific to memory reasoning, such as spatial conjunctions for heap representations. The recent work [22] has demonstrated that LLMs possess a foundational understanding of Separation Logic and can generate valid Separation Logic assertions. Second, the verification backend must be substituted with a solver capable of handling these theories, such as CVC5 [2] or SongBird [37]. With these modifications, the core mechanism of LORIS remains unchanged: the Synthesizer LLM proposes memory invariants, and the feedback loop corrects local reasoning errors regarding heap validity. As LLMs perform poorly on generating memory loop invariants directly [22], our approach has the potential to improve it.

7.4 Threats to Validity

We identify several threats to the validity of our study and the limitations of our current approach.

External Validity. The primary threat to external validity concerns the generalizability of our benchmarks. Our evaluation datasets consists of a main benchmark suite of 460 C programs and a non-linear benchmark suite of 50 C programs. While this suite covers a diverse range of linear and non-linear numerical properties, the programs are limited to single-loop functions involving numerical arithmetic. The benchmark does not contain programs involving complex data structures such as arrays, pointers, or recursive data types due to the limited capabilities of SMT solver. Consequently, while LORIS demonstrates high effectiveness on numerical invariants, its performance

Just Accepted

```c
int main() {
    int x, y;
    int a, b, p, q, r, s, c, k;
    a = x; b = y;
    p = s = 1;
    q = r = c = k = 0;

    while (b != 0) {
        c = a;
        k = 0;
        while (c >= b) {
            c = c - b;
            k = k + 1;
        }
        a = b;
        b = c;

        long long temp;
        temp = p;
        p = q;
        q = temp - q * k;
        temp = r;
        r = s;
        s = temp - s * k;
    }
    assert (p * x + r * y == a);
    return a;
}
```

Fig. 18. Example program with nested loops.

```c
/*@
loop invariant a == p * x + r * y;
loop invariant b == q * x + s * y;
*/
```

Fig. 19. Loop invariants for the outer loop proposed by the LLM.

```c
/*@
loop invariant c >= 0;
loop invariant a == k * b + c;
*/
```

Fig. 20. Loop invariants for the inner loop proposed by the LLM.

Just Accepted

on real-world software systems involving heap manipulation or complex memory models remains unverified. However, we posit that the proposed feedback mechanism—identifying and correcting local reasoning errors via auto-formalization—is generally applicable to these more complex scenarios. The feedback mechanism is agnostic to the specific program domain. While the current pipeline is constrained by the solver, the methodology of guiding LLMs via formal verification feedback is expected to generalize to broader classes of programs. We discuss how to generate our approach to more complex scenarios like nested or multiple loops and more complex logic in Section 7.3.

Construct Validity. A core component of our framework is the *auto-formalization* step, where an LLM translates natural language reasoning into first-order logic implications. A threat to construct validity is the fidelity of this translation. If the Formalizer LLM hallucinates or misinterprets the natural language step, the resulting feedback to the Synthesizer LLM may be misleading. Although we mitigate this by instructing the model to perform a lightweight, literal translation rather than generating complex proof scripts, the translation process is not formally verified. However, it is important to note that this threat affects only the *efficiency* of the synthesis loop, not the *soundness* of the final result. Because the final invariants are always verified by Frama-C, incorrect feedback might delay convergence or lead to a timeout, but it will not result in the acceptance of an incorrect invariant. Besides, we have validated that the accuracy of the auto-formalization step through a manual inspection in Section 7.2.

Internal Validity. To mitigate the stochastic nature of LLMs, we performed five independent runs for each benchmark and reported the aggregated results. In terms of baselines, we compared LORIS with Lam4Inv and Clause2Inv. While these works represent the state-of-the-art in LLM-assisted loop invariant synthesis, the

rapid evolution of this field means newer techniques may exist. Additionally, our reliance on Frama-C and Z3 implies that our tool's performance is upper-bounded by the capabilities of these underlying solvers. As noted in Section 7.1, some failures were attributed to Frama-C's handling of specific C features rather than a failure of our logic.

8 Related Work

Our work integrates Large Language Models with formal verification feedback on the model's thinking process to synthesize loop invariants. It is related to several research areas, including loop invariant synthesis, the application of LLMs to synthesize loop invariants, and auto-formalization.

Traditional Loop Invariant Synthesis. The automatic synthesis of loop invariants is a long-standing challenge in program analysis and verification. Traditional approaches largely follow the "guess-and-check" paradigm, in which a loop invariant is first proposed and then checked for validity. Symbolic methods, such as those based on abstract interpretation [4, 7] and predicate abstraction [18, 25], systematically explore a state space defined by abstract domains or pre-defined predicates. Template-based methods [14, 35] presuppose a parametric form for invariants and use solvers to find suitable parameter values. While effective, these methods are often constrained by the expressiveness of their pre-defined templates or abstract domains. Data-driven approaches, on the other hand, infer invariants from program execution traces [13, 28], or use machine learning models [31, 32, 42, 43] trained on large datasets of programs and their invariants. These methods can discover complex invariants but typically require substantial and domain-specific training data. In contrast, our approach leverages a pre-trained LLM, which does not require task-specific training and is not confined to rigid templates, allowing for the generation of more diverse and intuitive candidate invariants without heavy feature-engineering.

LLMs for Loop Invariant Synthesis. With their remarkable success in code generation and understanding, LLMs have recently been applied to various programming tasks. Several studies have explored using LLMs for loop invariant generation. Kamath et al. [20] directly leverage LLMs to generate inductive loop invariants and provides the LLM with a pass/fail feedback. Wen et al. [38] propose an approach that synthesizes program specifications in addition to loop invariants for verification. Wu et al. [40] employ LLMs to generate program properties that help the verification task. LaM4Inv [39] uses bounded model checking to filter correct invariants from multiple responses of the LLM, and provides counterexamples generated by an SMT solver when an invariant fails verification as feedback. Clause2Inv [5] prompts the LLM to generate atomic clauses rather than complete loop invariants, and then logically combine them using a heuristic-based algorithm. It also provides counterexamples as feedback to the LLM. Compared to previous works, which give a simple pass/fail or counterexample feedback, our approach goes further by analyzing the LLM's own thinking process to explain why it is wrong. By identifying the specific logical flaw in the LLM's thinking process, we provide a more fundamental and constructive form of feedback.

Auto-formalization by LLMs. A key component of our framework is the formalization of the LLM's natural language proof. This relates to the field of auto-formalization, which aims to translate informal, natural language text into formal, machine-readable logic. Recent efforts have explored using LLMs to generate formal proof scripts for interactive theorem provers like Coq [3], Isabelle [26], or Lean [10]. Jiang et al. [19] develop a "Draft, Sketch, and Prove" framework, which maps informal proofs to formal proof sketches using LLMs, and then uses the sketches to guide an automated prover to search for proofs for easier sub-problems. Zhou et al. [44] solve mathematical quantitative reasoning problems by transforming the problem statement and solution into Isabelle, and let an LLM give the formal proof for checking. Lu et al. [23] first prompt an LLM to generate an initial Coq proof, and then leverage targeted symbolic methods to repair the low-level problems. Compared to the auto-formalization works, the purpose of our approach is different. Our work does not aim to generate a

Just Accepted

complete, verifiable proof script as the end product. Instead, we guide the LLM towards the correct solution in the "guess-and-check" problem by pointing out the logical error in its thinking process. This leads to the lightweight formalization which can be checked by an SMT solver.

9 Conclusion and Future Work

In this paper, we addressed a fundamental challenge in applying Large Language Models to "guess-and-check" problems: while LLMs excel at generating high-level, intuitive solutions, they often fail to ensure their correctness due to hallucinations and logical errors. We proposed a novel paradigm that guides an LLM to refine its own solution by providing formal verification feedback on the local and detailed errors in its thinking process.

We instantiated this paradigm in a framework called LORIS for synthesizing loop invariants in C programs. LORIS prompts the LLM not only for a candidate invariant but also for a step-by-step natural language proof of its correctness. This proof is then formalized into a series of lightweight, first-order logic implications, which are then checked by an SMT solver to pinpoint the exact logical flaws. These flaws are then presented to the LLM as precise, constructive feedback for refinement. Our evaluation on a main benchmark suite of 460 programs and an additional benchmark suite of 50 programs each of which involves non-linear properties demonstrates the effectiveness of this approach. Using GPT-4.1, LORIS solved 445 programs, achieving a success rate of 93.1% on the main benchmark suite. On the more challenging non-linear benchmark suite, our approach can solve 47 of the 50 programs.

Our work opens several promising avenues for future research. A natural next step is to extend LORIS to handle more complex verification challenges. This includes synthesizing invariants for programs with multiple or nested loops, which requires reasoning about the interplay between different invariants, as well as handling programs with complex data structures and heap manipulations, which would necessitate a more expressive logic for the invariants and proofs.

More broadly, the core feedback paradigm proposed in this paper is not limited to loop invariant synthesis. We believe it can be generalized to a wide range of "guess-and-check" problems where LLMs can provide an initial solution sketch. For instance, in automated theorem proving, our method could be used to ask an LLM to justify a difficult proof step, formalize its explanation, and identify the logical leap. Exploring these and other domains would be a valuable direction to further validate and generalize the effectiveness of guiding LLMs through a formal critique of their own reasoning.

Data Availability Statement

We release the replication package of LORIS on https://github.com/ltcRandomwalk/LORIS. It includes all the source code, scripts, benchmarks and evaluation results.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback. This work was supported in part by the National Natural Science Foundation of China under Grant No. W2411051, and Ant Group Research Fund.

References

[1] Timos Antopoulos, Andrew Ruef, and Michael Hicks. 2016. A Counterexample-guided Approach to Finding Numerical Invariants. (2016).

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

Just Accepted

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent-vigouroux, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. 1997. The Coq Proof Assistant Reference Manual : Version 6.1. (06 1997).

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. doi:10.1145/780822.781153

[5] 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 (June 2025), 22 pages. doi:10.1145/3728920

[6] Fengxiang Cheng, Haoxuan Li, Fenrong Liu, Robert van Rooij, Kun Zhang, and Zhouchen Lin. 2025. Empowering LLMs with Logical Reasoning: A Comprehensive Survey. arXiv:2502.15652 [cs.AI] https://arxiv.org/abs/2502.15652

[7] 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* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388.

[11] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[12] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie,

Under Review

Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. *DeepSeek-V3 Technical Report.* arXiv:2412.19437 [cs] doi:10.48550/arXiv.2412.19437

[13] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 213–224. doi:10.1145/302405.302467

[14] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland) *(ICSE '00)*. Association for Computing Machinery, New York, NY, USA, 449–458. doi:10.1145/337180.337240

[15] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87.

[16] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[17] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2, Article 42 (Jan. 2025), 55 pages. doi:10.1145/3703155

[18] Ranjit Jhala and K. L. McMillan. 2006. A practical and complete approach to predicate refinement. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Vienna, Austria) *(TACAS'06)*. Springer-Verlag, Berlin, Heidelberg, 459–473. doi:10.1007/11691372_33

[19] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=SMa9EAovKMC

[20] 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:2311.07948 [cs.PL] https://arxiv.org/abs/2311.07948

[21] Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. *Transactions of the Association for Computational Linguistics* 12 (2024), 1417–1440. doi:10.1162/tacl_a_00713

[22] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. 2024. Towards general loop invariant generation: a benchmark of programs with memory manipulation. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 4101, 26 pages.

[23] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. doi:10.1145/3691620.3695521

[24] Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam Srivas. 2015. Accelerating invariant generation. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design* (Austin, Texas) *(FMCAD '15)*. FMCAD Inc, Austin, Texas, 105–111.

[25] Kenneth L. McMillan. 2010. Lazy annotation for program testing and verification. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 104–118. doi:10.1007/978-3-642-14295-6_10

[26] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg.

[27] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan

Just Accepted

Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. *GPT-4 Technical Report.* arXiv:2303.08774 [cs] doi:10.48550/arXiv.2303.08774

[28] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *SIGPLAN Not.* 51, 6 (June 2016), 42–56. doi:10.1145/2980983.2908099

[29] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1144, 25 pages.

[30] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2019. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. arXiv:1909.11542 [cs.LG] https://arxiv.org/abs/1909.11542

[31] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2019. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. arXiv:1909.11542 [cs.LG] https://arxiv.org/abs/1909.11542

[32] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (2018), Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7762–7773. https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html

[33] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II* (2020) *(Lecture Notes in Computer Science, Vol. 12225)*. Springer, 151–164. doi:10.1007/978-3-030-53291-8_9

[34] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 151–164.

[35] Fabio Somenzi and Aaron R. Bradley. 2011. IC3: where monolithic and incremental meet. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design* (Austin, Texas) *(FMCAD '11)*. FMCAD Inc, Austin, Texas, 3–8.

[36] SVCOMP. 2025. *Competition on software verification.* https://sv-comp.sosy-lab.org

[37] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2019. Automated mutual induction proof in separation logic. *Form. Asp. Comput.* 31, 2 (April 2019), 207–230. doi:10.1007/s00165-018-0471-5

[38] 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 *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 302–328. doi:10.1007/978-3-031-65630-9_16

[39] 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 406–417. doi:10.1145/3691620.3695014

Just Accepted

[40] 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*. https://openreview.net/forum?id=Q3YaCghZNt

[41] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html

[42] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 106–120. doi:10.1145/3385412.3385986

[43] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 175–187. doi:10.1145/3597926.3598047

[44] Jin Peng Zhou, Charles E Staats, Wenda Li, Christian Szegedy, Kilian Q Weinberger, and Yuhuai Wu. 2024. Don't Trust: Verify – Grounding LLM Quantitative Reasoning with Autoformalization. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=V5tdi14ple

[45] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *SIGPLAN Not.* 53, 4 (June 2018), 707–721. doi:10.1145/3296979.3192416

Just Accepted