LEMUR: INTEGRATING LARGE LANGUAGE MODELS IN AUTOMATED PROGRAM VERIFICATION

Haoze Wu
Department of Computer Science
Stanford University
haozewu@stanford.edu

Clark Barrett
Department of Computer Science
Stanford University
barrett@cs.stanford.edu

Nina Narodytska
VMware Research
n.narodytska@gmail.com

ABSTRACT

The demonstrated code-understanding capability of LLMs raises the question of whether they can be used for automated program verification, a task that often demands high-level abstract reasoning about program properties that is challenging for verification tools. We propose a general methodology to combine the power of LLMs and automated reasoners for automated program verification. We formally describe this methodology as a set of derivation rules and prove its soundness. We instantiate the calculus as a sound automated verification procedure, which led to practical improvements on a set of synthetic and competition benchmarks.

1 INTRODUCTION

AI-powered language models are being routinely used to help developers with their tasks. Examples include program synthesis from natural language descriptions by GPT-4 (OpenAI, 2023) or Github Copilot (Chen et al., 2021; GitHub, 2021), solving competitive programming problems with AlphaCode (Li et al., 2022), and repairing code by DeepRepair (White et al., 2019), among others. These models have shown impressive results in generating correct code in many programming languages.

An important research question is whether modern AI models are capable of understanding the logic behind the programs they analyze. Recently, several approaches have been proposed to combine the strengths of formal verification and Large Language Models (LLMs) that demonstrate such capabilities. For example, Pei et al. (2023) made an important step in this direction by investigating whether LLMs can generate program properties, namely, program invariants, which remains a crucial and challenging task for automated program verification (Clarke et al., 2018). The authors demonstrated that LLMs are effective in generating program invariants on a set of synthetic Java programs. Another example is the recent work by Charalambous et al. (2023), who demonstrated that LLM models can be used to repair vulnerabilities in code, given examples of incorrect behavior. They provided compelling evidence of the complementary strengths of LLMs, which serve as a generator for code repair snippets, and formal techniques, which are used to check the correctness of the generated code. While previous work shows promise in program analysis tasks, they do not provide a formalization of the interaction between LLMs and formal verifiers; require manual efforts, or are limited to the invariant generation process as a stand-alone procedure.

In this work, we propose a novel LLM-powered framework, LEMUR, for automated program verification tasks. Our key insight is to combine LLMs' ability to perform abstract high-level reasoning and automated reasoners' ability to perform precise low-level reasoning. Specifically, LLMs are employed to propose program invariants in the form of sub-goals, which are then checked by automated reasoners. This transforms the program verification tasks into a series of deductive steps suggested by LLMs and subsequently validated by automated reasoners. Our main contributions are:

- a novel framework to combine LLMs and automated reasoners for program verification;

- a presentation of LEMUR as a proof system and a proof of its soundness, which to the best of our knowledge, is the first formalization of such a hybrid approach;
- an instantiation of the LEMUR calculus that gives a sound and terminating algorithm;
- an implementation of the proposed framework (using OpenAI's GPT models) and several optimizations to enhance its practical efficiency;
- an experimental evaluation of LEMUR on two sets of benchmarks that demonstrates its efficiency compared with both existing AI-powered and conventional verification tools.

2 DEFINITIONS

Given a program $\mathcal{P} : \mathrm{Prog}$, a *reachability property*, or simply *property*, is a tuple $p = \langle \phi, l \rangle$, where $\phi : \mathrm{Pred}$ is a Boolean *predicate* of the program state and $l : \mathbb{N}$ is a *program line*. The *negation* of p, denoted $\neg p$, is defined as $\langle \neg\phi, l \rangle$. Next we introduce several useful definitions and their properties.

Definition 2.1. *A property $p = \langle \phi, l \rangle$ is an* invariant *on $\mathcal{P}$, denoted $\mathrm{Inv}(\mathcal{P}, p)$, iff p holds (i.e., ϕ always evaluates to true at line l) for all possible executions of the program $\mathcal{P}$.*

***Example* 2.1.** *Consider a simple program $\mathcal{P}$ on Figure 2 (the first frame, top row). $\mathcal{P}$ instantiates an unsigned 32-bit integer variable x to 0 and increases its value by 4 on each loop iteration. A property $p = \langle \phi, l \rangle$ is specified on the 4th line, so $\phi = (x \mathrel{!}= 30)$ and $l = 4$ for this property. It is easy to see that p is an invariant as x is not divisible by 3, for example.* ■

Next, we introduce a notion of *assumption* on a program $\mathcal{P}$. An assumption $q = \langle \phi, l \rangle$ is a property that is assumed in a program by altering a program behavior.

Definition 2.2. *An* assumption $q = \langle \phi, l \rangle$ *is a property that modifies the program as follows*

1. *if ϕ holds at line l then the program $\mathcal{P}$ continues execution without changes;*
2. *if ϕ does not hold at line l then $\mathcal{P}$ terminates at l.*

We use $\mathcal{P}' = \mathrm{Asm}(\mathcal{P}, q)$ to denote a modification of $\mathcal{P}$ with the assumption q. An assumption can itself be an invariant. We now introduce a special notion of an *implication*.

Definition 2.3. *Let $\mathcal{P}$ be a program, and p, q be properties on $\mathcal{P}$. We say that q* implies *p with respect to $\mathcal{P}$, denoted $q \underset{\mathcal{P}}{\longrightarrow} p$, iff p is an invariant in $\mathrm{Asm}(\mathcal{P}, q)$.*

***Example* 2.2.** *Consider the program $\mathcal{P}$ on Figure 2 and an assumption $q = \langle \phi = (\texttt{x\%4==1}), l = 3 \rangle$ that modifies the original program $\mathcal{P}$, giving $\mathcal{P}' = \mathrm{Asm}(\mathcal{P}, q)$. The first frame in the bottom row of Figure 2 depicts $\mathcal{P}'$. To see a difference between $\mathcal{P}$ and $\mathcal{P}'$, we observe that the loop is executed only once in $\mathcal{P}'$: `x=0` when it enters the loop so `(x%4) !=1`, the ϕ is violated, and $\mathcal{P}'$ terminates.*

If we consider an alternative assumption $q' = \langle \phi = (\texttt{x\%2==0}), l = 3 \rangle$ The second frame at the top depicts $\mathcal{P}'$ for q'. We can see that its predicate ϕ holds for all executions. Hence, q' is an invariant for the original program $\mathcal{P}$. Finally, we can see $q' \underset{\mathcal{P}}{\longrightarrow} p$, where p is from Example 2.1. ■

The following propositions follow from the definitions above.

Proposition 2.1. *Let $\mathcal{P}$ be a program, and p, q be properties on $\mathcal{P}$:*

- *The property p is an invariant on $\mathcal{P}$ if q is an invariant on $\mathcal{P}$ and q implies p with respect to $\mathcal{P}$. More formally, $(\mathrm{Inv}(\mathcal{P}, q) \wedge q \underset{\mathcal{P}}{\longrightarrow} p) \Rightarrow \mathrm{Inv}(\mathcal{P}, p)$.*
- *The property p is not an invariant on $\mathcal{P}$ if the property p is not an invariant on $\mathcal{P}' = \mathrm{Asm}(\mathcal{P}, q)$. More formally, $\neg\mathrm{Inv}(\mathcal{P}, q) \Rightarrow \neg\mathrm{Inv}(\mathcal{P}', p)$.*

Proposition 2.2. *For any property p on a program $\mathcal{P}$, $p \underset{\mathcal{P}}{\longrightarrow} p$.*

Proposition 2.3. *For any properties p, q, r on a program $\mathcal{P}$, $p \underset{\mathcal{P}}{\longrightarrow} q$ and $q \underset{\mathcal{P}}{\longrightarrow} r$, then $p \underset{\mathcal{P}}{\longrightarrow} r$.*

Note that it is possible that neither a property p nor its negation $\neg p$ holds on a program.

***Example* 2.3.** *Consider again our example from Example 2.1 and two properties at line 3: $p = \langle \phi = (\texttt{x\%8==4}), l = 3 \rangle$ and $p' = \langle \phi' = (\texttt{x\%8 !=4}), l = 3 \rangle$. Neither p nor p' is an invariant in $\mathcal{P}$. On the first loop iteration, we have that `x=0` before line 3 so ϕ' holds and ϕ does not at line 3. On the second loop iteration, we have that `x=4` before line 3 so ϕ holds and ϕ' does not.* ■

Definition 2.4. *A property $p = \langle \phi, l \rangle$ is* stable *for $\mathcal{P}$, denoted $\mathcal{S}(\mathcal{P}, p)$, if, for each execution of the program, either ϕ always evaluates to true at line l or ϕ always evaluates to false at line l.*

An invariant must be stable, but a property that is not an invariant might still be stable. For example, any property on a program without loops is stable. If a p is stable, then $\neg p$ is also stable.

Lemma 2.1. *Consider a program $\mathcal{P}$, two properties p, q on $\mathcal{P}$, and a program $\mathcal{P}' = \mathrm{Asm}(\mathcal{P}, q)$. The property p is an invariant on $\mathcal{P}$, if 1) q is stable for $\mathcal{P}$; 2) q implies p with respect to $\mathcal{P}$; and 3) $\neg q$ implies p with respect to $\mathcal{P}$. More formally: $\mathcal{S}(\mathcal{P}, q) \wedge (q \xrightarrow[\mathcal{P}]{} p) \wedge (\neg q \xrightarrow[\mathcal{P}]{} p) \Rightarrow \mathrm{Inv}(\mathcal{P}, p)$.*

Proof. In App. B. □

Assume we have a verifier $\mathcal{V} : \mathrm{Prog} \times \mathbb{P}(\mathrm{Prop}) \times \mathrm{Prop} \mapsto \{\textsc{True}, \textsc{False}, \textsc{Unknown}\}$, which takes as inputs a program $\mathcal{P}$, a set of assumptions $\mathcal{A}$ and a property p, and checks whether $\mathcal{A}$ implies p. More precisely, given set of assumption $\mathcal{A} = \{q_1, \ldots, q_n\}$ we construct a new program $\mathcal{P}' = \mathrm{Asm}(\mathrm{Asm}((\ldots, \mathrm{Asm}(\mathcal{P}, q_1)), q_{n-1}), q_n)$ and the verifier checks if p is an invariant in this program. Hence, a statement that $\mathcal{A}$ implies p on $\mathcal{P}$ means that p is an invariant in $\mathcal{P}'$. $\mathcal{V}$ is *sound*, meaning if $\mathcal{V}$ returns TRUE, then $\mathcal{A}$ implies p, and if $\mathcal{V}$ returns FALSE, then $\mathcal{A}$ does not imply p. Note that $\mathcal{A}$ can be empty, in which case the verifier essentially checks whether p is an invariant in general. When the verifier $\mathcal{V}$ returns TRUE, we say p is proven; and when $\mathcal{V}$ returns FALSE, we say the property is *falsified*. $\mathcal{V}$ is incomplete, meaning that $\mathcal{V}$ can return UNKNOWN.

In practice, $\mathcal{V}$ is instantiated as automated program verifiers such as CBMC (Kroening & Tautschnig, 2014), ESBMC (Gadelha et al., 2018), and UAUTOMIZER (Heizmann et al., 2013). We provide an overview of the main techniques that these tools employ in Section A and note here that a crucial challenge shared across existing verifiers is the automatic decomposition of a verification task into smaller, more manageable sub-tasks. This decomposition requires high-level reasoning that is difficult to automate using conventional formal methods, but plausible to be performed by LLMs, with their documented code-understanding capability. However, we must ensure soundness when LLMs are used to automatically perform this high-level reasoning in program verification tasks.

3 LEMUR: INTEGRATING LLMS IN PROGRAM VERIFICATION

We present a proof system LEMUR that combines LLMs and automated reasoners for proving a property on a program. The calculus operates over a *configuration*, which is either one of the distinguished symbols $\{\textsc{success}, \textsc{fail}\}$ or a tuple $\langle \mathcal{P}, \mathcal{A}, \mathcal{M} \rangle$, where $\mathcal{P}$ is a program, $\mathcal{A}$ is either $\varnothing$ or a singleton representing the assumption, and $\mathcal{M}$ is a list of properties referred to as *proof goals*. $\mathcal{M}$ itself is referred to as a *trail*. The last element of $\mathcal{M}$ represents the current property to prove. The rules describe the conditions under which a certain configuration can transform into another configuration. In this calculus, verifying whether $\mathrm{Inv}(\mathcal{P}, p)$ holds, boils down to finding a sequence of valid rule applications from the *starting configuration* $\langle \mathcal{P}, \varnothing, [p] \rangle$ to either SUCCESS or FAIL.

Our calculus performs oracle calls to LLMs to propose new properties and revise them. The oracle $\mathcal{O}_{\text{propose}}$ proposes new properties for a given program and the current proof goal as inputs. Namely, $\mathcal{O}_{\text{propose}} : \mathrm{Prog} \times \mathrm{Prop} \mapsto \mathbb{P}(\mathrm{Prop})$. An important insight here is that LLMs are capable of generating new properties that are likely to 1) be invariants, and 2) imply the proof goal given a prompt. We will discuss strategies to generate prompts in Section 4. Importantly, properties generated by an LLM are treated as assumptions until we can prove that they are invariants of the original program. The oracle $\mathcal{O}_{\text{repair}}$ revises previously proposed properties. e.g. if we determine that a property q previously produced by $\mathcal{O}_{\text{propose}}$ does not hold or does not imply the current proof goal. In this case, we request an LLM to repair q. We have $\mathcal{O}_{\text{repair}} : \mathrm{Prog} \times \mathrm{Prop} \times \mathrm{Prop} \times \{\textsc{False}, \textsc{Unknown}\} \mapsto \mathbb{P}(\mathrm{Prop})$, whose inputs comprise a program, two properties, and a solver return value. The first property is our current proof goal, and the second property q is usually an assumption previously proposed by oracles. The output of $\mathcal{O}_{\text{repair}}$ is a new set of properties. In practice, we implement it with a prompt to an LLM to either correct or strengthen q (see Section 4). Finally, the calculus performs an external call to a verifier $\mathcal{V}$ to check whether a property holds.

The proof rules of LEMUR are shown in Fig. 1. Each rule defines a set of preconditions that specify the configurations where it can be applied. Note again that our preconditions permit invocations of LLMs and/or verifiers. The rules within the calculus can be partitioned into four groups.

$$\frac{\mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q\}, \mathcal{M}} \textbf{(Propose)}$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \varnothing, \mathcal{M} :: q} \textbf{(Decide)}$$

$$\frac{\mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) \neq \textsc{True} \quad q' \in \mathcal{O}_{\text{propose}}(\mathcal{P}, p)}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \textbf{(Backtrack)}$$

$$\frac{\mathcal{A} = \{q\} \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{Unknown} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{Unknown})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \textbf{(Repair 1)}$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, q) = \textsc{False} \quad q' \in \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \textsc{False})}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \mathcal{P}, \{q'\}, \mathcal{M}' :: p} \textbf{(Repair 2)}$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \textbf{(Success 1)}$$

$$\frac{\mathcal{A} = \varnothing \quad \mathcal{M} = \mathcal{M}' :: p :: q \quad \mathcal{S}(\mathcal{P}, q) \quad \mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \textsc{True}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{success}} \textbf{(Success 2)}$$

$$\frac{\mathcal{M} = [p] \quad \mathcal{V}(\mathcal{P}, \mathcal{A}, p) = \textsc{False}}{\mathcal{P}, \mathcal{A}, \mathcal{M} \Longrightarrow \textsc{fail}} \textbf{(Fail)}$$

Figure 1: Deductive rules of the LEMUR calculus.

The first group contains rules that are responsible for generating new proof goals given specific configurations. These rules are **Propose**, **Repair 1**, and **Repair 2**. The **Propose** rule states that if the verifier is unable to prove or disprove the current proof goal p, we could invoke the oracle $\mathcal{O}_{\text{propose}}$ to obtain a property q, and update $\mathcal{A}$ to be $\{q\}$. It is also possible to modify the proposed property produced by $\mathcal{O}_{\text{propose}}$. The **Repair 1** rule can be applied when the current assumption q is not sufficient for the verifier to prove the current proof goal p. In this case, we could use the oracle $\mathcal{O}_{\text{repair}}$ to propose ways to strengthen q and choose one of them, q', as the new assumption. On the other hand, the **Repair 2** rule can be applied when q is already in the trail but is falsified by the verifier $\mathcal{V}$. In this case, we could use $\mathcal{O}_{\text{repair}}$ to repair q and update $\mathcal{A}$ accordingly.

The second group specifies how LEMUR makes progress in the proof. The **Decide** rule specifies that the condition under which the assumption q can be made the new proof goal (i.e., being appended to $\mathcal{M}$)—when the verifier $\mathcal{V}$ is able to prove that q implies the current proof goal.

The third group defines how LEMUR can recover from faulty assumptions. In particular, the **Backtrack** rule allows us to revert to the previous proof goal (the second to the last property in the trail $\mathcal{M}$) and pick a different assumption suggested by $\mathcal{O}_{\text{propose}}$, if there are at least two elements in the trail and the verifier cannot prove the current proof goal. Note that **Backtrack** might not be the only applicable rule in this case. For example, **Repair 1** is also applicable. In practice, we need a strategy to decide between multiple applicable rules. This discussion is deferred to Sec. 4.

The final group specifies three termination conditions that can be either SUCCESS or FAIL. The **Success 1** rule states that whenever the assumption is empty and the verifier is able to prove the current proof goal (i.e., the last property p in the trail $\mathcal{M}$), we can transition into the SUCCESS state. If the verifier can directly prove the original property, then the rule can be directly applied to the starting configuration to reach SUCCESS. Otherwise, p would come from the oracles and is different from the original property. **Success 2** states that if the last two elements of the trail $\mathcal{M}$ are p, and q, the current proof goal $q := \langle \phi, l \rangle$ is stable (as defined in Sec. 2), and the verifier is also able to also prove p under the assumption of $\langle \neg\phi, l \rangle$, then p is an invariant and we can transition to SUCCESS. The **Success 2** rule constitutes a way to utilize an *incorrect sub-goal* q proposed by the LLM-based oracles to decompose the verification task: we separately reason about the cases when q holds and when it does not hold. Finally, if the verifier $\mathcal{V}$ proves that the original property is not an invariant, whether under an assumption or not, then we transition to the FAIL state using **Fail**.

Note that the program $\mathcal{P}$ remains unchanged throughout the transitions. We keep it as part of the state for two reasons. First, $\mathcal{P}$ is an input to the verifiers and the oracles. Second, in the future, it

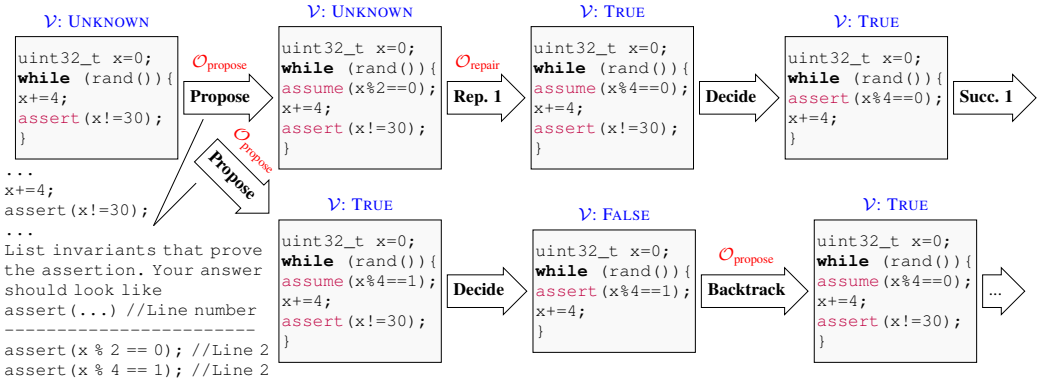

Figure 2: A running example of executing the LEMUR calculus.

might be possible to augment the proof system to update $\mathcal{P}$, by, for example, rewriting the program using LLMs in an invariant-preserving manner.

We state the following soundness properties about LEMUR. The proof is presented in App. C.1.

Theorem 3.1 (Soundness)**.** *Given a property p and a program $\mathcal{P}$, if* SUCCESS *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is an invariant on $\mathcal{P}$.*

Theorem 3.2 (Soundness 2)**.** *Given a property p and a program $\mathcal{P}$, if* FAIL *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is not an invariant on $\mathcal{P}$.*

***Example* 3.1.** *To provide more intuition about the proof system and to motivate the design choices when instantiating* LEMUR*, we consider again our running example. Figure 2 illustrates how* LEMUR *can be used to verify properties in practice. In Figure 2 each frame represents a state of the program. Transitions between states are depicted by arrows, with each arrow marked with the rule applied to execute this transition. In this example, our goal is to prove the property* `x!=30` *in a while loop that keeps adding 4 to an unsigned 32-bit integer variable* `x`*. We note that this particular verification task is adapted from a similar one in the SV-COMP competition.*[1] *While seemingly trivial, during the competition, 19 out of the 24 participating tools (including the overall winner of the competition* UAUTOMIZER*) were not able to solve this benchmark.*

Given such a verification problem, we create an initial configuration $\langle \mathcal{P}, \varnothing, [p] \rangle$ where $\mathcal{P}$ is the given problem and $p = \langle$ `x!=30`*$, 3 \rangle$.*[2] *Suppose the verifier $\mathcal{V}$ is unable to solve this problem and returns* UNKNOWN*. In this case, we need to generate a new proof goal, so the only rule we could apply is* ***Propose****. To do so, we invoke the LLM-based oracle $\mathcal{O}_{propose}$ to obtain a set of new properties that are potentially themselves invariants and might help prove the property. An example prompt is given on the left bottom part. This is not the exact prompt that we found the most effective in practice and we defer the discussion of* prompting strategies *to Sec. 4. Suppose the oracle returns two potential predicates at the beginning of the while loop:* `x%2==0` *and* `x%4==1` *at line 3. The* ***Propose*** *allows us to make one of them the current assumption.*

Case (`x%2==0`): *The top row illustrates what happens when we transition into $\langle \mathcal{P}, \{q = \langle$* `x%2==0`*$, 2 \rangle\}, [p] \rangle$. While q is indeed an invariant, it does not help to prove the assertion and $\mathcal{V}$ would return* UNKNOWN*. This satisfies the condition to apply the* ***Repair 1*** *rule, which would invoke the oracle $\mathcal{O}_{repair}$ to strengthen q. Suppose in this case, the oracle suggests the predicate $q' =$* `x%4==0`*, which clearly implies the original property* `x!=30`*. Suppose then $\mathcal{V}(\mathcal{P}, \{q'\}, p)$ returns* TRUE*. We could apply the* ***Decide*** *rule and transition to $\langle \mathcal{P}, \varnothing, [p, q'] \rangle$, making q' the current proof goal. Proving q' is arguably easier because* `x%4==0` *is* inductive *(i.e., if it holds in one iteration and then it will hold in the next iteration), making conventional automated reasoning techniques such as k-induction applicable. Suppose $\mathcal{V}(\mathcal{P}, \varnothing, q') =$* TRUE*, we could apply* ***Success 1*** *and transition into the* SUCCESS *state, thus completing the proof.*

[1] https://sv-comp.sosy-lab.org/2023/results/results-verified/META_ReachSafety.table.html#/table?filter=id_any(value(jain_5-2))

[2] 3 is the line number (in the snippet) where the predicate is asserted.

Case (`x%4==1`): *The bottom row illustrates a different chain of rule applications when we picked the property* $r = \langle$`x%4==1`$, 2\rangle$ *from the first proposal. While* r *does not hold, it does imply* `x!=30`*. Suppose this implication is proven by the verifier. We could apply* ***Decide*** *and transition to* $\langle \mathcal{P}, \varnothing, [p, r] \rangle$*. Since* r *is not an invariant,* $\mathcal{V}(\mathcal{P}, \varnothing, r)$ *would be either* UNKNOWN *or* FALSE*. Either way, we could apply* ***Backtrack*** *and try a new assumption proposed by* $\mathcal{O}_{propose}$*. In practice, we could either invoke the stochastic* $\mathcal{O}_{propose}$ *again or pick an un-attempted property (e.g.,* $\langle$`x%2==0`$, 2\rangle$ *proposed previously). In the illustration, we invoke* $\mathcal{O}_{propose}$ *again and obtain the "correct" predicate* `x%4==0`*, which would allow us to prove the property in two more steps.* ■

4 INSTANTIATING THE LEMUR CALCULUS

In this section, we present strategies to instantiate LEMUR as an automated decision procedure. While we showed that LEMUR calculus is a sound procedure, there are no guarantees that it terminates. Here, we will discuss two sources of non-termination in this calculus.

The first one corresponds to unbounded suggestions of new sub-goals to prove the current proof goal. Concretely, when trying to prove a particular proof goal p, we could get stuck if $\mathcal{V}(\mathcal{P}, \{q\}, p) =$ UNKNOWN or $\mathcal{V}(\mathcal{P}, \varnothing, q) =$ FALSE for each proposed assumption q. This could be due to limitations in either the LLM or the verifier. One way to avoid this type of non-termination is by putting an upper bound on the number of proposed assumptions to prove each proof goal. That is, for any proof goal p, we require that $\mathcal{V}(\mathcal{P}, \{q\}, p)$ is invoked for at most **k** different q's.

The second source of non-termination corresponds to an unbounded depth of the trail $\mathcal{M}$. Concretely, it is possible to construct an infinite sequence of **Propose** and **Decide** where 1) the verifier returns UNKNOWN on the current proof goal; 2) the oracle proposes an assumption that is not invariant but implies the current proof goal; 3) the verifier proves the implication; 4) the assumption becomes the new proof goal; and 5) repeat. This case can be avoided by adding a side condition to the rules that the property proposed by the oracles ($q = \langle \psi, l' \rangle$) must be at a smaller program line than the current proof goal ($p = \langle \phi, l \rangle$), that is,

$$\langle \psi, l' \rangle \in \mathcal{O}_*(\mathcal{P}, \langle \phi, l \rangle, \ldots) \Rightarrow l' < l \qquad \text{(Condition 1)}$$

Based on the strategy described above, a terminating (by Thm. 4.1 at the end of this section) and sound (by Thm. 3.1) algorithm for checking whether a property p is an invariant on a program $\mathcal{P}$ is presented in Alg. 1. Alg. 1 is a recursive procedure lemur_check. It takes a program $\mathcal{P}$ and a property p as inputs. If lemur_check returns SUCCESS, then the property is an invariant. If lemur_check returns FAIL, then the property is not an invariant. The function can also return UNKNOWN, if the analysis is inconclusive. At the high level, Alg. 1 searches for a potential subgoal q that implies the current goal p (lines 9–21). If such q is identified in line 13, we recurse to prove q (line 16). The `while` loop starting at line 10 ensures that at most k attempts can be utilized to generate a new subgoal for p. See a full description of Alg 1 in Appendix D. The comments in Alg. 1 show which rule is applied at the lines. The algorithm is sound as it only applies the rules of the calculus. We prove that the algorithm terminates in Appendix D.

Theorem 4.1 (Termination)**.** *Given a program* $\mathcal{P}$*, and a property* p *on the program, Alg. 1 terminates with either* SUCCESS*,* FAIL*, or* UNKNOWN*.*

5 EXPERIMENTS

We have presented the LEMUR calculus and described a sound and terminating algorithm based on LEMUR. In this section, we investigate the following question:

- Can we develop a practical automated verification procedure based on Alg 1? [Yes]
- Is LEMUR competitive with existing end-to-end learning-based verification approaches? [Yes]
- Can LEMUR already prove hard benchmarks that are beyond the reach of state-of-the-art conventional program verifiers? [In several cases]

Algorithm 1 The LEMUR procedure

Input: A program $\mathcal{P}$, a property p.
Output: SUCCESS only if $\text{Inv}(\mathcal{P}, p)$; FAIL only if $\neg\text{Inv}(\mathcal{P}, p)$; and UNKNOWN if inconclusive.
Parameters: Verifier $\mathcal{V}$, oracles $\mathcal{O}_{\text{propose}}$ and $\mathcal{O}_{\text{repair}}$ (which satisfy Condition 1), number of proposals $\mathbf{k}$
function lemur_check($\mathcal{P}, p$)
    $d \mapsto \mathcal{V}(\mathcal{P}, \varnothing, p)$
    if $d = \text{FALSE}$ **then return** FAIL ▷ **Fail**
    else if $d = \text{TRUE}$ **then return** SUCCESS ▷ **Success 1**
    else
        $i, Q \mapsto 0, \mathcal{O}_{\text{propose}}(\mathcal{P}, p)$
        while $i < \mathbf{k} \wedge |Q| > 0$ **do**
            $i \mapsto i + 1$
            $q \mapsto \text{pop}(Q)$
            $e \mapsto \mathcal{V}(\mathcal{P}, \{q\}, p)$ ▷ **Propose/Backtrack**
            if $e = \text{FALSE}$ **then return** FAIL ▷ **Fail**
            else if $e = \text{TRUE}$ **then**
                $f \mapsto \text{lemur\_check}(\mathcal{P}, q)$ ▷ **Decide**
                if $f = \text{SUCCESS}$ **then return** SUCCESS ▷ **Success 1**
                else if $\mathcal{S}(\mathcal{P}, q) \wedge (\mathcal{V}(\mathcal{P}, \{\neg q\}, p) = \text{TRUE})$ **then return** SUCCESS ▷ **Success 2**
                else if $f = \text{FAIL}$ **then** $Q \mapsto \text{join}(Q, \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{FALSE}))$ ▷ **Repair 2**
                else continue
            else $Q \mapsto \text{join}(Q, \mathcal{O}_{\text{repair}}(\mathcal{P}, p, q, \text{UNKNOWN}))$ ▷ **Repair 1**
        return UNKNOWN

5.1 BUILDING AN LLM-BASED PROGRAM VERIFIER

We report the practical considerations when building a prototype of Alg. 1. There are two types of external calls that Alg. 1 depends on. The first type is calls to $\mathcal{V}$. We use off-the-shelf verifiers in our framework that are extensively tested by the community (described in later paragraphs), so we have some expectations about their performance. However, the main source of uncertainty in building LEMUR comes from interaction with the second type of calls, calls to LLM oracles, as we treat them as black boxes. In our framework, the oracles $\mathcal{O}_{\text{propose}}$ and $\mathcal{O}_{\text{repair}}$ automatically prompt a GPT-family model through the OpenAI API and parse its outputs. We use GPT-4 by default. We found that while GPT has great potential in generating sensible loop invariants, it still has practical limitations. We report several tactics that we found useful in practice.

- **Formatting the output**: We initially investigated whether the popular chain-of-thought (CoT) reasoning (Wei et al., 2022) can be useful to discover new properties given $\mathcal{P}$ and p. We found that GPT's outputs were verbose and often contained irrelevant/incorrect statements, even in cases where useful invariants were contained in the outputs. This behavior increases the cost and makes it difficult to extract invariants from the output. To address these issues, we used in-context learning to format the output of GPT. For example, adding `Your output should be "assert(...);// Line number"` to the prompt is sufficient for GPT to consistently generate outputs of exactly this format, without providing verbose explanations.

- **Inserting markers to the program**: We found that the current versions of GPT are not good at counting program lines. In many cases, the predicate generated by GPT is "correct" but the line number is off by a small margin. This is highly undesirable as an invariant at a wrong position is of no use to the verifier. To mitigate this challenge, we inserted placeholder lines of the form `"// Line A"`, `"// Line B"` to the program and prompted GPT to generate invariants of the form `assert(...);// Line name` (for those specific locations). As a simple practical heuristics, we insert placeholders to right before the loop and the beginning of the loop.

- **Ordering the proposal**: The output of an oracle call is non-deterministic for a given prompt, depending on the hyper-parameters of the call. Moreover, the oracles produce a set of properties and we need good heuristics to choose the order of trying those properties. A heuristic we found practically useful is to prompt GPT multiple times and order the properties by the frequency they are proposed (breaking ties by preferring shorter expressions). Moreover, instead of relying on string matching, we treat two proposals the same if their abstract syntax trees are equivalent.

Configurations	Solved	Time	# proposal
Code2Inv	92	–	> 20
ESBMC	68	0.34	0
LEMUR	107	24.9	4.7

(a) The Code2Inv benchmarks.

Configurations	Solved	Time	# proposals
UAUTOMIZER	0	–	0
ESBMC	0	–	0
LEMUR	26	140.7	9.1

(b) The 50 SV-COMP benchmarks.

Table 1: Solved instances by ESBMC, LEMUR, and Code2Inv (1a) or UAUTOMIZER (1b) on two benchmark sets. We also report the average time and number of proposals on solved instances.

The exact prompts are described in Appendix F. We consider two state-of-the-art C program formal analyzers for $\mathcal{V}$, ESBMC (Gadelha et al., 2018) and UAUTOMIZER (Heizmann et al., 2013). The former is based on K-induction and the latter is based on predicate-abstraction. In particular, ESBMC and UAUTOMIZER are the top two performing non-portfolio solvers in the reachability track of the SV-COMP (Beyer, 2023). And UAUTOMIZER is the overall winner of the competition. By default, we impose a 30-second time limit for each invocation of the verifier. That is, if the verifier does not terminate within 30 seconds, then the verifier returns UNKNOWN. The total cost incurred by using the OpenAI API services for the experiments (including testing phases) is $1000. We will release the source code and the benchmarks for the community to make further improvements.

5.2 LOOP INVARIANT GENERATION BENCHMARKS

A prominent approach in learning-based end-to-end program verification is Code2Inv, which uses reinforcement learning to train an invariant synthesizer to propose loop invariants. In this section, we study how LEMUR compares with this approach. The Code2Inv (Si et al., 2020) benchmark set contains 133 benchmarks, each containing a C program and a property expressed as an assert statement in the program. Each program contains a single loop and each loop can have nested if-then-else blocks (without nested loops). Programs in the benchmark may also have uninterpreted functions (emulating external function calls) in branches or loop termination conditions. The assertion to check is always after the loop. As reported in the original Code2Inv paper, these problems can be efficiently solved using state-of-the-art invariant synthesis solvers and the goal was to evaluate the ability of Code2Inv to generate a real invariant that implies the property at the beginning of the loop. To have a fair comparison, we prompt the oracles to generate invariants in the same location as Code2Inv in Alg. 1.

We use the k-induction-based verifier, ESBMC, to check the implication (line 13 in Alg. 1) which aligns with the verification procedure used in Code2Inv. We report the number of solved instances as well as the number of failed suggestions (either itself cannot be verified or ESBMC times out on the implication check). As a point of comparison, we report those statistics from the original Code2Inv approach, which combines graph and recurrent neural networks to model the program graph and learn from counterexamples. Code2Inv was given a one-hour timeout. In addition, we also report ESBMC's performance on this set of benchmarks. The result is shown in Table 1a.

With a 10-minute timeout, ESBMC alone can solve 68 problems. On the other hand, LEMUR can solve 107 problems within the same time limit. Surprisingly, this approach solves more instances than Code2Inv, which is specifically designed for invariant synthesis tasks. Moreover, LEMUR is able to find the correct loop invariant with on average 4.7 attempts, while it takes Code2Inv on average > 20 attempts to do so. For problems unsolved by ESBMC but solved by LEMUR, a histogram of the values of Log_2 of the number of proposals is shown in Fig. 3. While in most cases, Alg. 1 can propose the correct proposals within 4 attempts, there are still benchmarks that take LEMUR many rounds of proposal and repair to find the desired loop invariant, e.g. one of the benchmarks took 177 proposals.

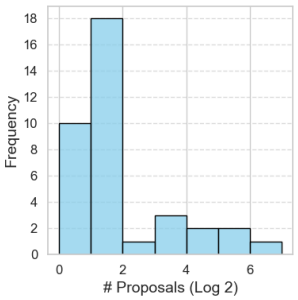

Figure 3: Number of proposals for LEMUR to solve a benchmark.

5.3 SOLVING HARD SV-COMP BENCHMARKS

Next, we study LEMUR's ability to solve hard benchmarks from the Software-Verification Competition 2023 (Beyer, 2023). Due to budget limitations, we focus on benchmarks with less than 150 tokens (after removing comments, unnecessary headers, and clang-formatting). We select 50 benchmarks that ESBMC and UAUTOMIZER are unable to solve within 20 minutes and evaluate LEMUR on them with the same timeout. The property is expected to hold in all benchmarks. These benchmarks are considerably more challenging than the Code2Inv programs. While the latter has one loop and follows a strict format, the SV-COMP benchmark presents a more diverse set of benchmarks, with multiple loops present in many programs.

The results are shown in Table 1b. Impressively, with the guidance of the proof goals suggested by the LLM, LEMUR is able to solve 26 of the 50 SV-COMP benchmarks. While neither ESBMC nor UAUTOMIZER can solve a single benchmark alone. Upon closer examination, 6 of the solved instances contain 2 loops, 3 contain 3 loops, and 3 contain 4 or more loops. **To our knowledge, this is the first time a learning-based verification approach 1) can handle programs with more than one loop; and 2) boosts the performance of state-of-the-art conventional C program verifiers.**

The average number of proposals before solving a problem is higher compared to the Code2Inv benchmarks (9.1 vs. 4.7). Fig. 4 sheds more light on the behavior of LEMUR. In particular, 16 of the 26 solved instances require more than 6 proposals in total.

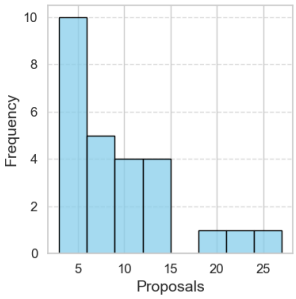

Figure 4: Number of proposals for LEMUR to solve a benchmark.

We found that the LLM-based oracles can produce surprisingly insightful loop invariants that are difficult for conventional formal methods to synthesize. While predicate-abstraction-based techniques typically generate predicates that involve only the operators and values in the program and follow a particular template, LLM is not constrained by these limitations. For example, for the program in Fig. 2, GPT-4 can consistently generate `x%4==0` as the loop invariant although the modulo operator is not present in the program. Appendix. F.1 shows an example where LLM understands the range of `unsigned char` as the key to proving the given property and suggests variable bounds as the assumption. There are also several cases where the LLM generates disjunctive invariants that precisely characterize the behavior of the loops.

6 DISCUSSION OF LIMITATIONS AND EXTENSIONS

In this work, we proposed a novel framework, LEMUR, that combines automated reasoning and LLMs. To the best of our knowledge, LEMUR is the first framework that provides a theoretical foundation for such integration, i.e., a formal calculus, and practical algorithmic instantiation of the calculus. We also implemented LEMUR as a fully automated framework and demonstrated its efficiency on standard benchmark sets. We conclude by discussing the current limitations of LEMUR, which point to future research directions to extend the framework.

As we mentioned above, the practical performance of LEMUR depends on two types of external calls: the verifiers and the LLMs. Any improvements in these tools should translate into LEMUR improvements. Currently, modern verifiers are capable of handling relatively small programs (see SV-COMP'23 (Beyer, 2023)). Interestingly, even when provided with a strong invariant, they sometimes cannot solve the verification problem. One research direction that we envision is to customize LEMUR to a particular backend verifier to obtain better performance and solve larger programs.

While our experience with LLMs was largely positive (see Section 5.1 for a discussion on the limitations we have successfully overcome), there are more interesting challenges to tackle. First, LLMs can take a limited number of tokens as inputs, and many practical programs exceed that limit. Second, it is sometimes challenging for LLMs to generate more complex logical formula such as nested if-then-else properties. We believe that to overcome this limitation we need to 1) develop a prompting language for invariant generation with LLMs, and 2) fine-tune LLMs for invariant generation tasks using this language.

Finally, due to the limitations of LLMs and automated reasoners, our hybrid framework is not yet able to offer a significant leap in automatically verifying complex properties on real-world C libraries. However, a modular approach, where large parts of the program are abstracted and summarized in the form of pre- and post-conditions, can benefit from frameworks like LEMUR.

REFERENCES

Dirk Beyer. Competition on software verification and witness validation: Sv-comp 2023. In Sriram Sankaranarayanan and Natasha Sharygina (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 495–522, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C. Cordeiro. A new era in software security: Towards self-healing software via large language models and formal verification, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423. URL https://doi.org/10.18653/v1/n19-1423.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. *CoRR*, abs/2303.04910, 2023. doi: 10.48550/arXiv.2303.04910. URL https://doi.org/10.48550/arXiv.2303.04910.

Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 888–891, 2018.

Inc. GitHub. GitHub Copilot. https://copilot.github.com/, 2021. Accessed: September 2023.

Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 641–643. Springer, 2013.

Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, pp. 389–391. Springer, 2014.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In *NeurIPS*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/18abbeef8cfe9203fdf9053c9c4fe191-Abstract-Conference.html.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, dec 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126%2Fscience.abq1158.

OpenAI. Gpt-4 technical report, 2023.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 27496–27520. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/pei23a.html.

Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2inv: A deep learning framework for program verification. In Shuvendu K. Lahiri and Chao Wang (eds.), *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pp. 151–164. Springer, 2020. doi: 10.1007/978-3-030-53291-8_9. URL https://doi.org/10.1007/978-3-030-53291-8_9.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, feb 2019. doi: 10.1109/saner.2019.8668043. URL https://doi.org/10.1109%2Fsaner.2019.8668043.

A BACKGROUND

Automated reasoning tools. We overview several popular techniques that are used by modern program verification solvers, like CBMC (Kroening & Tautschnig, 2014), ESBMC (Gadelha et al., 2018), and UAUTOMIZER (Heizmann et al., 2013).

The Bounded Model Checking (BMC) approach is an iterative technique that verifies the program for each unwind bound up to a maximum value, m, e.g. it can perform m loop unwinding. It either finds a counterexample within m steps or returns UNKNOWN. This approach usually employs SMT solvers to find counterexamples very efficiently at each step. However, for non-trivial systems unrolling can be expensive and memory-consuming. Moreover, vanilla BMC can only check finite reachability, e.g. it cannot prove loop invariants, for example.

Another popular technique is the k-induction method, which allows BMC-style methods to prove properties like loop invariants. This approach is also iterative. First, we check whether a property holds for k steps from a valid state. If it does not, we find a counterexample. Otherwise, we check an inductive hypothesis that if the property holds for k steps from a state then the property holds for the $k+1$th step. If it does, the property holds for any number of steps. If not, k-induction either increases k or returns UNKNOWN. As in the case of BMC, unrolling can be computationally expensive. Moreover, k-induction is not complete; there are properties that are not k-inductive for any k.

The last approach we consider is abstract interpretation verification methods. Such methods create abstract representations of program states and variables. These abstract representations are simplified versions of the actual program states, focusing on relevant aspects while ignoring irrelevant details. The choice of abstraction depends on the specific properties to verify. Moreover, if a property holds for an abstract program, then it holds for the original program. The reverse is not true. Hence, if a property does not hold for an abstract program, we need to refine the abstract representation to exclude the counterexample. The main challenge here is how to come up with a good abstraction and how to design a refinement procedure.

Large Language Models. Large Language Models belong to a class of artificial intelligence models used in natural language processing tasks. These models are designed to process and generate both human language and structured inputs, such as code in various programming languages. Examples of large language models include Generative Pre-trained Transformer models like GPT-3 (Brown et al., 2020) or GPT-4 (OpenAI, 2023), Bidirectional Encoder Representations from Transformers, BERT (Devlin et al., 2019), and others. LLMs are getting increasingly popular as an AI-assistance for code generation tasks, like PaLM (Chowdhery et al., 2022), GitHub Copilot (Chen et al., 2021; GitHub, 2021), etc.

LLMs are usually trained in two steps. The main phase is training, where these models are exposed to very large corpora of data, usually collected over the internet. The architecture of LLMs is based on transformers and has a very large number of parameters. Therefore, it can capture relations between different parts of the input text to produce coherent outputs. For some applications, we need to perform fine-tuning to expose the model to application-specific inputs. During inference, when a user provides inputs and a prompt that contains instructions to an LLM, it generates the output with respect to these instructions.

B DEFINITIONS

Lemma 2.1. *Consider a program $\mathcal{P}$, two properties p, q on $\mathcal{P}$, and a program $\mathcal{P}' = \mathrm{Asm}(\mathcal{P}, q)$. The property p is an invariant on $\mathcal{P}$, if 1) q is stable for $\mathcal{P}$; 2) q implies p with respect to $\mathcal{P}$; and 3) $\neg q$ implies p with respect to $\mathcal{P}$. More formally: $\mathcal{S}(\mathcal{P}, q) \wedge (q \underset{\mathcal{P}}{\longrightarrow} p) \wedge (\neg q \underset{\mathcal{P}}{\longrightarrow} p) \Rightarrow \mathrm{Inv}(\mathcal{P}, p)$.*

Proof. Let $q = \langle \phi, l \rangle$. By the definition of stability, for any execution of $\mathcal{P}$, either ϕ always evaluates to true at line l or $\neg\phi$ always evaluates to true at line l. In either case, the property p holds by the definition of implication. Therefore, p holds for all executions of $\mathcal{P}$, i.e., $\mathrm{Inv}(\mathcal{P}, p)$. □

C LEMUR: INTEGRATING LLMS IN PROGRAM VERIFICATION

C.1 SOUNDNESS OF LEMUR

Lemma C.1. *For any configuration $\langle \mathcal{P}, \mathcal{A}, \mathcal{M} \rangle$ created by a sequence of valid rule applications starting from an initial configuration $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, $\mathcal{M}$ is not empty.*

Proof. This can be proven by induction on the length of the sequence. In the base case, $\mathcal{M}$ is $[p_0]$. In the inductive case, the length of $\mathcal{M}$ does not reduce except in the **Backtrack** rule which requires $\mathcal{M}$ to have at least 2 elements in the pre-condition. Thus, $\mathcal{M}$ is not empty. □

Lemma C.2. *Let $\langle \mathcal{P}, \mathcal{A}, \mathcal{M} \rangle$ be a configuration created by a sequence of valid rule applications starting from an initial configuration $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, and let p be the last element of $\mathcal{M}$. We have $p \underset{\mathcal{P}}{\longrightarrow} p_0$.*

Proof. We prove a stronger property, that for *each* element p in $\mathcal{M}$, $p \underset{\mathcal{P}}{\longrightarrow} p_0$. We induct on the length of the sequence. In the base case, $p_0 \underset{\mathcal{P}}{\longrightarrow} p_0$ by proposition 2.2. In the inductive case, we proceed by cases. **Success 1**, **Success 2**, **Fail** cannot be applied. In the post conditions of **Propose**, **Backtrack**, **Repair 1**, and **Repair 2**, $\mathcal{M}$ either shrinks or remains the same. Therefore, the inductive hypothesis can be directly applied. If **Decide** rule is to be applied. In the pre-condition, the trail is $\mathcal{M} :: p$, the current assumption is $\{q\}$ and $q \underset{\mathcal{P}}{\longrightarrow} p$. In the post condition, $\mathcal{M}$ becomes $\mathcal{M} :: p :: q$. By the inductive hypothesis, $p \underset{\mathcal{P}}{\longrightarrow} p_0$. Furthermore, by Proposition 2.3, $q \underset{\mathcal{P}}{\longrightarrow} p_0$. □

Lemma C.3. *Let $\langle \mathcal{P}, \mathcal{A}, \mathcal{M} :: p :: p' \rangle$ be a configuration created by a sequence of valid rule applications starting from an initial configuration $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, we have $p' \underset{\mathcal{P}}{\longrightarrow} p$.*

Proof. This can be proven by induction on the length of the sequence. □

Theorem 3.1. *Given a property p and a program $\mathcal{P}$, if* SUCCESS *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is an invariant on $\mathcal{P}$.*

Proof. We can transition into SUCCESS by either the **Success 1** rule or the **Success 2** rule. In the pre-condition of **Success 1**, the trail is of the form $\mathcal{M} :: p$, and the verifier $\mathcal{V}$ proves that $\mathrm{Inv}(\mathcal{P}, p)$. By Lemma C.2, $p \underset{\mathcal{P}}{\longrightarrow} p_0$. Further by Proposition 2.1, we have $\mathrm{Inv}(\mathcal{P}, p_0)$. On the other hand, in the pre-condition of **Success 2**, the trail is of the form $\mathcal{M} :: p :: p'$. By Lemma C.3, $p' \underset{\mathcal{P}}{\longrightarrow} p$. In addition, p' is stable and $\neg p' \underset{\mathcal{P}}{\longrightarrow} p$. Therefore, by Lemma 2.1, p is an invariant of $\mathcal{P}$. Since we also know from Lemma C.2 that $p \underset{\mathcal{P}}{\longrightarrow} p_0$, it follows from Proposition 2.1 that p_0 is an invariant of $\mathcal{P}$. □

Theorem 3.2. *Given a property p and a program $\mathcal{P}$, if* FAIL *is reached by a sequence of valid rule applications starting from $\langle \mathcal{P}, \varnothing, [p_0] \rangle$, then p_0 is not an invariant on $\mathcal{P}$.*

Proof. We transition into the FAIL state only when the verifier $\mathcal{V}(\mathcal{P}, \mathcal{A}, p_0) = \text{FALSE}$. Even if $\mathcal{A}$ is not empty, p_0 is still not an invariant by Prop. 2.1. □

D INSTANTIATING THE LEMUR CALCULUS

Here, we describe Alg. 1. First, the algorithm checks whether the current p can be verified by $\mathcal{V}$ or if a counterexample exists (line 5). If so, it returns either SUCCESS or FAIL to the upper level of recursion or terminates if lemur_check is at the top level. If $\mathcal{V}$ cannot prove p, i.e. it returns UNKNOWN, lemur_check enters a new phase of subgoal generation, where LLM oracles are used to propose new or repair existing properties (lines 9–21). In this phase, we start by calling $\mathcal{O}_{\text{propose}}$ to generate a new subgoal (line 9). The `while` loop at line 10 ensures that at most k attempts can be unitized to generate a new subgoal for p. In line 13, we call $\mathcal{V}$ to check whether q implies

p. If $\mathcal{V}$ returns FALSE, we know that p is not an invariant and return FAIL (line 14). If $\mathcal{V}$ returns UNKNOWN, then we need to repair q; for example, we might strengthen q and try again to prove implication. Otherwise, if q *does* imply p, we recurse to prove q (line 16). The last logical block of lemur_check in lines 17–20 addresses the output of the recursive call. If we have successfully proved that q is an invariant, we return SUCCESS. Otherwise, if q is stable (see Definition 2.4), we can check whether $\neg q$ implies p (line 18). If so, by Lemma 2.1, we can conclude that p is an invariant and also return SUCCESS. If we prove that q is FALSE, we can repair q by informing an LLM oracle that the property does not hold (line 19). Finally, if f is UNKNOWN then we continue to the next iteration of the `while` loop and consider the next proposed sub-goal.

Second, we present a proof of Theorem 4.1.

Theorem 4.1. *Given a program $\mathcal{P}$, and a property p on the program, Alg. 1 terminates with either* SUCCESS, FAIL, *or* UNKNOWN.

Proof. Suppose $p = \langle \phi, l \rangle$. We prove with a decreasing argument on l. When $l = 0$, the algorithm terminates without entering the while loop, because $\mathcal{O}_{\text{propose}}$ satisfies Condition 1 and $\mathcal{O}_{\text{propose}}(\mathcal{P}, p) = \varnothing$. In the recursive case, the while loop is executed for at most $\mathbf{k}$ iterations. In each iteration, we show that for the second input to lemur_check (Line 16), $q = \langle \psi, l' \rangle$, we have $l' < l$. This is true because q is generated either by $\mathcal{O}_{\text{propose}}(\mathcal{P}, p)$ or $\mathcal{O}_{\text{repair}}(\mathcal{P}, p, \ldots)$, both satisfying Condition 1. □

E RELATED WORK

There has been a lot of interest in using LLMs to augment formal reasoning. Charalambous et al. (2023) proposed a novel framework, ESBMC-AI, that integrated LLMs reasoning and formal verification. They also applied their framework to the analysis of C programs focusing on memory safety properties. The main idea is to use LLMs as a code repair generator that can fix faulty code using a carefully designed prompt, a program, and a counterexample provided by a bounded model checker. However, ESBMC-AI assumes that program rewriting done by an LLM is valid, i.e. syntactically and semantically correct. The latter is challenging to prove in an automatic manner as it requires program equivalence checking. Our framework does not use LLMs to modify code and treat the outputs of the LLM as suggestions until we prove that they are correct. Another example of an automated framework is Baldur (First et al., 2023), which uses an LLM, Minerva (Lewkowycz et al., 2022), to generate theorem proofs that are checked by Isabelle theorem prover. They also proposed a proof repair procedure. In contrast, our interactive decision procedure relies on an automated reasoner to generate proofs and only uses LLMs generated program properties.

The most related work to our framework is Code2Inv (Si et al., 2020), which proposed learning program invariants using machine learning techniques and employed automatic reasoning to verify the programs. The main principle of partitioning responsibilities between automated reasoners and LLMs is similar to our framework. However, we provide a formalization for such interactive procedures with formal calculus and a strategy to use it in practice. Our procedure is more general as it allows the generation of sequences of logically related properties, and we demonstrate that it is more efficient in practice. Finally, recent work by Pei et al. (2023) investigates the potential of invariant generation for Java programming language. While this framework does not incorporate automated reasoning components, it shows the potential of LLMs to uncover program properties.

F PROMPTING THE GPT

In this section, we describe how we automatically constructed the prompts in $\mathcal{O}_{\text{propose}}$ and $\mathcal{O}_{\text{repair}}$, and show examples of the prompts and the GPT outputs. We provided in the supplementary materials the execution traces of LEMUR on solved benchmarks used in our experiments.

F.1 PROPOSING NEW PROPERTIES

Given a program $\mathcal{P}$ and a property represented as a C assert statement in $\mathcal{P}$, we inserted the placeholder lines "`// Line A`", "`// Line B`"... to dedicated program lines as described in 5.1. Our prompt has the following structure:

```
---------
[P1]
Print [P2] as valid C assertions at line[P3] [P4] that
help prove the assertion. Use '&&' or '||' if necessary.
Don't explain. Your answer should be 'assert(...); // line [P5]'
---------
```

`P1` is the program with the placeholder lines and the C assertion; `P2` is either `facts` or `loop invariants` depending on whether all the placeholder lines are in loops; `P3` is either empty or `s` depending on the number of placeholder lines; `P4` are the placeholder line names (e.g., `A, B`); and `P5` is either `A`, if there is only one placeholder line, or `name`, if there are multiple.

Below is an example extracted from the execution of our prototype on an SV-COMP benchmark:

```
int main() {
unsigned char n = (unsigned char) rand();
if (n == 0) {
return 0;
}
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
while (i < n) {
// Line A
v = (unsigned char) rand();
s += v;
++i;
}
assert(s >= v);
return 1;
}
Print loop invariants as valid C assertions at line A that
help prove the assertion. Use '&&' or '||' if necessary.
Don't explain. Your answer should be 'assert(...); // line A'
```

Below are GPT-4's responses on multiple attempts as well as the final set of extracted properties our prototype proceeds to prove.

```
GPT output 1 with penalty 1.5:
assert(i <= n); // line A
assert(s >= i); // line A
GPT output 2 with penalty 1.5:
assert(i <= n); // line A
assert(s >= 0 && v >= 0); // line A
GPT output 3 with penalty 1.5:
assert(i <= n); // line A
assert(s >= 0 && s <= i*UCHAR_MAX); // line A
GPT output 4 with penalty 1.5:
assert(i <= n); // line A
assert(s >= 0 && s <= i*UCHAR_MAX); // line A
GPT output 1 with penalty 2:
assert(i <= n); // line A
assert(s >= 0 && s <= UCHAR_MAX * i); // line A
GPT output 2 with penalty 2:
assert(i <= n); // line A
assert(s >= i*v && s <= (i*255)); // line A
GPT output 3 with penalty 2:
assert(i <= n); // line A
assert(s >= 0 && s <= i*UCHAR_MAX); // line A
GPT output 4 with penalty 2:
```

```
assert(i <= n); // line A
assert(s >= 0 && s <= UCHAR_MAX * i); // line A

Found 6 potential sub-goals
Goal 1: i <= n after line 8
Goal 2: s >= 0 && s <= i*0xff after line 8
Goal 3: s >= 0 && s <= 0xff * i after line 8
Goal 4: s >= i after line 8
Goal 5: s >= 0 && v >= 0 after line 8
Goal 6: s >= i*v && s <= (i*255) after line 8
```

F.2 REPAIRING A PREVIOUS PROPOSAL

The oracle $\mathcal{O}_{\text{repair}}$ uses a similar prompting template:

```
---------
[P1]
Print [P2] as valid C assertions at line A that
help prove the assertion. Your previous answer [P3]
is [P4]. Use '&&' or '||' if necessary. Don't explain.
Your answer should be 'assert(...);'
---------
```

`P1` is the program with the placeholder line (corresponding to the assumption $p = \langle \phi, l \rangle$ to repair) and the C assertion; `P2` is either `facts` or `loop invariants` depending on whether p is in a loop; `P3` is p; `P4` is either `incorrect` or `too weak`.

Below is an example extracted from the execution of our prototype on another SV-COMP benchmark:

```
int main() {
unsigned int n = (unsigned int) rand();
unsigned int x = n, y = 0, z;
while (x > 0) {
x--;
y++;
}
z = y;
// Line A
while (z > 0) {
x++;
z--;
}
assert(y + z == n);
return 1;
}
Print facts as valid C assertions at line A that help
prove the assertion. Your previous answer 'x + z == n'
is too weak. Use '&&' or '||' if necessary. Don't explain.
Your answer should simply be 'assert(...);'
```

The GPT outputs on different prompting attempts with different penalties and the extracted properties:

```
GPT output 1 with penalty 1.5:
assert(x + y == n); // Line A
assert(x + z == y); // Line B
GPT output 2 with penalty 1.5:
assert(x + y == n); // Line A
assert(x + z == n); // Line B
```

```
GPT output 1 with penalty 2:
assert(x + y == n); // Line A
assert(x + z == n); // Line B
GPT output 2 with penalty 2:
assert(x + y == n); // Line A
assert(z + x == n); // Line B

Found 4 potential adapted sub-goals
Goal 1: x + y == n after line 7
Goal 2: x + z == n after line 7
Goal 3: x + z == y after line 7
Goal 4: z + x == n after line 7
```