Large Lemma Miners: Can LLMs Do Induction Proofs for Hardware?

Romy Peled[1,2], **Daniel Kroening**[2,3], **Michael Tautschnig**[2,4] and **Yakir Vizel**[1,2]
[1]Technion – Israel Institute of Technology
[2]Amazon
[3]University of Oxford
[4]Queen Mary University of London
romypel@amazon.com, dkr@amazon.com, tautschn@amazon.at, yvizel@cs.technion.ac.il

Abstract

Large Language Models (LLMs) have shown potential for solving mathematical tasks. We show that LLMs can be utilized to generate proofs by induction for hardware verification and thereby replace some of the manual work done by Formal Verification engineers and deliver value to industry. We present a neurosymbolic approach that includes two prompting frameworks to generate candidate invariants, which are checked using a formal symbolic tool. Our results indicate that with sufficient reprompting, LLMs are able to generate inductive arguments for mid-size open-source RTL designs. For 90% of our problem set, at least one of the prompt setups succeeded in producing a provably correct inductive argument.

1 Introduction

Large Language Models (LLMs) have attracted enormous attention, research effort, and financial investment for their potential to perform mathematical reasoning. The International Math Olympiad (IMO), in particular, has served as a high-profile testbed for demonstrating advanced mathematical reasoning capabilities of state-of-the-art (SOTA) LLMs. Remarkably, SOTA models have matched the performance of leading contestants at the 2025 IMO, with Google DeepMind achieving an official gold-medal standard [1] . It is therefore only natural to consider LLMs as a tool that can assist in complex verification tasks, which currently rely on humans doing mathematical reasoning.

Formal Verification (FV) is a crucial step in the design cycle of hardware, supported by decades of research and substantial industrial investment. Modern FV tools rely on well-established model checking algorithms [Ivrii and Vizel, 2024]. Yet, the application of FV tools to large industrial designs remains a challenge. This is mainly due to the fact that most FV tools operate at the "bit-level". This prevents these tools from constructing a high-level mathematical proof that is agnostic to the number of bits in the design. Constructing such proofs is how a human mathematician approaches such problems. As a result, verifying complex properties that cover meaningful functionality of the design requires extensive manual effort, which is usually performed by experienced FV engineers.

One of the main FV methods is *proof by induction*. This method requires the use of auxiliary lemmas that capture key intermediate facts. These lemmas must first be proven correct and can then be used to construct an inductive argument establishing the property of interest.

In this paper, we set out to answer the following question: *can Large Language Models (LLMs) be used to mine such auxiliary lemmas and construct inductive arguments for proving complex properties?* Such a capability has far-reaching industrial implications, as it may greatly increase the applicability of formal verification for hardware designs.

Our initial experiments with LLMs have exhibited a well-known pattern: the state-of-the-art LLMs produce many answers, and many of the lemmas generated by the LLMs are indeed correct inductive arguments. However, the LLMs also hallucinate: in addition to the correct lemmas, we obtain lemmas that do not even parse, are not inductive, or do not prove the property we are interested in.

We therefore propose a *neurosymbolic* approach that combines the LLM with a formal checking framework that sifts through the LLM's output to find lemmas that form a valid formal proof. Our lemma mining framework has two main components: (i) A single Non-Agentic *Few-Shot* prompt and an *Agentic* LLM prompting setup that are responsible for suggesting candidate lemmas that can serve as inductive arguments; and (ii) a formal reasoning algorithm that finds among the outputs of the LLMs a subset of useful lemmas that form an inductive argument for the given verification task. Then, to evaluate our framework, we use a dataset consisting of 109 verification tasks. A verification task consists of a `SystemVerilog` design and a property to be verified. We evaluate our LLM-based framework on this dataset and show that for 90% of these verification tasks, at least one of our prompting setups produces a provably correct inductive argument. In addition, to demonstrate the effectiveness of our LLM-based framework and to probe the extent to which it can produce nontrivial inductive arguments, we identify verification tasks that are challenging for SOTA tools and evaluate our approach against them. Our results show that our LLM-based approach can solve instances that SOTA tools fail to solve.

Our work makes the following contributions:

[1]https://tinyurl.com/5n7abr7h

1. We present the use case of neurosymbolic lemma generation to construct formal proofs for hardware verification.
2. We implement a lemma mining framework, which uses two LLM prompting setups, one of them Agentic, to generate candidate lemmas, and detail the methods used to evaluate their effectiveness.
3. We provide an empirical study of our suggested framework that highlights both the potential and the current limitations of our approach.

2 Related Work

There is a large body of work on the application of Artificial Intelligence (AI) to mathematics [Davies *et al.*, 2021; Fawzi *et al.*, 2022; Alfarano *et al.*, 2024; He *et al.*, 2023; Pozdnyakov, 2024; Babei *et al.*, 2025; Romera-Paredes *et al.*, 2024]. In particular, many LLM-based approaches are used in mathematics and the generation of formal proofs for interactive theorem provers [Jiang *et al.*, 2023; Lin *et al.*, 2025; Pan *et al.*, 2024; Petrov *et al.*, 2025; Yang *et al.*, 2024b; Huang and Chang, 2023; Clark *et al.*, 2020; Tafjord *et al.*, 2021; Zheng *et al.*, 2025]. These works do not consider hardware.

The works in [Janßen *et al.*, 2024; Kamath *et al.*, 2023; Wu *et al.*, 2024; Yao *et al.*, 2023] study the ability of LLMs to generate loop invariants or class invariants [Sun *et al.*, 2025] for programming languages such as C and RUST. This can be viewed as somewhat similar to our goal of constructing inductive arguments for hardware proofs. However, these two tasks are conceptually different. Loop invariants are local to the loop and require the right state predicate that is preserved by the loop body. By contrast, inductive arguments for a hardware design are global, can include temporal modality/operators, and capture facts about all reachable states of a transition system. The use of Agentic setups as a mechanism to deal with incorrect LLM responses is commonplace for software engineering tasks [Zhang *et al.*, 2025; Yang *et al.*, 2024a].

Machine learning has been applied for hardware proofs in [Giacobbe *et al.*, 2024]. However, this work trains a task-specific neural network rather than using an LLM, and produces formal guarantees in the form of ranking functions.

LLMs have been applied to hardware designs in [Orenes-Vera *et al.*, 2023; Yan *et al.*, 2025; Kande *et al.*, 2024; Thakur *et al.*, 2024; Fan *et al.*, 2025]. In [Orenes-Vera *et al.*, 2023], an LLM is used to generate `SystemVerilog Assertions` (SVA) from RTL code; [Yan *et al.*, 2025] propose a framework for capturing functional behavior in SVA from specification files containing waveform and natural language descriptions; [Kang *et al.*, 2025] evaluate LLMs on the generation of SVA assertions from natural language specifications or from RTL alone; [Kande *et al.*, 2024] investigate the use of LLMs in generating hardware security assertions; [Thakur *et al.*, 2024] fine-tune pretrained LLMs to produce syntactically correct `Verilog` code; and [Fan *et al.*, 2025] construct hardware common weakness enumeration (CWE) knowledge graphs to generate secure `Verilog` code. [Qayyum *et al.*, 2024] incrementally constructs invariants for design modules. Yet, none of these works use LLMs to produce proofs for hardware with respect to a given formal specification.

3 Motivating Example

The following is a short example in our dataset of a round-robin arbiter, taken from `v2c` [Mukherjee *et al.*, 2016]. Comments have been removed to save space.

```
module main(clk,rst,ir0,ir1,ack0,ack1);
    input  clk, rst, ir0, ir1;
    output ack0, ack1;
    reg req0, req1, ack0, ack1, robin;

    task initialize; begin
            ack0 = 0; ack1 = 0; robin = 0;
            req0 = ir0; req1 = ir1;
    end
    endtask

    always @ (posedge clk) begin
    if (rst) initialize;
    else begin
        if (~req0) ack0 <= 0;
        else if (~req1) ack0 <= 1;
        else if (~ack0 & ~ack1) ack0 <= ~robin;
        else ack0 <= ~ack0;

        if (~req1) ack1 <= 0;
        else if (~req0) ack1 <= 1;
        else if (~ack0 & ~ack1) ack1 <= robin;
        else ack1 <= ~ack1;

        if (req0 & req1 & ~ack0 & ~ack1)
          robin <= ~robin;
          req0 <= ir0;
          req1 <= ir1;
      end
  end

  property prop;
    @(posedge clk) disable iff (rst)
      (req1==1 && ack0==1 |-> ##1 ack1==1);
  endproperty
endmodule
```

The specified property, while true, is not an inductive invariant.

When executing the LLM-based framework presented in this paper on this example, all variants generate an inductive strengthening. Among others, the following lemma is generated:

```
property lemma_1;
@(posedge clk) disable iff (rst)
  ~(ack0 && ack1);
endproperty
```

The suggested lemma holds in the given design and forms an inductive strengthening for the specified property. That is, the following property is an inductive invariant:

```
property lemma_prop;
  lemma_1 and prop;
endproperty
```

4 Preliminaries

4.1 Safety Verification

Without loss of generality we formalize the safety verification problem using reachability properties. Our approach is not limited to reachability and is applicable to the full safety fragment of LTL and SVA.

An RTL design D and an LTL safety property φ can be translated into a reachability problem by compiling the RTL and the property into a transition system and a reachability property of the form $\mathbf{AG}p$. A transition system T is a tuple $(\bar{v}, Init, Tr)$, where $\bar{v}$ is a set of Boolean variables that

defines the states of T (i.e., all valuations of $\bar{v}$), $Init$ is a formula with variables in $\bar{v}$ defining the set of initial states, and Tr is a formula that has free variables in $\bar{v} \cup \bar{v}'$, defining the transition relation. A state $s \in T$ is said to be reachable iff there exists a state $s_0 \in Init$ and $(s_i, s_{i+1}) \in Tr$ for $0 \leq i < N$, such that $s = s_N$. The property $\mathbf{AG}p$ requires that p, which is a formula over $\bar{v}$ representing the set of "safe states", holds in all reachable states of T.

Definition 1 (Inductive Invariant)**.** *Given a transition system T, an* inductive invariant *is a formula Inv satisfying: $Init(\bar{v}) \rightarrow Inv(\bar{v})$ (initiation) and $Inv(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \rightarrow Inv(\bar{v}')$ (consecution).*

A transition system T is *safe* iff all reachable states in T satisfy p. Equivalently, there exists a *safe inductive invariant* Inv satisfying: Inv is an inductive invariant and $Inv(\bar{v}) \rightarrow p(\bar{v})$. A *safety* verification problem is to decide whether a transition system T is safe or not, i.e., whether there exists a safe inductive invariant, or a counterexample that shows $\neg p$ is reachable from an initial state.

In simple cases, p is already an inductive invariant. Complex properties, however, are often not inductive and an inductive invariant Inv must be synthesized to prove them. However, model checking algorithms [Clarke *et al.*, 2018], which are the backbone of most EDA tools, sometimes fail to synthesize Inv in a reasonable amount of time.

In such scenarios, manual intervention is required. An FV engineer can add auxiliary lemmas that form a safe inductive invariant, making the safety verification problem tractable for model checking algorithms.

Definition 2 (Inductive Strengthening)**.** *Given a transition system $T = (\bar{v}, Init, Tr)$, let $L = \{\mathbf{AG}q_1, \ldots, \mathbf{AG}q_k\}$ be a set of properties. L is an inductive strengthening for $\mathbf{AG}p$ if $\bigwedge_{i=1}^{k} q_i \wedge p$ is an inductive invariant.*

4.2 Large Language Models

We regard an LLM as a function that maps textual input to textual output. The input may take the form of a single prompt or a sequence of messages (i.e., a conversation context). While LLMs are fundamentally deterministic, they are typically used with deliberately randomized decoding strategies that allow the same input to produce different outputs across different queries. Hence, it is often highly effective to re-prompt the LLM with the same input in order to obtain varied responses and gather additional information, as demonstrated in numerous prior works [Kamath *et al.*, 2023; Farquhar *et al.*, 2024; Kossen *et al.*, 2024]. Throughout this paper, when we query an LLM k times with the same input—whether it is a single prompt in the Non-Agentic setup or a sequence of messages in the Agentic setup—we refer to this as obtaining k sample responses for that input.

Two well-known prompting strategies for LLMs are Few-Shot and Chain-of-Thought (CoT) prompting. In Few-Shot, the input for the LLM contains several auxiliary examples in the form of $\langle question, answer \rangle$. These examples illustrate the expected output format and guide the LLM towards giving appropriate solutions. Chain-of-Thought prompting extends this idea by additionally providing a sequence of reasoning steps for every $\langle question, answer \rangle$ example. This chain of reasoning mimics the way a human would reason about the question and derive the answer. Chain-of-Thought prompting has been shown to elicit reasoning traces from LLMs and to substantially improve their performance on reasoning tasks [Wei *et al.*, 2022].

Algorithm 1: Non-Agentic

Input: A pair (D, φ) of a `SystemVerilog` design and a property, respectively; positive integers k and n
Output: An inductive strengthening L

```
1 S ← GetFewShotPrompt(D, φ, k);
2 C ← ∅;
3 for i = 1 ... n do
4     C ← C ∪ LLM.Prompt(S);
5 L ← FindIndStrength(D, φ, C);
6 return L;
```

5 Lemma-Mining Framework

In this section, we describe our lemma mining framework. We start by describing our prompting configurations that drive the LLM to generate candidate lemmas. We then briefly describe an algorithm that, given a design, property and set of candidate lemmas, finds an inductive strengthening for the property.

5.1 Prompting Setups

To generate candidate lemmas with LLMs, we develop two configurations as described below. Throughout this section we consider a pair of an RTL design and a property, (D, φ), for which we would like to construct an inductive strengthening.

Single Prompt, Non-Agentic

The prompt used in Algorithm 1 (line 1) is few-shot. Note that it remains constant throughout the execution of the algorithm. To create the few-shot prompt, we construct a pool of Chain-of-Thought responses (CoT pool). This pool consists of triples that include an RTL design, a property, and a set of lemmas that serve as an inductive strengthening w.r.t. the property and design. Since this is a CoT, responses also include detailed reasoning as to why the given set of lemmas serves as an inductive strengthening.

Given a pair (D, φ), the constructed few-shot prompt for this pair has a fixed template followed by k examples (i.e., triples) from the CoT pool described above, where k is a parameter. The fixed template specifies the task, provides general guidelines, and defines the expected output format. Note that since only k examples are chosen from the pool, it is desirable to choose those that can guide the LLM towards the best response. For this purpose, our framework embeds (D, φ) and all entries in the pool using *sentence transformers* [Reimers and Gurevych, 2019], and chooses the k examples that are most similar to (D, φ) based on the dot product of the computed embeddings.

Since the LLM may return different responses for the same prompt, we employ sampling (i.e., prompting the LLM multiple times with the same prompt, see Section 6). The number of re-prompting iterations is determined by a parameter n

Algorithm 2: Agentic

Input: A pair (D, φ) of a SystemVerilog design and a property, respectively; positive integers k and n
Output: An inductive strengthening L

1 $S \leftarrow$ GetFewShotPrompt(D, φ, k);
2 $C \leftarrow \emptyset$, $All \leftarrow \emptyset$;
3 **for** $i = 1 \dots n$ **do**
4 $C \leftarrow$ LLM.Prompt(S);
5 $L \leftarrow$ FINDINDSTRENGTH(D, φ, C);
6 **if** $L \neq \emptyset$ **then**
7 **return** L;
8 $All \leftarrow All \cup C$;
9 $L \leftarrow$ FINDINDSTRENGTH(D, φ, All);
10 **if** $L \neq \emptyset$ **then**
11 **return** L;
12 $S \leftarrow$ Concat $(S,$ GenerateRepairMsg$(C))$;
13 **return** $\emptyset$;

(lines 3–4). All LLM responses are collected in a set of candidate lemmas C (line 4). Then, the set C is passed to FINDINDSTRENGTH (see Section 5.2), which is an algorithm that attempts to return an inductive strengthening, if such exists in C (line 5).

LLM Agent

In the Agentic setting of Algorithm 2, lemma generation is framed as an iterative interaction between an LLM and a verifier. Instead of generating one, static batch of candidate lemmas, the LLM is engaged in a loop of proposal, verification, and refinement.

Similar to Algorithm 1, it starts with the same few-shot prompt (line 1). In each iteration, the LLM is asked to generate a set of candidate lemmas (line 4). Upon receiving the candidate lemmas from the LLM, FINDINDSTRENGTH is invoked (line 5) to determine whether the verification task has been *solved*. That is, an inductive strengthening has been found. Throughout a conversation (i.e., the loop in lines 3–12), a record of all proposed lemmas is maintained in the set All. If the candidates generated by the LLM in a given iteration i do not contain an inductive strengthening, FINDINDSTRENGTH is executed on the set of lemmas All (line 9). If an inductive strengthening is found in All, it is returned and the algorithm terminates.

Otherwise, a feedback message is constructed indicating that the generated candidates do not include an inductive strengthening (line 12). The feedback also indicates for each suggested lemma whether it holds or not, based on the analysis performed by FINDINDSTRENGTH. In addition, every two rounds, the feedback also includes general reminders of the task and the expected output format.

5.2 Finding an Inductive Strengthening

Given the pair (D, φ) and a set of candidate lemmas, we need to determine whether the candidate lemmas admit an inductive strengthening for the design and property. We implement an algorithm FINDINDSTRENGTH, which attempts to return a subset of lemmas that can be used as inductive strengthening, if such a subset exists. Since this algorithm is based on formal

Algorithm 3: FINDINDSTRENGTH

Input: A SystemVerilog design D, a property φ, and a set of candidate lemmas $Cand$
Output: An inductive strengthening L

1 $(T, AGp) \leftarrow$ GetTransitionSystemAndProp(D, φ);
2 $C \leftarrow \emptyset$, $Ind \leftarrow \emptyset$, $All \leftarrow \emptyset$;
3 **for** $\ell_i \in Cand$ **do**
4 **if** IsOneInductive(T, ℓ_i) **then**
5 $Ind \leftarrow Ind \cup \{\ell_i\}$;
6 **if** IsOneInductive$(T, \ell_i \wedge p)$ **then**
7 **return** $\{\ell_i\}$;
8 **else if** HoldsUpToBound(T, ℓ_i, N) **then**
9 $C \leftarrow C \cup \{\ell_i\}$;
10 $All \leftarrow$ CONCAT(SORT(Ind), SORT(C));
11 $L \leftarrow \emptyset$;
12 **for** $i = 1 \dots All$.Size() **do**
  // L is the first i elements of All
13 $L \leftarrow L \cup \{All[i]\}$;
14 **if** IsOneInductive$(T, \bigwedge L \wedge p)$ **then**
15 **return** L;
16 **return** $\emptyset$;

reasoning, if it determines that a subset is indeed an inductive strengthening, then we conclude that the property holds.

FINDINDSTRENGTH appears in Algorithm 3. To identify a possible subset, FINDINDSTRENGTH partitions the set of candidates into two sets of lemmas (lines 3–9): those that are inductive invariants (line 5), and those that hold up to a given bound N (line 9). If one of the candidate lemmas is an inductive strengthening by itself (line 6), the algorithm terminates and returns it. Candidate lemmas that fail both checks (e.g., do not hold and have a counterexample) are discarded.

Next, FINDINDSTRENGTH merges the two sets of inductive lemmas and lemmas that hold up to bound N into a sorted list (line 10). The list is ordered such that lemmas that are inductive invariants appear before lemmas that hold up to bound N. Clearly, lemmas that are known to be inductive invariants have higher chances of being part of an inductive strengthening. For lemmas of the same category, a lexicographic order is used to enforce an ordering that is both deterministic and reproducible.

Lastly, FINDINDSTRENGTH iterates over the prefixes of the sorted list of lemmas and checks if some prefix is an inductive strengthening (lines 12–15). If such a prefix is found, the algorithm terminates, and returns it. Otherwise, an empty set is returned, indicating that FINDINDSTRENGTH failed to find an inductive strengthening. Note that we only look for *some* inductive strengthening, and not for a maximal or minimal subset (e.g., as in Houdini [Flanagan *et al.*, 2001])[2]. Moreover, since FINDINDSTRENGTH does not consider all possible subsets, it may fail to find an inductive strengthening represented by a subset that is not a prefix. This limitation stems from the fact that not all lemmas in the sorted list are

[2]There are many possible implementations that can be used to identify such a subset, e.g., Houdini, checking all possible subsets in parallel, or [Ivrii *et al.*, 2014] and other similar approaches.

Algorithm 4: `IsOneInductive`

Input: A transition system T, and a formula ψ
Output: *true* if ψ is an inductive invariant w.r.t. T, otherwise *false*

if IsSAT$(Init(\bar{v}) \wedge \neg\psi(\bar{v}))$ **then**
  return *false*
if IsSAT$(\psi(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \wedge \neg\psi(\bar{v}'))$ **then**
  return *false*
return *true*

individually inductive.

FINDINDSTRENGTH uses the function `IsOneInductive` to check if a property is an inductive invariant, and the function `HoldsUpToBound` to check if the property has a counterexample of up to some given length. The implementation of `IsOneInductive` appears in Algorithm 4. A call to `IsOneInductive`(T, ψ) performs *initiation* and *consecution* checks with respect to the transition system T and the property ψ using a SAT-solver [Vizel *et al.*, 2015] (see Definition 1). `HoldsUpToBound`(T, ψ, N) invokes a Bounded Model Checking (BMC) [Biere *et al.*, 2003] engine with respect to the transition system T, a property ψ and a bound N.

Theorem 1. *If* FINDINDSTRENGTH *returns a non-empty set L, then L is an inductive strengthening for φ, and consequently φ holds in the design D.*

Following the above Theorem, we say that an example is *solved* by an LLM if an invocation of FINDINDSTRENGTH on some set of candidate lemmas proposed by the LLM returns a non-empty set.

6 Experimental Setup

6.1 Prompt Configuration

For the few-shot prompt (Algorithm 1), we experimented with $k = 0, 1, 2, 3, 4$ few-shot examples and report the aggregate results across these configurations. We provide a per-k breakdown in Table 3. For the Agentic setup, we allow a maximum of seven iterations ($n = 7$ in Algorithm 2). While it is possible to provide counterexamples for suggested lemmas that fail verification, we empirically determined that it degraded the performance of the LLMs, possibly because including counterexamples made the feedback responses substantially longer. The prompts we use can be found in [Peled *et al.*, 2026a; Peled *et al.*, 2026b].

6.2 Formal Reasoning Configuration

For the implementation of FINDINDSTRENGTH we used EBMC [Mukherjee *et al.*, 2015] (version 5.9). We check if a property is an inductive invariant using k-induction [Sheeran *et al.*, 2000] with $k = 1$ (i.e., 1-inductive checks). For the bounded correctness checks (i.e., Bounded Model Checking [Biere *et al.*, 2003]), we use a bound of 30. We use bound 30 as a practical bound for bug-finding, sufficient to detect short counterexamples and quickly filter out incorrect candidates while also keeping the cost of this check manageable. All queries use a timeout of 120 seconds.

6.3 LLMs and Inference Parameters

We employ six state-of-the-art LLMs: `OpenAI`'s `GPT-5`[3] and `Anthropic`'s `Claude4`, `Claude4.5-Sonnet`, `Claude4.5-Haiku`, `Claude4.5-Opus` and `Claude4.7-Opus`[4].

For all Claude models, we take the default inference configuration provided with `Amazon Bedrock`'s `Converse API`. For `GPT-5` we take the default hyper-parameters assigned within `OpenAI`'s `Responses API`. We emphasize that `OpenAI`'s `Responses API` is used without supplying IDs of previous responses, and thus our prompting queries are **stateless**.

6.4 Re-Prompt Sampling

In the case of the Agentic setup, we opt for sampling one response from the LLM for each sequence of messages.

For the Non-Agentic setup, we experiment with a varying number of samples, up to a total of five samples. When sampling multiple responses in this setup, we need to decide how to invoke FINDINDSTRENGTH on them. In principle, for five sampled responses there are five sets of candidate lemmas. Namely, there are 2^5 possible combinations we can consider. Each one of them gives rise to a different possible set of candidate lemmas that can be considered by FINDINDSTRENGTH. We opt for a simple approach in which we aggregate all candidate lemmas generated across all sampled responses into a single set and invoke FINDINDSTRENGTH on them all.

6.5 Dataset

We compile a dataset of 57 `SystemVerilog` designs with corresponding safety properties in SVA. Some modules have multiple associated properties, and some are parametrized and contribute multiple design instances. As discussed before, we consider one property per example, resulting in a total of 109 (D, φ) pairs. The designs and properties in our dataset are taken from several sources: `VCEGAR` [Jain *et al.*, 2007], `NeuralMC` [Giacobbe *et al.*, 2025], `VIS` [Irfan *et al.*, 2016], `v2c` [Mukherjee *et al.*, 2016], an Edge-Detector from [Circuitcove.com, 2026], and a counter from [Krishnan *et al.*, 2019]. In addition, we include one parametrized example from [Irfan *et al.*, 2016] (`Buffers`), and 3 additional parametrized designs of a FIFO (`Queues`), Least-Recently-Granted (`LRG`) arbiter and a content-addressed-memory (`CAM`).

In addition to these hardware designs, we collected 47 short examples adapted from programs originally written in `C`. These examples originate from a benchmark suite for loop invariant generation [Microsoft, 2024; Kamath *et al.*, 2023]. Each example in this benchmark is a short `C` program containing a loop and a property to be verified (`assert` statement). To adapt these for our needs, we translate the `C` programs into simple finite state machines (FSMs) in `SystemVerilog`. Of these examples, 16 are used to populate our *CoT pool* described earlier. For seven of these examples, we generate a total of 25 additional variants, which are examples with structurally similar

[3] snapshot `gpt-5-2025-08-07`

[4] anthropic.claude-sonnet-4-20250514-v1:0, anthropic.claude-sonnet-4-5-20250929-v1:0, anthropic.claude-haiku-4-5-20251001-v1:0, anthropic.claude-opus-4-5-20251101-v1:0 and anthropic.claude-opus-4-7 in `Amazon Bedrock`, respectively

underlying FSM, but small variations such as different parameter instantiations, different register sizes, swapped names for registers, a distracting extra register and/or an uninitialized register. Full details about the benchmarks are provided in [Peled *et al.*, 2026a].

As our framework requires the RTL source code, the number of available open-source examples is limited. In particular, we cannot use numerous examples provided by the hardware model checking competitions (HWMCCs) [Biere *et al.*, 2024], as they are given in a *netlist* format and do not include the original RTL code.

Lastly, 3 other examples from [Circuitcove.com, 2026] are used solely in the CoT pool. These examples are simple hardware constructs – a frequency divider, a bidirectional counter and a 3-tap FIR, for which we manually write properties.

7 Experimental Results

The experiments were executed on Amazon EC2 m6i.32xlarge instances with 128 vCPUs and 512 GB memory for a total of 1519.9 hours, of which 20% were used by the LLMs and 80% by the symbolic engine. Across all runs, the LLMs generated 277.41 million tokens. The maximum time it took our framework to solve an example was 5 hours, with a mean and median of $232.54s$, and $85.1s$, respectively. Instructions for reproducing the results can be found in [Peled *et al.*, 2026b].

Results are summarized in Table 1. As can be seen from the table, `Claude4.7-Opus` achieves the best performance. Overall, using all configurations, the Agentic and Non-Agentic setups solve 98 and 90 out of 109 examples, respectively. This accounts for a success rate of 90% for the Agentic setup and 83% for the Non-Agentic setup. Excluding examples that are part of the CoT pool, or variants thereof, 84% of the examples are solved by at least one of the configurations.

7.1 SVA Familiarity

We begin by noting that all LLMs occasionally exhibit gaps in their familiarity with SVA syntax. This further underscores the benefit of re-prompting to increase our collected pool of syntactically correct candidate lemmas. `GPT-5` in one case incorrectly stated that `|->` is SVA's non-overlapping operator, and provided a faulty reasoning step as a result. We have also observed an instance of an undefined variable sum', referring to the next state of an existing register sum. Many models used invalid loop constructs in their assertions, particularly for lemmas involving indices of large registers. We also encountered some encoding issues with `Claude4`'s lemmas, where the generated lemmas contained non-ASCII characters, such as the Unicode symbol of the bidirectional arrow $\Leftrightarrow$ instead of the valid ASCII representation `<->`. `Claude4.5-Haiku` has the highest error rate, roughly twice that of `GPT-5`. Table 1 reports the number of errors per model and setup.

7.2 Agentic vs. Non-Agentic

We observe indicators for some relative advantage of the Agentic setup. Note that the Agentic setup is in a sense more efficient: for half of the LLMs, the Agentic setup solves more examples than its Non-Agentic counterpart with a substantially smaller number of lemmas. The apparent advantage of the Agentic setup can be attributed to the fact that FINDINDSTRENGTH is invoked on more subsets in this setup. Invoking FINDINDSTRENGTH on more subsets results in more queries to the verification tool with more combinations of candidate lemmas, which increases the chance of finding an inductive strengthening. However, we have observed cases which suggest that the additional calls to FINDINDSTRENGTH are not the sole advantage of the Agentic setup over the Non-Agentic one. In these cases, a given LLM in the Non-Agentic setup fails to find a strengthening for a property, whereas its Agentic counterpart, over the course of several rounds, is steered away from incorrect or non-inductive lemmas and eventually proposes an inductive strengthening with a lemma that does not appear in the Non-Agentic setting at all. We therefore conjecture that the Agentic setup has an added value.

7.3 Solved and Unsolved Examples

Of 109 examples, 11 were not solved by any setting. A summary of the number of unsolved examples per category is provided in [Peled *et al.*, 2026a]. As expected, for all examples that were used to construct the CoT pool, the LLMs copied the lemmas they were shown, at times along with additional lemmas. Hence, all models solved the majority of this subset. For the `C` programs group, the LLMs demonstrated no difficulty with swapped variable names and different parameter instantiations, but did struggle when the variants required an extra lemma accounting for possible overflow and/or accounting for an uninitialized variable. Two unsolved examples from `v2c` are two of three examples in our dataset that include properties with a sequence of three time frames (`##2`) or more. In fact, the design in one of these examples appears in another example in our dataset. However, the other example has a property that refers to two time frames (`##1`), and it is solved. Another unsolved example from `v2c` is a conjunction of 6 conceptually distinct, operator-dense properties.

7.4 Challenging Benchmarks

The lemmas obtained in our experiments can vary widely in their complexity; for some designs, an inductive strengthening may be a simple auxiliary invariant, whereas others may require nontrivial ones. In this subsection, we therefore target a more stringent question—Can LLMs derive *nontrivial* lemmas when constructing an inductive strengthening? It is hard to characterize and agree upon what constitutes a "nontrivial" lemma. Therefore, we adopt a pragmatic approach wherein we use SOTA model checkers as a proxy for nontriviality. Notable SOTA tools in our domain are Cadence's JasperGold (version 2024.06p002), Synopsys' VC-Formal (version X-2025.06-SP2P3) and the open-source IC3-based rIC3 [Su *et al.*, 2025] (version 1.5.2). We consider an example to require a *nontrivial* lemma if one of the aforementioned tools does not prove its property in under 10 minutes. We justify this decision by the fact that the SOTA tools in our domain are based on mature and well-established algorithms, backed by decades of research and extensive industrial adoption.

We identify 31 such challenging examples in our dataset:

(i) a content-addressed memory design (`CAM`, 2 instances of different sizes); (ii) a Least-Recently-Granted arbiter (`LRG`, 3 instances); (iii) a buffer design from `vis` (3 instances); (iv) two

	Agentic					Non-Agentic				
Model	Total Lemmas	Correct	1-Inductive	Error	Solved	Total Lemmas	Correct	1-Inductive	Error	Solved
`GPT-5`	8090	5976	3187	175	84	12110	8647	5592	245	77
`Claude4`	6465	4651	2302	167	84	6720	4709	2839	307	68
`Claude4.5-Sonnet`	10255	7941	3636	203	89	8347	5982	3531	427	70
`Claude4.5-Haiku`	10479	7189	3204	307	79	10384	6557	3521	503	71
`Claude4.5-Opus`	8290	6851	3292	99	89	8498	6640	3956	271	84
`Claude4.7-Opus`	6251	5025	2093	208	94	6233	4910	2840	284	88
Virtual Best					98					90

Table 1: Aggregate per (model, setup) across all main-experiment runs ($k \in \{0, 1, 2, 3, 4\}$ few-shot examples; up to five samples per Non-Agentic prompt). *Total Lemmas*: candidate lemmas generated by the LLM. *Error*: candidates the verifier could not parse or otherwise failed on. *Correct*: candidates that hold in the design up to bound 30 (BMC). *1-Inductive* ($\subseteq$ *Correct*): 1-inductive invariants. *Solved*: verification tasks (out of 109) for which FINDINDSTRENGTH extracted an inductive strengthening from the candidates. *Virtual best*: tasks solved by at least one of the six models in the given setup.

designs checking equivalence between FIFOs (`Queues`). One of these designs is parametrized and contributes two instances. Others are drawn from `VIS` and the additional categories of our dataset; (v) 20 counter designs (`Counters`), 18 drawn from the `C` programs category, one from `VCEGAR`, and one example adapted from [Krishnan *et al.*, 2019].

The comparison of our approach and the aforementioned SOTA tools on these examples is summarized in Table 2. A timeout of one hour was used for all tools.

Our pipeline solved 27 out of these 31 examples. For the larger buffer, the LLMs produced a valid inductive strengthening, but the subsequent induction check exceeded our given timeout of 2 minutes. We note that JasperGold can complete that induction check within the 2 minute timeout and report this example as TIMEOUT†. Another example was solved by the Agentic `Claude4.7-Opus` in 2 hours which exceeds the given timeout of one hour, and we report it as TIMEOUT‡. Interestingly, for the larger LRG-arbiter instances, the main obstacle for some LLMs such as `Claude4.5-Sonnet` was using invalid loop constructs in their lemmas. Enumerating the loops explicitly does yield valid inductive strengthenings.

Category	Design	Our virtual best	rIC3	JasperGold	VCF
CAM	simple_cam_16	356.06	0.06	188.00	TIMEOUT
	simple_cam_8	22.98	0.03	187.00	TIMEOUT
arbiters	lrg_arb_lrg_16	TIMEOUT	TIMEOUT	907.00	750.00
	lrg_arb_lrg_4	28.09	0.02	3.00	1.00
	lrg_arb_lrg_8	TIMEOUT	2.55	5.00	2.00
buffers	buffer_128	TIMEOUT†	0.06	TIMEOUT	TIMEOUT
	buffer_32	261.00	0.01	188.00	TIMEOUT
	buffer_64	419.10	0.03	435.00	TIMEOUT
counters	counter_66	12.27	TIMEOUT	TIMEOUT	15.00
	dp	15.10	TIMEOUT	6.00	TIMEOUT
	ex15	9.91	TIMEOUT	187.00	TIMEOUT
	ex3	7.08	TIMEOUT	54.00	1.00
	ex4	6.89	TIMEOUT	3.00	1.00
	ex51	8.01	TIMEOUT	4.00	2.00
	ex51_57_1	19.32	TIMEOUT	6.00	TIMEOUT
	ex80	13.10	TIMEOUT	2340.00	TIMEOUT
	gr2006	10.42	275.51	820.00	1.00
	gulwani_cegar2	8.21	TIMEOUT	187.00	TIMEOUT
	gulwani_cegar2_1	18.79	123.82	287.00	TIMEOUT
	gulwani_cegar2_2	19.60	119.36	356.00	TIMEOUT
	gulwani_cegar2_3	25.26	TIMEOUT	233.00	TIMEOUT
	gulwani_cegar2_4	21.27	TIMEOUT	218.00	TIMEOUT
	gulwani_cegar2_5	29.40	TIMEOUT	TIMEOUT	TIMEOUT
	gulwani_fig1a	731.92	TIMEOUT	192.00	1.00
	gulwani_fig1a_2	TIMEOUT‡	TIMEOUT	853.00	1.00
	gulwani_fig1a_4	859.28	TIMEOUT	613.00	1.00
	gulwani_fig1a_5	1235.65	TIMEOUT	192.00	1.00
	gulwani_fig1a_6	970.68	TIMEOUT	TIMEOUT	1.00
queues	fifo_ref_16	96.93	0.04	TIMEOUT	TIMEOUT
	fifo_ref_64	195.33	0.15	TIMEOUT	TIMEOUT
	fifo_vis	849.16	619.63	1580.00	TIMEOUT

Table 2: Runtime comparison (in seconds)

7.5 Ablations

Number of CoT Examples

The Few-Shot prompt can be constructed using an arbitrary number k of examples drawn from the CoT pool. In our experiments, we considered $k \in \{0, 1, 2, 3, 4\}$ and reported results aggregated across these five choices. To assess the sensitivity of our framework to the parameter k (i.e. the number of CoT examples used in the prompt), we evaluate our framework with a fixed $0 \leq k \leq 5$. Table 3 summarizes the results of this experiment. For each configuration, the table presents the percentage of examples solved for every value of $0 \leq k \leq 5$. Darker shades of blue indicate a higher success rate. For every LLM, the results are broken down by three dataset categories: 1. *seen* examples, i.e., examples that appear in the CoT pool; 2. *C examples* that do not appear in the CoT pool (i.e., from *c, generalize* and *c, others*); 3. and all remaining designs (from `VCEGAR`, `VIS`, `v2c`, `NeuralMC`, and the additional parametrized designs). The VB (Virtual Best) column aggregates the results across the different values of k, while the Virtual Best row presents the aggregation across all LLMs.

We observe some variation in the sensitivity of different settings to the number of CoT examples. The Agentic `Claude4.7-Opus` configuration is relatively robust, while Agentic `Claude4.5-Haiku` is more affected. Increasing the number of few-shot examples does not produce a monotonic improvement, but omitting the few-shot examples altogether does reduce performance: the virtual-best success rate increases from 82% of examples for $k = 0$ to 86% for $k = 4$. We do not observe an improvement when increasing the number of few-shot examples to $k = 5$.

Number of Iterations for the LLM Agent

In the Agentic setup, the LLM interacts with the symbolic engine for n iterations. To assess how sensitive our framework is to the iteration count n, we evaluate the Agentic framework across $1 \leq n \leq 10$ iterations, and report the results in Figure 1. For each n, we provide the virtual best (VB) results broken down by model, by parameter $0 \leq k \leq 4$, and aggregated across all models and k values. In addition, we perform the

Model	Tag	Total	Agentic							Non-Agentic						
			$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	VB	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	VB
Claude4	c examples	31	74%	**84%**	71%	81%	77%	81%	94%	**68%**	65%	61%	58%	61%	65%	74%
	others	60	50%	48%	48%	47%	**52%**	48%	62%	**40%**	38%	37%	37%	38%	38%	47%
	seen	18	72%	94%	94%	94%	**100%**	89%	100%	61%	**94%**	**94%**	89%	**94%**	89%	94%
Claude4.5-Haiku	c examples	31	61%	71%	68%	**74%**	**74%**	65%	77%	52%	58%	**61%**	58%	**61%**	**61%**	84%
	others	60	40%	35%	35%	**47%**	45%	42%	67%	35%	35%	**40%**	37%	37%	32%	50%
	seen	18	50%	**89%**	72%	**89%**	78%	**89%**	94%	56%	**94%**	**94%**	**94%**	89%	89%	94%
Claude4.5-Opus	c examples	31	74%	81%	77%	**84%**	77%	81%	90%	77%	**84%**	74%	68%	68%	71%	87%
	others	60	68%	**72%**	67%	63%	67%	70%	73%	**60%**	**60%**	57%	**60%**	57%	**60%**	68%
	seen	18	89%	**100%**	94%	94%	94%	94%	100%	83%	**94%**	**94%**	89%	**94%**	89%	94%
Claude4.5-Sonnet	c examples	31	81%	**87%**	81%	84%	**87%**	**87%**	97%	61%	65%	61%	**71%**	61%	58%	71%
	others	60	60%	58%	58%	52%	58%	**63%**	70%	45%	43%	43%	38%	35%	**48%**	57%
	seen	18	89%	94%	94%	94%	**100%**	**100%**	100%	72%	**94%**	89%	89%	89%	89%	94%
Claude4.7-Opus	c examples	31	90%	90%	94%	90%	90%	**100%**	100%	77%	84%	81%	**87%**	71%	74%	97%
	others	60	67%	**73%**	70%	70%	**73%**	72%	75%	60%	60%	62%	60%	**63%**	57%	68%
	seen	18	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	100%	89%	94%	94%	**100%**	94%	89%	100%
GPT-5	c examples	31	74%	77%	71%	77%	74%	**81%**	87%	74%	77%	**81%**	**81%**	**81%**	77%	84%
	others	60	50%	50%	52%	**55%**	48%	48%	70%	**48%**	43%	40%	43%	45%	42%	57%
	seen	18	72%	**94%**	**94%**	**94%**	**94%**	**94%**	94%	89%	89%	**94%**	**94%**	89%	**94%**	94%
Virtual best	c examples	31	90%	94%	94%	100%	97%	100%	100%	84%	87%	87%	94%	84%	87%	97%
	others	60	73%	75%	72%	72%	77%	75%	82%	68%	67%	67%	65%	67%	67%	72%
	seen	18	100%	100%	100%	100%	100%	100%	100%	89%	94%	94%	100%	94%	94%	100%

Table 3: Few-shot ablation results

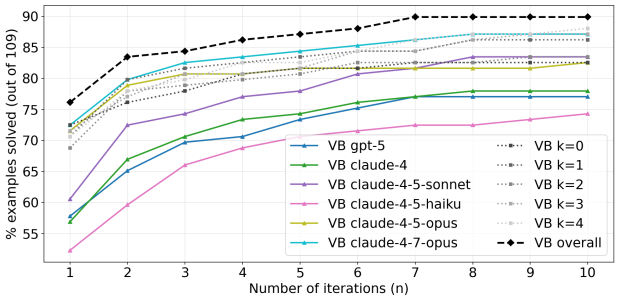

Figure 1: Examples solved by number of Agentic iterations. "VB" (virtual best) per-model: examples solved by at least one few-shot configuration for that model; "VB" per few-shot: examples solved by at least one of the models for that few-shot configuration; "VB" overall: examples solved by at least one of the models and few-shot configurations.

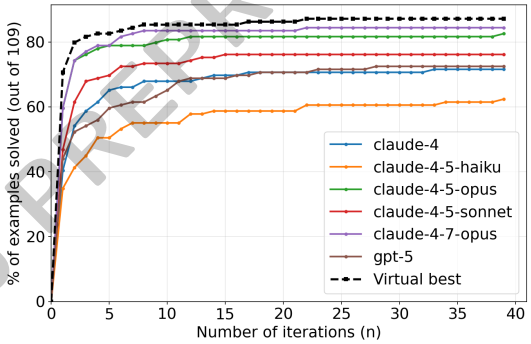

Figure 2: Examples solved within 2 hours by number of Agentic iterations. Results are shown for configurations with $k = 1$ few-shot examples.

following experiment: We fix the number of few-shot examples to $k = 1$, bound the Agentic setup with a timeout of 2 hours and let it use as many iterations as it needs (no bound on n) to complete the verification task. Figure 2 presents a cactus plot that summarizes the results of this experiment. The X-axis represents the number of iterations n, while the Y-axis represents the percentage of solved examples. As can be seen from the plot, the majority of examples are solved with up to 5 iterations ($n \leq 5$), while all LLMs saturate within approximately 10 iterations.

8 Limitations and Threats to Validity

8.1 Data Contamination

Data contamination is a well-known concern when evaluating LLMs on generalization tasks. If benchmark instances or fragments thereof appear in pretraining data, strong performance may result from memorization, and not from genuine reasoning capabilities of the LLM, undermining the validity of the evaluation. Some of the examples in our dataset have been publicly available long enough to plausibly appear in the pretraining corpus of the LLMs we evaluate. However, to the best of our knowledge, there are no available auxiliary lemmas which could serve as inductive strengthenings for them. More specifically, the benchmarks collected from [Irfan *et al.*, 2016; Mukherjee *et al.*, 2016; Jain *et al.*, 2007] were originally designed for a different task, which reduces the risk of reference answers being available. As for the C-examples we collected from [Microsoft, 2024], they originally appear in a substantially different structure and in a different language than the one we use in our experiments. Since their task is closer to ours, there is greater concern if answers for them, i.e., loop invariants, exist online. If so, the LLMs could have leveraged their exposure to these loop invariants by means of transfer-learning. Either way, we contend that our reported evaluation results should not be dismissed as memorization.

8.2 Representativeness of Dataset

While the models we prompt in the two setups perform very well collectively on our dataset, this dataset may not be representative of real-world industrial examples. Furthermore, our lemma-mining framework requires the RTL source code

in `SystemVerilog`, and is not applicable to netlist-level representations of hardware designs. We are thus unable to assess our current framework on most publicly available circuits.

8.3 Selection of CoT-Pool

Our CoT examples were selected manually as a convenience sample, which may introduce selection bias. We chose simple hardware designs from [Circuitcove.com, 2026], a short example from [Jain *et al.*, 2007], and examples from [Microsoft, 2024]. Most of the CoT examples are taken from [Microsoft, 2024], which is independent of the other benchmark sets, reducing the risk that they are representative. The example taken from [Jain *et al.*, 2007] is short and structurally simple, and is unlikely to be representative of other designs within this subset. For [Microsoft, 2024], we explicitly report results on generalization within this subset and performance on the seen examples, making any potential bias visible.

9 Conclusions

We present a neurosymbolic framework that can be used to automatically generate inductive proofs for hardware verification. While this is only a first step, we believe such an approach holds promise for industrial settings, where it can assist FV engineers when dealing with complex properties.

In future work, we would like to extend our approach to handle designs with multiple properties simultaneously, allow it to handle larger designs, and evaluate it in an industrial setting.

Acknowledgments

The research leading to these results is also partially funded by the Israel Science Foundation (ISF), grant no. 2875/21 and by the European Union (ERC, StrongMC, 101231745). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. The authors would also like to thank Amit Eisinger for his help in conducting the rIC3 experiments.

References

[Alfarano *et al.*, 2024] Alberto Alfarano, François Charton, and Amaury Hayat. Global Lyapunov functions: A long-standing open problem in mathematics, with symbolic transformers. In *NeurIPS*, 2024.

[Babei *et al.*, 2025] Angelica Babei, François Charton, Edgar Costa, Xiaoyu Huang, Kyu-Hwan Lee, David Lowry-Duda, Ashvni Narayanan, and Alexey Pozdnyakov. Learning Euler factors of elliptic curves. *CoRR*, 2502.10357, 2025.

[Biere *et al.*, 2003] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 2003.

[Biere *et al.*, 2024] Armin Biere, Nils Froleyks, and Mathias Preiner. Hardware model checking competition 2024. In *Formal Methods in Computer-Aided Design, FMCAD*. IEEE, 2024.

[Circuitcove.com, 2026] Circuitcove.com. https://circuitcove.com/, 2026.

[Clark *et al.*, 2020] Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language. In *IJCAI*, 2020.

[Clarke *et al.*, 2018] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018.

[Davies *et al.*, 2021] Alex Davies, Petar Velickovic, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomasev, Richard Tanburn, Peter W. Battaglia, Charles Blundell, András Juhász, Marc Lackenby, Geordie Williamson, Demis Hassabis, and Pushmeet Kohli. Advancing mathematics by guiding human intuition with AI. *Nat.*, 2021.

[Fan *et al.*, 2025] Fanghao Fan, Yingjie Xia, and Li Kuang. SecV: LLM-based secure Verilog generation with clue-guided exploration on hardware-CWE knowledge graph. In *International Joint Conference on Artificial Intelligence, IJCAI*, pages 8049–8057, 2025.

[Farquhar *et al.*, 2024] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nat.*, 2024.

[Fawzi *et al.*, 2022] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 2022.

[Flanagan *et al.*, 2001] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 2001.

[Giacobbe *et al.*, 2024] Mirco Giacobbe, Daniel Kroening, Abhinandan Pal, and Michael Tautschnig. Neural model checking. In *NeurIPS*, 2024.

[Giacobbe *et al.*, 2025] Mirco Giacobbe, Daniel Kroening, Abhinandan Pal, and Michael Tautschnig. Let a neural network be your invariant. In *NeurIPS*, 2025.

[He *et al.*, 2023] Yang-Hui He, Kyu-Hwan Lee, and Thomas Oliver. Machine learning invariants of arithmetic curves. *J. Symb. Comput.*, 2023.

[Huang and Chang, 2023] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. In *ACL*, 2023.

[Irfan *et al.*, 2016] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2SMV: A tool for word-level verification. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016*, pages 1156–1159. IEEE, 2016.

[Ivrii and Vizel, 2024] Alexander Ivrii and Yakir Vizel. Bit-level model checking. In *Handbook of Computer Architecture*. Springer, 2024.

[Ivrii *et al.*, 2014] Alexander Ivrii, Arie Gurfinkel, and Anton Belov. Small inductive safe invariants. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, 2014.

[Jain *et al.*, 2007] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. VCEGAR: Verilog counterexample guided abstraction refinement. In *TACAS*. Springer, 2007.

[Janßen *et al.*, 2024] Christian Janßen, Cedric Richter, and Heike Wehrheim. Can ChatGPT support software verification? In *FASE*, 2024.

[Jiang *et al.*, 2023] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *ICLR*. OpenReview.net, 2023.

[Kamath *et al.*, 2023] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *CoRR*, 2311.07948, 2023.

[Kande *et al.*, 2024] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (Security) assertions by large language models. *IEEE Trans. Inf. Forensics Secur.*, 2024.

[Kang *et al.*, 2025] Minwoo Kang, Mingjie Liu, Ghaith Bany Hamad, Syed M. Suhaib, and Haoxing Ren. FVEval: Understanding language model capabilities in formal verification of digital hardware. In *Design, Automation & Test in Europe Conference, DATE*, 2025.

[Kossen *et al.*, 2024] Jannik Kossen, Jiatong Han, Muhammed Razzak, Lisa Schut, Shreshth A. Malik, and Yarin Gal. Semantic entropy probes: Robust and cheap hallucination detection in LLMs. *CoRR*, 2406.15927, 2024.

[Krishnan *et al.*, 2019] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In *Computer Aided Verification, CAV*. Springer, 2019.

[Lin *et al.*, 2025] Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. Lean-STaR: Learning to interleave thinking and proving. In *ICLR*, 2025.

[Microsoft, 2024] Microsoft. Loop invariant generation with LLMs, 2024. https://github.com/microsoft/loop-invariant-gen-experiments.

[Mukherjee *et al.*, 2015] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware verification using software analyzers. In *VLSI*. IEEE, 2015.

[Mukherjee *et al.*, 2016] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. v2c – A Verilog to C translator. In *TACAS*. Springer, 2016.

[Orenes-Vera *et al.*, 2023] Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using LLMs to facilitate formal verification of RTL. *CoRR*, 2309.09437, 2023.

[Pan *et al.*, 2024] Leyan Pan, Vijay Ganesh, Jacob D. Abernethy, Chris Esposo, and Wenke Lee. Can transformers reason logically? A study in SAT solving. *CoRR*, 2410.07432, 2024.

[Peled *et al.*, 2026a] Romy Peled, Daniel Kroening, Michael Tautschnig, and Yakir Vizel. Large lemma miners: Can LLMs do induction proofs for hardware? *CoRR*, 2511.02521, 2026.

[Peled *et al.*, 2026b] Romy Peled, Michael Tautschnig, Daniel Kroening, and Yakir Vizel. Large lemma miners. https://github.com/TechnionFV/large_lemma_miners, 2026.

[Petrov *et al.*, 2025] Ivo Petrov, Jasper Dekoninck, Lyuben Baltadzhiev, Maria Drencheva, Kristian Minchev, Mislav Balunovic, Nikola Jovanovic, and Martin T. Vechev. Proof or bluff? Evaluating LLMs on 2025 USA Math Olympiad. *CoRR*, 2503.21934, 2025.

[Pozdnyakov, 2024] Alexey Pozdnyakov. Predicting root numbers with neural networks. *Int. J. Data Sci. Math. Sci.*, 2024.

[Qayyum *et al.*, 2024] Khushboo Qayyum, Muhammad Hassan, Sallar Ahmadi-Pour, Chandan Kumar Jha, and Rolf Drechsler. Late breaking results: LLM-assisted automated incremental proof generation for hardware verification. In *Design Automation Conference, DAC*, 2024.

[Reimers and Gurevych, 2019] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *EMNLP-IJCNLP*, 2019.

[Romera-Paredes *et al.*, 2024] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 2024.

[Sheeran *et al.*, 2000] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, 2000.

[Su *et al.*, 2025] Yuheng Su, Qiusong Yang, Yiwei Ci, Tianjun Bu, and Ziyu Huang. The rIC3 hardware model checker. In *Computer Aided Verification, CAV*. Springer, 2025.

[Sun *et al.*, 2025] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark W. Barrett, David L. Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. ClassInvGen: Class invariant synthesis using large language models. *CoRR*, 2502.18917, 2025.

[Tafjord *et al.*, 2021] Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In *ACL/IJCNLP*, 2021.

[Thakur *et al.*, 2024] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. VeriGen: A large language model for Verilog code generation. *ACM Trans. Design Autom. Electr. Syst.*, 2024.

[Vizel *et al.*, 2015] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, 2015.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

[Wu *et al.*, 2024] Haoze Wu, Clark W. Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *ICLR*. OpenReview.net, 2024.

[Yan *et al.*, 2025] Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs. In *ASPDAC*, 2025.

[Yang *et al.*, 2024a] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. VERT: verified equivalent Rust transpilation with few-shot learning. *CoRR*, 2404.18852, 2024.

[Yang *et al.*, 2024b] Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in AI. *CoRR*, 2412.16075, 2024.

[Yao *et al.*, 2023] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in Rust. *CoRR*, 2311.03739, 2023.

[Zhang *et al.*, 2025] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. Scalable, validated code translation of entire projects using large language models. *PLDI*, 2025.

[Zheng *et al.*, 2025] Tong Zheng, Lichang Chen, Simeng Han, R. Thomas McCoy, and Heng Huang. Learning to reason via mixture-of-thought for logical reasoning. *CoRR*, 2505.15817, 2025.