Towards Language Model Guided TLA$^+$ Proof Automation

Yuhao Zhou(✉) and Stavros Tripakis

Northeastern University, Boston, MA, USA
zhou.yuhao@northeastern.edu

Abstract. Formal theorem proving with TLA$^+$ provides rigorous guarantees for system specifications, but constructing proofs requires substantial expertise and effort. While large language models have shown promise in automating proofs for tactic-based theorem provers like Lean, applying these approaches directly to TLA$^+$ faces significant challenges due to the hierarchical proof structure of the TLA$^+$ proof system. We present a prompt-based approach that leverages LLMs to guide hierarchical decomposition of complex proof obligations into simpler sub-claims, while relying on symbolic provers for verification. Our key insight is to constrain LLMs to generate normalized claim decompositions rather than complete proofs, significantly reducing syntax errors. We also introduce a benchmark suite of 119 theorems adapted from (1) established mathematical collections and (2) inductive proofs of distributed protocols. Our approach consistently outperforms baseline methods across the benchmark suite.

Keywords: Formal Verification · Theorem Proving · Large Language Models · TLA$^+$

1 Introduction

Formal verification plays a crucial role in ensuring the correctness of critical systems, particularly distributed systems where subtle errors can have severe consequences. As systems grow in complexity and become more interconnected, the need for rigorous verification methods becomes increasingly vital. The TLA$^+$ specification language [26] has emerged as a powerful framework for modeling and verifying such systems, with significant adoption in companies like Amazon, Intel, and Microsoft [5,20,35,36]. Despite its effectiveness, constructing formal proofs in TLA$^+$ remains time-consuming and requires substantial expertise, creating a bottleneck in the verification process [45,47].

Proof automation is fundamentally challenging: the underlying problem of proving theorems in expressive logics is undecidable [10,56] and state-of-the-art provers still require substantial human guidance for complex proofs [33,51]. Therefore, any progress in techniques that assist or automate proof construction

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16556, pp. 619–640, 2026.
https://doi.org/10.1007/978-3-032-26204-2_32

represents a significant opportunity to lower the barrier to formal verification, by making it more practical and scalable.

Recent advances in Large Language Models (LLMs) have shown promise in automating formal theorem proving tasks, particularly in *tactic-based* theorem provers like Lean [65] and Rocq (previously known as Coq) [50]. These approaches leverage LLMs' capabilities to generate sequences of proof tactics that incrementally transform proof states toward the goal. However, TLA$^+$ employs a fundamentally different, *hierarchical* proof structure. Unlike Lean and Rocq, which sequentially transform proof states by tactics, TLA$^+$ proofs are organized as trees of claims and sub-claims. For example, while a tactic-based proof consists of a sequence of transformations (e.g., 'expand definition, apply distributive property, simplify'), a TLA$^+$ proof introduces intermediate claims that collectively establish the goal. This distinction is illustrated by the proofs of theorem T1 in Fig. 1 (TLA$^+$) and Fig. 2 (Lean).

```
1  -------- MODULE sums_even -------
2  EXTENDS Naturals, TLAPS
3
4  Even(x) == x % 2 = 0
5
6  THEOREM L0 == ASSUME NEW x ∈ Nat PROVE
       Even(x + x) = Even(x * 2)
7  OBVIOUS
8
9  THEOREM L1 == ASSUME NEW x ∈ Nat PROVE
       Even(x * 2) = ((x * 2) % 2 = 0)
10 BY DEF Even
11
12 THEOREM T1 == ASSUME NEW x ∈ Nat PROVE
       Even(x + x)
13 BY L0, L1 DEF Even
14 ======================
```

Fig. 1. Theorem T1 proven in TLA$^+$.

```
1  import Mathlib.Tactic.Ring
2
3  def even (x : Nat) : Prop := x % 2 = 0
4
5  theorem T1 : ∀ x : Nat, even (x+x) := by
6      intro x
7      ring_nf
8      dsimp [even]
9      simp
```

Fig. 2. The same theorem T1 of Fig. 1, now proven in Lean. The Lean proof consists of a sequence of *tactics* (lines 6–9) that transform the proof state to solve the goal.

Additionally, while systems like Lean and Rocq have extensive libraries of formalized proofs that can serve as training data and benchmarks for machine learning approaches, TLA$^+$ lacks comparable datasets, creating a significant challenge for developing and evaluating learning-based proof automation.

In this paper, we present a language-model based approach to automating TLA$^+$ proof generation. Our method, called *Language Model Guided Proof Automation* (LMGPA), accommodates the hierarchical structure of TLA$^+$ proofs through a recursive decomposition strategy. This approach guides LLMs to recursively break down complex claims into simpler sub-claims that can be independently verified, mirroring the natural structure of TLA$^+$ proofs. Our system verifies each decomposition step, providing feedback to the LLM when necessary, and recursively applies the same process to each sub-claim until all claims can be verified by backend provers.

2 Preliminaries and Problem Statement

TLA$^+$ and TLA$^+$ Proof System. TLA$^+$ is a formal specification language [26] designed for specifying and verifying properties of complex systems and algorithms, particularly distributed systems and concurrent algorithms. It has been adopted in both academia and industry, with companies such as Amazon, Microsoft, and Intel successfully applying it to verify critical systems and protocols [5,20,25,35]. TLA$^+$ is supported by the TLA$^+$ Foundation – see https://foundation.tlapl.us/.

As a language grounded in mathematical logic, TLA$^+$ enables not only precise specification but also rigorous verification through model checking [67] and theorem proving. While model checking is an essential formal verification method [4,11,12], in the industry it is typically used for finding error traces quickly, and for verifying correctness of finite-state systems or bounded instances of infinite-state systems. In this paper, we focus on formal theorem proving, which allows to prove correctness of unbounded/infinite systems. Theorem proving for TLA$^+$ is implemented in the TLA$^+$ Proof System (TLAPS) [8], which serves as a bridge between human-written specifications and automated verification tools. TLAPS translates TLA$^+$ specifications and proofs into forms supported by backend provers like Z3 [14], Zenon [6], and Isabelle [37,40].

The proving approach in TLA$^+$ represents a distinct paradigm when compared to other prominent formal theorem provers, particularly in how proofs are structured and developed. In what follows, we discuss the most important differences.

TLAPS vs Tactic-based Interactive Theorem Provers. In the landscape of formal theorem proving, many popular *Interactive Theorem Provers* (ITPs) such as Lean [33], Rocq [51], and Isabelle/HOL [37] support a tactic-based approach to proof construction.[1] In the tactical style of proving, machine-checkable formal proofs are expressed as sequences of *tactics*—commands that systematically transform the proof state. Users guide the proof development by iteratively applying these tactics, effectively directing the prover through the proving process. The Lean proof in Fig. 2 illustrates this: the proof of `T1` is a sequence of tactics (lines 6–9) like `intro`, `ring_nf`, and `simp` that manipulate and solve the proof goal.

The proof methodology in TLA$^+$, however, follows a fundamentally different structure. Rather than tactical transformations, TLA$^+$ proofs are organized hierarchically—users establish complex claims by identifying and introducing intermediate sub-claims. For instance, the TLA$^+$ proof in Fig. 1 demonstrates this structure, the explicit intermediate sub-claims `L0` and `L1` collectively establish the goal `T1`. This hierarchical proof continues growing until the entire proof

[1] Tactical proofs are not the only style these provers support. For example, Lean also supports calculational proofs. Isabelle also supports hierarchically structured proofs similar to TLAPS; in fact, Isabelle is one of the backend provers used by TLAPS.

```
1 -------- MODULE amc12a_2015_p10 -------
2 EXTENDS Integers, TLAPS
3
4 THEOREM Main ==
5     ∀ x, y ∈ Int: (0 < y) ∧ (y < x) ∧ (x
        + y + (x * y) = 80) ⇒ (x = 26)
6 ========================
```

Fig. 3. An example theorem represented in TLA$^+$ as input to our proof generation system.

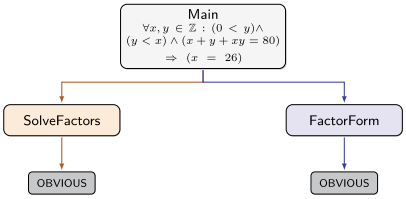

Fig. 4. Visualization of the proof tree for the proof in Fig. 5.

```
1  -------- MODULE amc12a_2015_p10 -------
2  EXTENDS Integers, TLAPS
3
4  THEOREM FactorForm ==
5      ASSUME  NEW x ∈ Int, NEW y ∈ Int,
6              0 < y, y < x,
7              x + y + (x * y) = 80
8      PROVE   (x + 1) * (y + 1) = 81
9  OBVIOUS
10
11 THEOREM SolveFactors ==
12     ASSUME  NEW x ∈ Int, NEW y ∈ Int,
13             0 < y, y < x,
14             (x + 1) * (y + 1) = 81
15     PROVE   x = 26
16 OBVIOUS
17
18 THEOREM Main ==
19     ∀ x, y ∈ Int: (0 < y) ∧ (y < x) ∧ (x
         + y + (x * y) = 80) ⇒ (x = 26)
20 BY FactorForm, SolveFactors
21 ========================
```

Fig. 5. Complete TLA$^+$ proof of the theorem in Fig. 3 (the entire proof was generated fully automatically by our system).

is directly machine-checkable by backend provers. This methodological distinction has significant implications for how proofs are developed, understood, and potentially automated within the TLAPS.

It is important to note that the proofs in Figs. 1 and 2 are presented purely for illustrative purposes to highlight this methodological difference. More direct or idiomatic proofs of T1 exist in both systems. While the theorem T1 is adapted from the TLA$^+$ example repository [54], the TLA$^+$ proof structure was intentionally modified to explicitly demonstrate the differences between hierarchical and tactic-based proving approaches.

TLA$^+$ Proof Structure. In TLA$^+$, a proof is a hierarchical arrangement of claims, where each claim represents a theorem to prove. To illustrate this structure, we refer to the examples in Figs. 3 and 5, which demonstrate a theorem and its corresponding proof in TLA$^+$. Using these examples as reference points, we now define the key terminology used throughout this paper:

- A *claim* is a boolean-valued expression written in TLA$^+$. For instance, line 5 in Fig. 3 (which is identical to line 19 in Fig. 5) is a claim. Lines 5–8 of Fig. 5 collectively form another claim.
- A *goal* is a specific claim that requires proof, representing the theorem or lemma of interest. In our example, the Main claim serves as the goal.
- *Context* is the collection of definitions and assumptions that provide the logical foundation for the goal. This includes imported modules such as the Integers module in Fig. 3, which provides the definition of Int.
- A *proof obligation* is a tuple of a context and a goal.

– A core construct in TLA$^+$ proofs is the ASSUME-PROVE structure, as seen in FactorForm and SolveFactors in Fig. 5. These are interpreted as logical implications where ASSUME F PROVE G means $\vdash F \Rightarrow G$, i.e., prove that F implies G.

While TLA$^+$ offers a rich and expressive proof language with multiple approaches to establishing claims, this paper focuses on a specific subset of proof structures for clarity and tractability. Specifically, we consider proofs that follow the pattern demonstrated in Fig. 5, where a claim may be associated with one of the following:

– An *auto proof*, where the claim can be directly verified by backend provers. In TLAPS, auto proofs use either the keyword OBVIOUS or the form BY DEF with a list of definitions to unfold. For example, the proofs of theorems FactorForm and SolveFactors in Fig. 5 use OBVIOUS, meaning they can be verified directly without unfolding any definitions. On the other hand, the proof of theorem Main is not an auto proof because it references other theorems.
– A proof by sub-claims, where the parent claim is established by a set of sub-claims. In TLAPS, this is expressed using the BY keyword followed by a list of the sub-claims. In Fig. 5, the parent claim Main is supported by two sub-claims: FactorForm and SolveFactors, which together provide a justification. Formally, if the parent claim asserts $F \Rightarrow G$, and it is supported by sub-claims A and B, then we are submitting to the solver the proof obligation $(A \wedge B) \Rightarrow (F \Rightarrow G)$, which is logically equivalent to $(A \wedge B \wedge F) \Rightarrow G$.
– No attached proof, as seen in Fig. 3 where the claim Main stands with no proof provided.

An important aspect of TLA$^+$ proof development, which is central to our work, is that appropriate sub-claims must be discovered to establish the parent claim. In Fig. 5, the sub-claims FactorForm and SolveFactors were not given in the original theorem statement (Fig. 3). They had to be formulated by the user with knowledge of quadratic equations. This discovery of effective intermediate steps represents a significant challenge in proof development. Human users must manually determine these sub-claims through mathematical insight and domain knowledge. Our system, however, attempts to automatically discover appropriate sub-claims.

TLA$^+$ provides several *proof directives* that instruct backend provers on how to prove the claims. These include OBVIOUS (indicating that the backend provers should verify the claim directly, as seen in line 9 of Fig. 5), BY (which proves the claim using specified facts and definitions, as demonstrated in line 20), BY SMT (restricting verification to only SMT solvers), and so on. These directives serve as an interface between the high-level proof structure and the specialized reasoning capabilities of various backend provers.

The status of a claim—whether it is considered *proved* or *unproved*—follows a recursive definition that reflects the hierarchical nature of TLA$^+$ proofs:

– A claim is *proved* if either:

  - It has an attached auto proof that is accepted by the backend provers, or
  - It is supported by a set of sub-claims that are themselves all *proved*, and the backend provers confirm that these sub-claims collectively establish the parent claim.
- Any claim not meeting these criteria remains *unproved*.

This hierarchical structure naturally gives rise to a tree representation of proofs, as visualized in Fig. 4 for the proof shown in Fig. 5. In this tree:

- Each node corresponds to a claim (`Main`, `FactorForm`, and `SolveFactors` in our example).
- The root node represents the primary goal (`Main` in our example).
- The edges capture the logical dependencies between claims, showing how sub-claims support their parent claims.

A proof achieves the status of a *complete proof* precisely when its root goal is *proved* according to the recursive definition above. This completion signifies that the entire proof has been successfully checked by backend solvers.

As constructing formal proofs manually requires significant expertise in both the problem domain and the formal prover itself, there exists a substantial barrier to the wider adoption of formal proving. Building on the framework outlined above, the central challenge addressed in this paper is the automated generation of complete proofs for TLA$^+$ proof obligations.

Problem Statement. Given a module containing an *unproved* claim (as in Fig. 3 where `Main` lacks a proof), our objective is to automatically construct a *complete proof* (like the one shown in Fig. 5).

3 Language Model Guided Proof Automation

Automated proof generation for TLA$^+$ presents unique challenges due to its hierarchical proof structure and rigorous verification requirements. In this section, we first describe the main challenges that naive methods face. We then introduce our approach, which leverages the reasoning capabilities of Large Language Models (LLMs) while addressing their limitations through a recursive claim decomposition strategy.

3.1 Challenges of Naive Methods

We consider two "naive" methods: (1) a symbolic method which simply attempts to use `TLAPS` to automatically prove the theorem; (2) an LLM-based method which prompts the LLM asking for a proof, up to a maximum of k times (this method is actually less naive when $k > 1$, as it uses feedback in subsequent prompts). We discuss each of these two naive methods next.

Naive Symbolic Method: `TLAPS OBVIOUS` The basic approach to automatically proving TLA$^+$ claims is to delegate them directly to `TLAPS`'s backend provers

Algorithm 1 Direct LLM-Based Proof Generation

```
1: function DirectLLM-ProveObligation(context, goal)
2:     feedback ← null
3:     repeat
4:         proof ← LLMGenProof(context, goal, feedback)
5:         proved, feedback ← VerifyByTLAPS(proof)
6:     until proved or max retries reached
7:     return proved, proof
8: end function
```

without providing further information. In the TLA$^+$ proof language, this is done by asserting the claim as `OBVIOUS`. While this method works for simple claims, it often fails for more complex ones that require intermediate proof steps to be explicitly provided.

Direct LLM-Based Proof Generation. A straightforward approach to automated TLA$^+$ proof generation involves prompting LLMs to generate complete proofs in a single prompt. This method relies entirely on the LLM's ability to produce syntactically correct and logically sound proofs from the provided theorem statements and context.

Algorithm 1 outlines this direct approach. For a given proof obligation (*context*, *goal*), the algorithm repeatedly prompts the LLM to generate a complete proof (Line 4). After each generation, it verifies the proof using `TLAPS` (Line 5). If the proof is valid, the process terminates; otherwise, the algorithm incorporates feedback from the verification step (e.g., error messages) into subsequent prompts to guide the LLM's next attempt (Line 4). This loop continues until a valid proof is found or the maximum number of retries is reached.
Generating TLA$^+$ proofs directly presents several challenges:

- **Syntactic Correctness:** Our experimental results (Sect. 4.4) demonstrate that state-of-the-art general-purpose LLMs, including OpenAI o3-mini [39] and Google Gemini [18], frequently generate TLA$^+$ proofs containing syntax errors, even when provided with the prover's feedback. These errors prevent programmatic verification by `TLAPS`.
- **Monolithic Generation:** When generating complete proofs from a single prompt, LLMs may introduce errors at any point in the proof. Because verification occurs only after the entire proof is generated rather than after each individual step, early errors propagate through subsequent reasoning. This lack of incremental verification limits LLMs' ability to maintain sound reasoning throughout multi-step proofs.

Recent approaches such as ReProver [65] and COPRA [50] address similar challenges in tactic-based theorem provers by constraining LLMs to generate only tactics and premises for a given proof state, enabling step-by-step verification and eliminating syntactic correctness issues. However, as discussed in

Sect. 2, the proof structure and methodology of TLA$^+$ differ fundamentally from tactic-based provers, necessitating a specialized approach aligned with TLA$^+$ proof methodology. Recent studies [60,64] have demonstrated promising results with single-pass Lean proof generation by fine-tuning LLMs on extensive Lean proof corpora, but again are not applicable to TLA$^+$. In this paper, we propose a *hierarchical proof generation* approach tailored to TLA$^+$'s proof methodology.

3.2 System Architecture and Key Ideas

Our Language Model Guided Proof Automation system (LMGPA) leverages the complementary strengths of LLMs and symbolic methods: we use LLMs for their reasoning abilities to decompose complex claims into simpler sub-claims, while relying on symbolic provers for rigorous verification and for proving simple claims. The key components include:

- **Claim Decomposition:** LLMs guide the decomposition of complex goals into simpler, more manageable sub-claims.
- **Automated Proof Generation:** For sufficiently simple claims, the system attempts to generate *auto proofs* using TLA$^+$ directives (e.g., OBVIOUS) that can be directly verified by TLAPS.
- **Proof Validation:** The system uses TLAPS to verify that (1) sub-claims collectively establish their parent claim, and (2) auto proofs are valid.

Our hierarchical, recursive proof generation algorithm (detailed in Sect. 3.3) directly addresses the two challenges identified above. First, it mitigates *syntactic correctness* issues by (1) restricting LLMs to generating only claim decompositions rather than complete proofs, and (2) normalizing LLM-generated sub-claim structures (Sect. 3.4), which significantly reduces opportunities for syntax errors (Sect. 4.4). Second, it overcomes *monolithic generation* limitations through incremental verification at each recursive step, enabling localized error correction without discarding entire proof attempts.

3.3 Recursive Proof Generation Algorithm

Algorithm 2 presents our hierarchical proof generation approach. The algorithm recursively decomposes complex claims until reaching claims that can be directly verified by the backend provers, mirroring the hierarchical structure of TLA$^+$ proofs described in Sect. 2.

For a given proof obligation ($context, goal$), the algorithm first attempts to generate an auto proof (Line 2). If this proof is successfully verified (Line 3), the algorithm returns it immediately. Otherwise, it leverages LLMs to decompose the goal into simpler sub-claims (Line 7) and verifies that these sub-claims collectively establish the original goal (Line 8). This verification-feedback loop continues until either a valid decomposition is found or the maximum number of retries is reached.

Once a valid decomposition is established, the algorithm recursively applies itself to each sub-claim (Lines 14–20), constructing a hierarchical proof structure consistent with TLA$^+$ proof conventions. If `ProveObligation` fails for any sub-claim (Lines 16–18), the entire proof attempt fails and the algorithm terminates without returning a valid proof, since all sub-claims must be discharged for the overall proof to be valid.

This recursive approach effectively combining the strengths of both symbolic provers (for rigorous verification) and LLMs (for non-trivial claim decomposition) to automate TLA$^+$ proof generation.

The final proof structure is assembled in Line 21, creating a complete TLA$^+$ proof that follows the hierarchical structure, with sub-claims serving as lemmas that collectively establish the parent claim.

Algorithm 2 Hierarchical Proof Generation

```
1: function ProveObligation(context, goal)
2:     autoProof ← GenerateAutoProof(context, goal)
3:     if VerifyProof(autoProof) then
4:         return autoProof
5:     end if
6:     repeat
7:         subClaims ← DecomposeIntoSubclaims(context, goal)
8:         decompositionValid ← VerifyDecomp(context, goal, subClaims)
9:     until decompositionValid or max retries reached
10:    if not decompositionValid then
11:        return failure
12:    end if
13:    proofs ← ∅
14:    for all claim ∈ subClaims do
15:        proof ← ProveObligation(context, claim)
16:        if proof is failure then
17:            return failure
18:        end if
19:        proofs ← proofs ∪ {(claim, proof)}
20:    end for
21:    return ConstructHierarchicalProof(goal, subClaims, proofs)
22: end function
```

In the following subsections, we delve deeper into the key components of our system, including LLM-guided claim decomposition, auto proof generation, and verification procedures. While developing this system, we explored various optimization techniques beyond those presented here. We focus on methods that demonstrated meaningful improvements in our experimental evaluation, while additional optimizations that did not yield significant benefits are documented in the Appendix of the extended version [73].

3.4 LLM-Guided Claim Decomposition

The `DecomposeIntoSubclaims` function forms the core of our approach, utilizing pretrained LLMs such as Claude [2] and o3-mini [39] to identify appropriate intermediate sub-claims that collectively establish a complex parent claim within the hierarchical proof structure. Notably, we use these models without any fine-tuning, relying instead on specialized prompting strategies to guide their reasoning toward valid claim decompositions.

To effectively leverage these models for claim decomposition and to overcome syntactic correctness challenges, we prompt the LLMs to generate normalized sub-claims that adhere to a specific structure (the complete prompt template is available in the Appendix of the extended version [73]).

Normalized Claim Structure. We constrain the generated sub-claims to follow a normalized format:

- Each LLM-generated sub-claim consists of a structured list containing assumptions (boolean expressions or definition references) and a single goal.
- Grammar constraints for expressions are embedded in the prompts, restricting output to ASCII characters and providing a table of acceptable notation.
- Our system parses these normalized claims and converts them into valid `TLA`$^+$ `ASSUME-PROVE` statements, eliminating a significant source of syntax errors.

Adaptive Feedback Loop. Although normalization substantially reduces syntax errors, complete correctness cannot be guaranteed. When `TLAPS` verification fails, our system feeds back the verifier's output to the LLM, allowing it to generate improved sub-claims in subsequent attempts.

3.5 Symbolic Auto Proof Generation

The `GenerateAutoProof` function employs a heuristic approach to efficiently handle simple claims without querying LLMs. This function first analyzes the parse tree of the given proof module to identify any definitions in the context that need to be explicitly unfolded in the goal claim. Based on this analysis, it generates appropriate auto proofs. If no definitions need unfolding, it applies the `TLA`$^+$ directive `OBVIOUS`, instructing backend provers to attempt verification directly. When the system determines that specific definitions must be unfolded to complete a proof, it generates a proof using the `TLA`$^+$ directive `BY` $l_1, l_2, \ldots$ `DEF` $d_1, d_2, \ldots$, where $l_1, l_2, \ldots$ are the assumptions and $d_1, d_2, \ldots$ are the definitions to be unfolded, both identified through syntax analysis.[2]

[2] We also explored a retrieval augmented [28] proof generation strategy but our preliminary results did not show sufficient improvements (c.f. the Appendix of the extended version [73]).

3.6 Verification Procedures

Our system includes two key verification procedures that ensure the correctness of both auto proofs and claim decompositions:

Auto Proof Verification. Function `VerifyProof` directly invokes `TLAPS` to determine whether a generated auto proof (e.g., `OBVIOUS`) is sufficient to justify a claim.

Decomposition Verification. Function `VerifyDecomp` validates that a set of sub-claims collectively establishes their parent claim. The system constructs a `TLA`$^+$ module that includes all sub-claims and the parent claim. `TLAPS` then verifies that the sub-claims collectively establish the parent claim.

4 Implementation and Evaluation

We present the implementation details of our system and evaluate its performance on the benchmark suite described in Sect. 4.2.

4.1 Implementation

We implemented the LMGPA system in Python 3.12, with components for parsing, verification, and LLM interactions. For syntax-level analysis in the auto proof generation phase (Sect. 3.5), we utilized the Tree-sitter `TLA`$^+$ parser [53], which enables efficient analysis of `TLA`$^+$ parse trees. LLM interactions are managed through LangChain [27], providing a unified interface to different language models.

The core verification pipeline integrates the `TLAPS` binary [55], with wrapper functions that handle the generation of temporary proof modules, execution of verification commands, and parsing of verification results. Our prompt templates include detailed instructions on the normalized claim format, examples of correct decompositions, and specific guidance on `TLA`$^+$ syntax constraints. Prompt templates are provided in the Appendix of the extended version of this paper [73]. The source code is available on GitHub [72] and the artifact on Zenodo [74].

4.2 Benchmarks

To evaluate our LMGPA system, we constructed a benchmark suite of `TLA`$^+$ theorems drawn from diverse sources to ensure variety in theorem types. The benchmark suite consists of: (a) 93 mathematical theorems adapted from the `miniF2F` [71] and `ProofNet` [3] collections; plus (b) 26 inductiveness proofs of candidate inductive invariants of distributed protocols, taken from [46] and its code repository [44]. `miniF2F` and `ProofNet` are standard benchmarks for evaluating AI-powered formal proof generation [65]. As these collections lack `TLA`$^+$ formalizations, we manually translated a curated subset of these theorems into

TLA$^+$ (for details, see the Appendix of the extended version of this paper [73]). Both the benchmark suite and our tool are publicly available, as an artifact [74] as well as in a GitHub repository [72].

We acknowledge the potential for data leakage in our benchmark. The mathematical theorems from miniF2F and ProofNet are commonly used benchmarks, and while our TLA$^+$ translations are new, the underlying proof strategies may have been encountered by the LLMs during pretraining in other formal languages such as Lean. Similarly, the inductiveness proofs from [46] are publicly available and could appear in pretraining corpora. Addressing potential data leakage issues can be done by constructing a leakage-free TLA$^+$ proof benchmark suite; this is beyond the scope of the current paper and part of future work.

4.3 Experimental Setup

We evaluated LMGPA on the benchmarks of Sect. 4.2. For our experiments, we selected state-of-the-art LLMs: Claude 3.7 Sonnet [2], Deepseek-V3.2-Exp [15], Gemini 2.0 Flash [18], Gemini 2.5 Flash [19], o3-mini-high [39], and GPT-5 [38]. This selection provides a diverse range of models, including both general language models (Claude and Gemini) and models optimized for reasoning tasks (o3-mini-high), as well as both open-source and proprietary models. We used all language models without any fine-tuning or additional training, relying solely on prompting strategies to guide these pretrained models.

For all experiments, we set consistent parameters across all models. We limited each model to a maximum of 4 decomposition attempts per claim (Algorithm 2, Line 9) and a maximum of 4 retries per proof obligation for direct LLM proof generation (Algorithm 1, Line 6). To ensure fair comparison and obtain results that are as deterministic as possible, we set the temperature to 0 for all LLM calls. All LLM requests were sent to the API provided by the LLM providers.

All experiments ran on a computer with a 16-core CPU and 64 GB RAM. We configured `TLAPS` to use 16 worker processes to fully utilize the available CPU capacity. We adhere to the default `TLAPS` timeouts. For each proof obligation, `TLAPS` attempts three backend provers in sequence: Z3 (5 s), Zenon (10 s), and Isabelle (30 s), resulting in a maximum total timeout of 45 s per obligation [1].

We evaluated our LMGPA system against the following baselines:

- Naive symbolic method (`TLAPS OBVIOUS`): see Sect. 3.1.
- Symbolic Auto Proof Generation (SAPG): see Sect. 3.5.
- Direct LLM Proof Generation (DLPG): see Sect. 3.1.

4.4 Results

Our evaluation focuses on three metrics: (1) the percentage of theorems successfully proved, which is our primary effectiveness measure, (2) the total number of LLM queries, and (3) the total time taken to process the entire benchmark suite.

Table 1. Evaluation results on the distributed protocol and mathematical benchmarks (c.f. Section 4.2). **Proved** is the percentage of theorems proved. **#Q** is the total number of queries made to the LLM. **Time** is the total execution time. The best result in each category is highlighted in **bold**.

	Distributed protocols			Mathematical		
Method	**Proved**	**#Q**	**Time**	**Proved**	**#Q**	**Time**
`TLAPS OBVIOUS`	0.0%	none	1m	44.1%	none	25m
SAPG	38.5%	none	14m	49.5%	none	34m
DLPG[Claude-3.7-Sonnet]	**15.4%**	98	4h 43m	17.2%	321	5h 14m
DLPG[Deepseek-V3.2-Exp]	0.0%	104	11h 27m	**29.0%**	336	39h 51m
DLPG[Gemini-2.0-Flash]	0.0%	104	41m	4.3%	359	39m
DLPG[Gemini-2.5-Flash]	0.0%	104	1h 30m	3.2%	364	3h 18m
DLPG[GPT-5]	3.8%	104	6h 36m	20.7%	307	24h 23m
DLPG[o3-mini-high]	0.0%	104	1h 5m	0.0%	372	8h 30m
LMGPA[Claude-3.7-Sonnet]	42.3%	83	1h 32m	53.8%	231	4h 49m
LMGPA[Deepseek-V3.2-Exp]	**50.0%**	73	9h 14m	**59.1%**	261	33h 17m
LMGPA[Gemini-2.0-Flash]	38.5%	54	39m	54.8%	251	1h 10m
LMGPA[Gemini-2.5-Flash]	46.2%	72	1h 25m	54.8%	224	5h 2m
LMGPA[GPT-5]	42.3%	89	4h 26m	58.1%	245	9h 58m
LMGPA[o3-mini-high]	42.3%	96	1h 44m	57.0%	241	5h 50m

The results are shown in Table 1. Across all benchmarks and all of the tested LLMs, our LMGPA system demonstrates consistent improvements in proof success rates compared to the baselines. The SAPG baseline itself consistently outperforms the `OBVIOUS`-only baseline, demonstrating the effectiveness of our heuristic-based symbolic proof generation component.

Timing Considerations. The timing differences are heavily influenced by factors unrelated to the models' capabilities. Network conditions, model architecture, caching strategies, and the hardware infrastructure of different model providers all significantly impact execution times—making raw timing comparisons between models less meaningful for evaluating proof generation effectiveness. The total time for evaluating the entire benchmark suite is thus reported for completeness.

Comparison with Combined Baselines. To further understand our approach, we compared it against a combined baseline that represents the best performance achievable by either baseline independently: see Table 2. Specifically, we consider a theorem as proved by the combined *SAPG+DLPG* baseline if either the SAPG or the DLPG successfully proves it. We also define a *total combined* approach,

which considers a theorem as proved if it is successfully proved by any of the three methods, SAPG, or DLPG, or LMGPA.

Table 2. Comparison between combined baseline and our system

Model	Proved			
	SAPG+DLPG	*LMGPA*	*Total Combined*	
Claude-3.7-Sonnet	52.9%	51.3%	54.6%	
Deepseek-V3.2-Exp	52.9%	57.1%	61.3%	
Gemini-2.0-Flash	49.6%	51.3%	52.9%	
Gemini-2.5-Flash	49.6%	52.9%	55.5%	
GPT-5	54.6%	54.6%	62.2%	
o3-mini-high	47.1%	53.8%	53.8%	

Syntax Errors in LLM-Generated Content. We also analyzed the syntactic validity of the content generated by LLMs in both DLPG and LMGPA systems. Because the generation targets differ, we aligned our evaluation with the specific output of each LLM query. For DLPG, we evaluated the full proofs, whereas for LMGPA, we evaluated the decompositions (sub-claims). We focused on decompositions for LMGPA because other proof structures are generated symbolically; thus, checking the decomposition isolates the LLM's actual contribution. Table 3 shows that while DLPG suffers from low syntactic validity, LMGPA achieves significantly higher syntactic validity rates. These results suggest that most of DLPG's failures stem from syntax errors, which aligns with our motivation for using normalized claim structure.

Table 3. Comparison of Syntactic Validity of LLM-Generated Proofs Between Approaches

Model	*DLPG*		*LMGPA*	
	Syn. Valid/#Queries	Percentage	Syn. Valid/#Queries	Percentage
Claude-3.7-Sonnet	88/434	20.3%	206/314	65.6%
Deepseek-V3.2-Exp	84/425	19.8%	287/334	85.9%
Gemini-2.0-Flash	27/463	5.8%	195/305	63.9%
Gemini-2.5-Flash	4/468	0.9%	228/296	77.0%
GPT-5	66/411	16.1%	290/334	86.8%
o3-mini-high	1/476	0.2%	252/337	74.8%

```
---- MODULE exercise_1_27 ----
EXTENDS Integers, TLAPS

Cube(x) == x * x * x

THEOREM L1 == ∃ x, y, z, w ∈ Int : Cube(x) + Cube(y) = 1729 ∧ Cube(z) + Cube(w) = 1729 ∧
      x ≠ z ∧ x ≠ w ∧ y ≠ z ∧ y ≠ w
THEOREM L2 == ∀ n ∈ Nat : (n < 1729) ⇒ ¬(∃ x, y, z, w ∈ Int : Cube(x) + Cube(y) = n ∧
     Cube(z) + Cube(w) = n ∧ x ≠ z ∧ x ≠ w ∧ y ≠ z ∧ y ≠ w)
THEOREM exercise_18_4 == ∀ n ∈ Nat : (∃ x, y, z, w ∈ Int : Cube(x) + Cube(y) = n
      ∧ Cube(z) + Cube(w) = n ∧ x ≠ z ∧ x ≠ w ∧ y ≠ z ∧ y ≠ w)  ⇒ n ≥ 1729
BY L1, L2 DEF Cube
=====================
```

Fig. 6. An example of TLAPS limitation. Theorem L2 is the contrapositive of the target theorem, making the implication (L1 ∧ L2 ⇒ exercise_18_4) trivially valid. However, TLAPS fails to prove this within the timeout.

Failures Due to Prover Limitations. Beyond syntax errors and logically incorrect decompositions, another failure mode in LMGPA arises when LLMs generate mathematically valid decompositions that TLAPS fails to verify automatically. Figure 6 shows an LLM-generated decomposition attempt of theorem exercise_18_4 into sub-claims L1 and L2. Observe that L2 is the contrapositive of the target theorem. Thus, the decomposition is logically valid: (L1 ∧ L2) ⇒ exercise_18_4 holds (the assumption L1 is redundant, but this does not affect the validity of the implication). However, TLAPS cannot prove this implication within the default timeout.

Quality of Generated Proofs. We qualitatively analyzed the proofs generated by LMGPA. The generated proofs are generally well-structured, following TLA$^+$ syntax with hierarchical organization of claims and sub-claims (e.g., Figs. 5 and 7). However, they do not yet match the quality of human-written proofs. Some proofs, while logically valid, do not correspond to the most concise and natural way to prove the theorem. For example, in the proof shown in Fig. 7, sub-claims CubeOf97_1 and SumIs16_4 are trivial calculations that are unnecessary in establishing the goal. We observed similar patterns elsewhere: decompositions that introduce more sub-claims than necessary or use intermediate claims uncommon in human proofs. Also, in some cases decompositions are not useful. For example, in Fig. 6, L2 is a restatement of the goal, so the LLM-generated decomposition does not meaningfully simplify the target theorem. These observations suggest room for improvement in generating more concise and natural proofs, as well as better decompositions. We nevertheless note that our system is capable of generating non-trivial proofs. For example, a distributed protocol inductiveness proof generated by our LMGPA system is 227 lines long (including 88 lines of definitions). This entire proof is listed in the Appendix of the extended version of this paper [73]. All proofs generated in our experiments can be found in the public repository [72] as well as in the artifact of this paper [74].

5 Related Work

LLM-assisted Theorem Proving. Recent years have seen significant advances in applying LLMs to formal reasoning tasks. In the domain of theorem proving, GPT-f [41] is a generative language model for automated theorem proving using Metamath [31]. Baldur [17] shows the LLMs' abilities on generating and repairing formal proofs in the Isabelle/HOL. Tahat et al. [49] present a case study on proof repair utilizing LLMs in Coq. LeanDojo [65] demonstrates the use of language models and retrieval-augmented generation for generating proof tactics and selecting premises in the Lean theorem prover, providing both tactical suggestions and a comprehensive benchmark suite for evaluating LLMs on formal proof tasks. COPRA [50] applies in-context learning to both Rocq and Lean provers, demonstrating how learning from existing examples can improve proof generation in these provers. Hilbert [57] leverages this idea and uses both general purpose and specialized LLMs for different levels of proof generation in Lean. Cobblestone [23] presents a divide-and-conquer approach for synthesizing Rocq proofs using off-the-shelf LLMs. Unlike these approaches, our method targets generating `TLA`$^+$ proofs.

Liang et al. [29] argue that general purpose LLMs perform well on high-level proof decomposition comparing to specialized models fine-tuned for theorem proving tasks. Zhang et al. [69] give a detailed analysis of LLMs' capabilities in formal theorem proving and proposes general suggestions to enhance their performance. Despite the successes in LLM-assisted reasoning, Mirzadeh et al. [32] discuss the limitations of LLMs in mathematical reasoning.

Fine-tuning approaches have also shown promise, with DeepSeek-Prover-V1.5 [64] and TheoremLlama [60] achieving notable results in single-pass Lean proof generation through specialized training on extensive Lean proof corpora. POETRY [58] generates Isabelle proofs recursively using a fine-tuned language model, whereas our approach leverages general-purpose LLMs to generate `TLA`$^+$ proofs. Rango [52] fine-tunes a language model to generate Rocq tactics step by step, augmenting the generation with retrieved similar proofs and relevant lemmas. Other approaches include LEGO-Prover [59], which employs a growing library of verified lemmas to augment LLMs' theorem proving capabilities, and work by Jiang et al. [22], which maps informal proofs to formal proof sketches that guide automated provers. DeepSeek-Prover-V2 [42] explores subgoal decomposition strategies via reinforcement learning to enhance formal reasoning capabilities for LLMs in Lean. Our work differs from these approaches by focusing on `TLA`$^+$, which employs a different proof structure.

These advances have been supported by standardized benchmarks such as miniF2F [71] and ProofNet [3], which provide diverse collections of mathematical problems for evaluating theorem provers across different formal systems.

LLMs for Software Verification. Beyond mathematical theorem proving, LLMs have shown promise in software verification tasks. Clover [48] leverages LLMs to generate Dafny code and annotations, while Chakraborty et al. [7] apply them to loop invariant generation. Wen et al. [62] combines LLMs with static analysis tools for program specification synthesis, and the Lemur system [63] demonstrates how LLMs can enhance traditional program verification frame-

```
---- MODULE mathd_numbertheory_234 ----
EXTENDS TLAPS, Integers

THEOREM CubeOf97_1 == 97 * 97 * 97 = 912673
OBVIOUS

THEOREM OnlySolution_2 ==
ASSUME NEW a ∈ Nat, NEW b ∈ Nat, a ≥ 1, a ≤ 9, b ≤ 9,
    (10 * a + b) * (10 * a + b) * (10 * a + b) = 912673
PROVE 10 * a + b = 97
OBVIOUS

THEOREM UniqueDigits_3 ==
ASSUME NEW a ∈ Nat, NEW b ∈ Nat, a ≥ 1, a ≤ 9, b ≤ 9, 10 * a + b = 97
PROVE a = 9 ∧ b = 7
OBVIOUS

THEOREM SumIs16_4 ==
ASSUME NEW a ∈ Nat, NEW b ∈ Nat, a = 9, b = 7
PROVE a + b = 16
OBVIOUS

THEOREM mathd_numbertheory_234 ==
∀ a, b ∈ Nat :
        (a ≥ 1) ∧ (a ≤ 9) ∧ (b ≤ 9) ∧
        ((10 * a + b) * (10 * a + b) * (10 * a + b) = 912673)
        ⇒ (a + b = 16)
BY CubeOf97_1, OnlySolution_2, UniqueDigits_3, SumIs16_4
==================================
```

Fig. 7. A valid but verbose proof generated by LMGPA: sub-claims `CubeOf97_1` and `SumIs16_4` are both trivial and redundant.

works. Laurel [34] provides a framework for using LLMs to generate and verify program specifications in Dafny and a benchmark extracted from real-world codebase. DafnyBench [30] provides a benchmark suite for evaluating LLMs in the context of Dafny program verification. Selene [68] proposes a benchmark for automated software verification, grounded in seL4 kernel [24].

Prompt Engineering and In-Context Learning. Research has explored LLMs' capabilities in general reasoning tasks [21,70] and the role of prompt engineering in formal methods applications [9,13]. Techniques such as in-context learning [16, 43] and dynamic prompt adjustment [61,66] have proven effective in improving LLMs' performance on tasks requiring precise logical reasoning.

6 Conclusion

We present a language model-guided approach for automating TLA$^+$ proof generation through hierarchical decomposition of complex proof obligations. Our key insight is that by constraining LLMs to generate normalized claim decompositions rather than complete proofs, we can leverage their reasoning capabilities while mitigating their tendency to produce syntactically incorrect formal proofs. Our evaluation shows substantial gains over direct LLM proof generation while highlighting the importance of combined LLM+symbolic tools.

Future work includes exploring specialized training methods, such as fine-tuning on TLA$^+$ proof corpora, to address persistent syntax errors, and to improve decomposition quality and the overall success rates. We also plan to investigate advanced prompting strategies and retrieval-augmented techniques. Another direction for future work is investigating how to guide LLMs to generate decompositions that are not only mathematically valid but also readily verifiable by automated provers. Training or guiding LLMs to understand the capabilities and limitations of symbolic provers could lead to more effective proof automation strategies.

Acknowledgments. We would like to thank William Schultz and Ian Dardik who provided the inductive invariant proofs that formed our distributed protocol benchmarks. This work has been supported by NSF's FMitF (Formal Methods in the Field) program, under awards 2319500 and 2525087.

References

1. TLA+ Proof System documentation: Tactics. https://proofs.tlapl.us/doc/web/content/Documentation/Tutorial/Tactics.html
2. Anthropic: Claude 3.7 Sonnet (2025). https://www.anthropic.com/news/claude-3-7-sonnet
3. Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E.W., Radev, D., Avigad, J.: ProofNet: autoformalizing and formally proving undergraduate-level mathematics. arXiv preprint arXiv:2302.12433 (2023)
4. Baier, C., Katoen, J.P.: Principles of Model Checking, MIT Press (2008)
5. Beers, R.: Pre-RTL formal verification: an Intel experience. In: Proceedings of the 45th ACM/IEEE Design Automation Conference, pp. 806–811 (2008)
6. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: an extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75560-9_13
7. Chakraborty, S.: Ranking LLM-generated loop invariants for program verification. arXiv preprint arXiv:2310.09342 (2023)
8. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^{\text{â}\check{\text{A}}\text{Ľ}+\text{â}\check{\text{A}}\text{Ľ}}$ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 142–148. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_12
9. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: transforming natural languages to temporal logics using large language models. arXiv preprint arXiv:2305.07766 (2023)
10. Church, A.: An unsolvable problem of elementary number theory. Am. J. Math. **58**(2), 345–363 (1936)
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking, MIT Press (2000)
12. Clarke, E.M., Henzinger, T.A., Veith, H.: Handbook of Model Checking. Presented at the (2018). https://doi.org/10.1007/978-3-319-10575-8
13. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: NL2SPEC: interactively translating unstructured natural language to temporal logics with large language models. In: International Conference on Computer Aided Verification, pp. 383–396. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_18

14. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. DeepSeek-AI: Introducing DeepSeek-V3.2-Exp (2025). https://api-docs.deepseek.com/news/news250929
16. Dong, Q.: A survey on in-context learning. arXiv preprint arXiv:2301.00234 (2022)
17. First, E., Rabe, M.N., Ringer, T., Brun, Y.: Baldur: Whole-proof generation and repair with large language models. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1229–1241 (2023)
18. Google DeepMind: Gemini 2.0 flash: a powerful workhorse model with low latency and enhanced performance (2025). https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash
19. Google DeepMind: Gemini 2.5 flash (2025). https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash
20. Hackett, F., Rowe, J., Kuppe, M.A.: Understanding inconsistency in azure cosmos DB with TLA. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 1–12. IEEE (2023)
21. Ho, N., Schmid, L., Yun, S.Y.: Large language models are reasoning teachers. arXiv preprint arXiv:2212.10071 (2022)
22. Jiang, A.Q.: Draft, sketch, and prove: guiding formal theorem provers with informal proofs. arXiv preprint arXiv:2210.12283 (2022)
23. Kasibatla, S.R., Agarwal, A., Brun, Y., Lerner, S., Ringer, T., First, E.: Cobblestone: a divide-and-conquer approach for automating formal verification. arXiv preprint arXiv:2410.19940 (2024)
24. Klein, G.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 207–220 (2009)
25. Konnov, I., Kuppe, M., Merz, S.: Specification and verification with the TLA+ trifecta: TLC, Apalache, and TLAPS. In: International Symposium on Leveraging Applications of Formal Methods, pp. 88–105. Springer (2022). https://doi.org/10.1007/978-3-031-19849-6_6
26. Lamport, L.: Specifying systems: the TLA+ language and tools for hardware and software engineers (2002)
27. LangChain-AI: LangChain (2022). https://github.com/langchain-ai/langchain
28. Lewis, P.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Advances in Neural Information Processing Systems, vol. 33, pp. 9459–9474 (2020)
29. Liang, Z.: Towards solving more challenging IMO problems via decoupled reasoning and proving. arXiv preprint arXiv:2507.06804 (2025)
30. Loughridge, C.: DafnyBench: a benchmark for formal software verification. arXiv preprint arXiv:2406.08467 (2024)
31. Megill, N., Wheeler, D.A.: Metamath: a computer language for mathematical proofs. Lulu. com, (2019)
32. Mirzadeh, I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., Farajtabar, M.: GSM-symbolic: understanding the limitations of mathematical reasoning in large language models. arXiv preprint arXiv:2410.05229 (2024)
33. Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 625–635. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_37
34. Mugnier, E., Gonzalez, E.A., Polikarpova, N., Jhala, R., Yuanyuan, Z.: Laurel: unblocking automated verification with large language models. Proc. ACM Program. Lang. **9**(OOPSLA1), 1519–1545 (2025)

35. Newcombe, C.: Why amazon chose TLA+. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp. 25–39. Springer (2014). https://doi.org/10.1007/978-3-662-43652-3_3
36. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9_5
38. OpenAI: OpenAI GPT-5 system card (2025). https://openai.com/index/gpt-5-system-card/
39. OpenAI: OpenAI o3-mini system card (2025). https://cdn.openai.com/o3-mini-system-card-feb10.pdf
40. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer (1994). https://doi.org/10.1007/bfb0030558
41. Polu, S., Sutskever, I.: Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393 (2020)
42. Ren, Z.: DeepSeek-Prover-V2: advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. arXiv preprint arXiv:2504.21801 (2025)
43. Rubin, O., Herzig, J., Berant, J.: Learning to retrieve prompts for in-context learning. arXiv preprint arXiv:2112.08633 (2021)
44. Schultz, W.: GitHub repository for plain and simple inductive invariant inference for distributed protocols in TLA+. https://github.com/will62794/endive
45. Schultz, W., Dardik, I., Tripakis, S.: Formal verification of a distributed dynamic reconfiguration protocol. In: CPP 2022, Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 143–152. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503688
46. Schultz, W., Dardik, I., Tripakis, S.: Plain and simple inductive invariant inference for distributed protocols in TLA. In: 2022 Formal Methods in Computer-Aided Design (FMCAD), pp. 273–283. IEEE (2022)
47. Schultz, W., Zhou, S., Dardik, I., Tripakis, S.: Design and analysis of a logless dynamic reconfiguration protocol. In: Bramas, Q., Gramoli, V., Milani, A. (eds.) 25th International Conference on Principles of Distributed Systems (OPODIS 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 217, pp. 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.OPODIS.2021.26, https://drops.dagstuhl.de/opus/volltexte/2022/15801
48. Sun, C., Sheng, Y., Padon, O., Barrett, C.: Clover: closed-loop verifiable code generation. arXiv preprint arXiv:2310.17807 (2023)
49. Tahat, A., Hardin, D., Petz, A., Alexander, P.: Proof repair utilizing large language models: a case study on the copland remote attestation proofbase. In: International Conference on Bridging the Gap between AI and Reality, pp. 145–166. Springer (2024). https://doi.org/10.1007/978-3-031-75434-0_10
50. Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., Chaudhuri, S.: An in-context learning agent for formal theorem-proving. In: First Conference on Language Modeling (2023)
51. The Rocq Development Team: The Rocq reference manual - release 8.19.0 (2025). https://rocq-prover.org/doc/V9.0.0/refman

52. Thompson, K., et al.: Rango: adaptive retrieval-augmented proving for automated software verification. In: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, pp. 347–359. IEEE (2025). https://doi.org/10.1109/ICSE55347.2025.00161
53. TLA+ Community: tree-sitter-tlaplus: TLA+ grammar for tree-sitter (2023). https://github.com/tlaplus-community/tree-sitter-tlaplus
54. TLA+ Foundation: Examples of TLA+ specifications (2025). https://github.com/tlaplus/Examples
55. TLA+ Foundation: The TLA+ proof system (2025). https://github.com/tlaplus/tlapm
56. Turing, A.M., et al.: On computable numbers, with an application to the Entscheidungsproblem. J. Math **58**(345–363), 5 (1936)
57. Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., Ye, K.: Hilbert: recursively building formal proofs with informal reasoning. arXiv preprint arXiv:2509.22819 (2025)
58. Wang, H.: Proving theorems recursively. In: Advances in Neural Information Processing Systems, vol. 37, pp. 86720–86748 (2024)
59. Wang, H.: LEGO-prover: neural theorem proving with growing libraries. arXiv preprint arXiv:2310.00656 (2023)
60. Wang, R.: TheoremLlama: transforming general-purpose LLMs into Lean4 experts. arXiv preprint arXiv:2407.03203 (2024)
61. Wei, J.: Chain-of-Thought prompting elicits reasoning in large language models. In: Advances in Neural Information Processing Systems, vol. 35, pp. 24824–24837 (2022)
62. Wen, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: International Conference on Computer Aided Verification, pp. 302–328. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_16
63. Wu, H., Barrett, C., Narodytska, N.: Lemur: integrating large language models in automated program verification. arXiv preprint arXiv:2310.04870 (2023)
64. Xin, H.: DeepSeek-Prover-V1.5: harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. arXiv preprint arXiv:2408.08152 (2024)
65. Yang, K.: LeanDojo: theorem proving with retrieval-augmented language models. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
66. Yao, S.: Tree of thoughts: deliberate problem solving with large language models. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
67. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6
68. Zhang, L., Lu, S., Duan, N.: Selene: pioneering automated proof in software verification. arXiv preprint arXiv:2401.07663 (2024)
69. Zhang, S.D., Ringer, T., First, E.: Getting more out of large language models for proofs. arXiv preprint arXiv:2305.04369 (2023)
70. Zhang, Y.: LLM as a mastermind: a survey of strategic reasoning with large language models. arXiv preprint arXiv:2404.01230 (2024)
71. Zheng, K., Han, J.M., Polu, S.: MiniF2F: a cross-system benchmark for formal olympiad-level mathematics. arXiv preprint arXiv:2109.00110 (2021)
72. Zhou, Y.: GitHub repository for language-model guided TLA+ proof automation (2026). https://github.com/YUH-Z/lmgpa

73. Zhou, Y., Tripakis, S.: Towards language model guided TLA+ proof automation. arXiv:2512.09758 (2025)
74. Zhou, Y., Tripakis, S.: Artifact for paper: language-model guided TLA+ proof automation (2026). https://doi.org/10.5281/zenodo.18637323