From Informal to Formal – Incorporating and Evaluating LLMs on Natural Language Requirements to Verifiable Formal Proofs

Jialun Cao[1], **Yaojie Lu**[2], **Meiziniu Li**[1], **Haoyang Ma**[1], **Haokun Li**[3], **Mengda He**[3], **Cheng Wen**[3], **Le Sun**[2], **Hongyu Zhang**[4], **Shengchao Qin**[3,5], **Shing-Chi Cheung**[1], **Cong Tian**[5]
[1] The Hong Kong University of Science and Technology
[2] Institute of Software, Chinese Academy of Sciences
[3] Guangzhou Institute of Technology, Xidian University
[4] Chongqing University [5] ICTT and ISN Laboratory, Xidian University
jcaoap@cse.ust.hk, luyaojie@iscas.ac.cn

Abstract

The research in AI-based formal mathematical reasoning has shown an unstoppable growth trend. These studies have excelled in mathematical competitions like IMO and have made significant progress. However, these studies intertwined multiple skills simultaneously—problem-solving, reasoning, and writing formal specifications—making it hard to precisely identify the LLMs' strengths and weaknesses in each task. This paper focuses on formal verification, an immediate application scenario of formal reasoning, and breaks it down into sub-tasks. We constructed 18k high-quality instruction-response pairs across five mainstream formal specification languages (Coq, Lean4, Dafny, ACSL, and TLA+) in six tasks by distilling gpt-4o and evaluated against ten open-sourced LLMs, including recent popular DeepSeek-R1. We found that LLMs are good at writing proof segments when given either the code, or the detailed description of proof steps. Also, the fine-tuning brought about a nearly threefold improvement at most. Interestingly, we observed that fine-tuning with formal data also enhances mathematics, reasoning, and coding capabilities. Fine-tuned models are released to facilitate subsequent studies at `https://huggingface.co/fm-universe`.

1 Introduction

> *The more we formalize, the more of our implicit knowledge becomes explicit.*
>
> — Terence Tao (Tao, 2024)

As AI-based formal mathematical reasoning reached an inflection point (Yang et al., 2024b), significant attention and progress in this field have been observed. AlphaProof (AlphaProof and Teams, 2024) achieved silver medal level in the International Mathematical Olympiad (IMO), AlphaGeometry (Trinh et al., 2024) specialized in proving Euclidean geometry theorems. As reported, the number of publications in this field nearly doubled in 2023, indicating an unstoppable growth trend (Li et al., 2024). As Fields Medalist Terence Tao imagined, "*In the future, instead of typing up our proofs, we would explain them to some GPT*" (Tao, 2024).

However, most current benchmarks cannot precisely reflect the capability to convert informal proofs or requirements in natural language into formal proofs. Most of these benchmarks take mathematical problems (AlphaProof and Teams, 2024; Trinh et al., 2024; Welleck et al., 2021) or theorems to be solved (Yang et al., 2023; Welleck et al., 2022; Yang and Deng, 2019) as input, and informal or formal proofs (or parts of proofs) as output. However, these end-to-end benchmarks assess multiple capabilities (*e.g.,* problem-solving, mathematical reasoning, formal specification writing) in an intertwined manner, making it difficult to isolate and observe LLMs' true capabilities in writing formal proofs or models for verification.

Therefore, we ***break down the process*** from informal requirements to formal verifiable proof, as shown in Figure 1. Inspired by the code generation (shown in blue) which translates a description of implementation into executable code (Chen et al., 2021; Austin et al., 2021a), the formal reasoning process (shown in green) can be seen as translating an ***informal requirement*** into ***a verifiable formal proof*** or checkable formal model [1]. Particularly, we decompose this process and formulate six tasks (Figure 2). By doing so, the intertwined capabilities can be separated and individually assessed, providing a clearer understanding of LLMs' strengths and weaknesses in each task.

Scope and Targets – We focus on ***formal verification*** (Appel, 2011; Klein et al., 2009; Leroy et al., 2016; Hawblitzel et al., 2014) because it is an immediate application scenario of formal mathe-

[1] For ease of expression, we generally refer to *verifiable formal proofs* and *checkable models* as "proofs" for the sake of presentation simplification.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 26984–27003
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

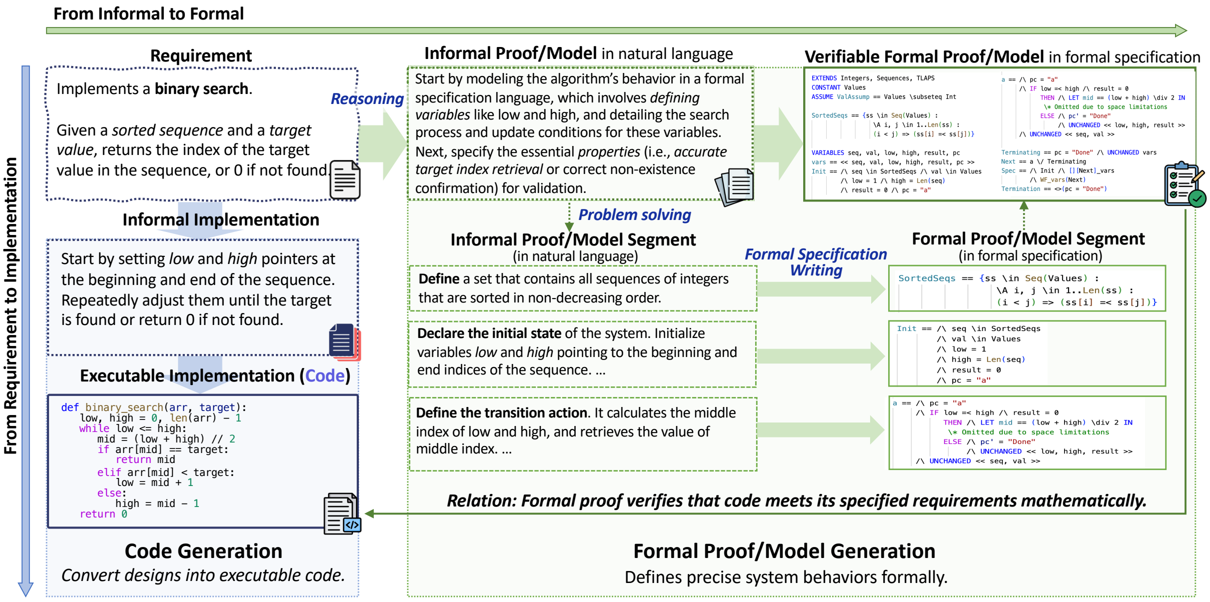

Figure 1: The Illustration of Formal Proof Generation and Its Relation with Code Generation.

matical reasoning and the correctness of the output can be verified mechanically. In this paper, we mainly explore four research questions (RQs):

RQ1. How well do LLMs perform in various formal verification tasks? After decomposing the formal verification task into subtasks, we explore LLMs' initial performance in these tasks with zero-shot and few-shot, investigating the strengths and weaknesses that vary between LLMs and tasks.

RQ2. Do LLMs show variability in their capability across different formal specification languages? When mathematicians and proof engineers consider using LLMs to assist in formal verification, they often face uncertainty about which formal specification language is best supported by LLMs. This RQ is designed to provide hints on it.

RQ3. Can fine-tuning improve LLMs' performance in formal verification? Although recent efforts have been made to fine-tune models (Wang et al., 2024; Yang et al., 2023), these LLMs are typically fine-tuned with single formal languages instead of multi-lingual (e.g., combining Coq, Lean, etc.) (Yang et al., 2024b). Therefore, we instruction fine-tuned (Wei et al., 2021; Sanh et al., 2021) three base LLMs to see whether our constructed fine-tuning dataset FM-ALPACA could improve their capability in formal verification tasks.

RQ4. Can fine-tuning with formal verification data benefit other related tasks (mathematics, reasoning, code)? As recent works have shown LLMs' potential transferability of skills (Tihanyi et al., 2023) we thus extend our study to see if models fine-tuned on formal data could show enhanced capabilities in mathematics, reasoning, and coding.

To facilitate the study, we constructed ***18k high-quality instruction-response pairs*** across five formal specification languages (*i.e.,* Coq, Lean4, Dafny, ACSL, and TLA+) in six formal-verification-related tasks by distilling gpt-4o inspired by prior work (Wang et al., 2024; Ding et al., 2023; Wang et al., 2022), then split them into 14k instruction fine-tuning data (FM-ALPACA) and 4k benchmarking data (FM-BENCH). In particular, we provide ***executable contexts*** for all these formal specifications and automated ***validation scripts*** to validate the correctness of the generated formal proofs inspired from the prior work's artifact preparation (Jimenez et al., 2023). Finally, we release the fine-tuned LLMs based on three base models at https://huggingface.co/fm-universe.

Interestingly, there has been recent discussion on the topic of domain transfer (Yang et al., 2024b), particularly the transfer of knowledge from other domains such as coding and reasoning to formal domains in order to increase LLMs' reliability (Spiess et al., 2024), and the anticipated potential of AI in enhancing formal verification processes to support mathematical proofs (Tao, 2024; Tao et al., 2023). Our experimental results could potentially provide empirical support for these hypotheses or offer directions for further experimental inquiries.

The contribution of this paper includes:

✔ ***Problem Formulation***: We decompose the formal verification process into six essential tasks. By doing so, the intertwined capabilities can be separated and individually assessed, providing a clearer understanding of LLMs' strengths and weaknesses in each task.

✔ ***Dataset and Benchmark***: We constructed 18k high-quality instruction-response pairs across five mainstream formal specification languages (*i.e.,* Coq, Lean4, Dafny, ACSL, and TLA+) in six formal-verification-related tasks by distilling gpt-4o. They are split into a 14k+ fine-tuning dataset FM-ALPACA and a 4k benchmark FM-BENCH.

✔ ***Executable context and automated validation mechanism***: We provide a Docker container equipped with necessary scripts to facilitate the evaluation of FM-BENCH, significantly lowering the entry barrier for this scenario and making subsequent contributions easier.

✔ ***Insight and Vision***: We fine-tuned several models on FM-ALPACA and observed promising benefits to not only the formal verification tasks, but also mathematics, reasoning, and coding. Our experimental results provide empirical support for the potential of LLMs' capability transfer and hope to shed some light on future research.

2 Task Formulation

Figure 2 illustrates the six sub-tasks. We elaborate on them in detail as follows.

Task 1. Requirement Analysis (abbrev. ReqAna). Requirement analysis (Davis, 1990; Anton, 1996; Grady, 2010; Jin, 2017) is a critical and long-standing research area in software engineering. It facilitates collecting, identifying, categorizing and modeling the users' needs and expectations using various techniques (Taggart Jr and Tharp, 1977; Deeptimahanti and Babar, 2009; Javed and Lin, 2021; Wang et al., 2021; Jin et al., 2024; Zhou et al., 2022). In this paper, the requirements are the descriptions in natural language (English) (Jin et al., 2024) that details the requirements of the verification/modeling goal and an overall description of the proofs/models. The task is to analyze and break down the final goal into detailed steps described in natural language. The natural language used in this paper is English.

Task 2. Full Proof Generation (abbrev. ProofGen). This task formalizes a requirement in natural language into verifiable proofs or models written in formal specification languages, similar task formulation to existing works (Fatwanto, 2012; Zhou et al., 2022; Davril et al., 2013).

Task 3. Proof Segment Generation (abbrev. SegGen). Unlike ProofGen, which requires generating complete proofs/models, SegGen provides more detailed descriptions in natural language and requires LLMs to write less. Given a text description articulating how to implement the proofs/modeling, the task outputs a segment written in the formal specification that serves as a component in the complete proof/model. This task formulation is similar to prior work (Wang et al., 2024; Wu et al., 2022; Jiang et al., 2022) and similar to the formulation of code generation (Chaudhary, 2023; Sun et al., 2024; Welleck et al., 2022, 2021).

Task 4. Proof Completion (abbrev. ProofComplete). Similar to code completion (Raychev et al., 2014; Husein et al., 2024; Svyatkovskiy et al., 2019; Dakhel et al., 2023), ProofComplete suggests the suffix of the given prefix, similar to prior work (Song et al., 2024). Note that in order to prevent LLMs from deviating from the original verification goal, we also provide the requirement in our evaluation, although it is not compulsory for this task formulation.

Task 5. Proof InFilling (abbrev. ProofInfill). Given a proof/model with a mask in the middle, the task requires LLMs to fill proper formal specifications so that the completed proofs/models can pass the verifier. This formulation is the same as code infilling (Fried et al., 2022). Also, similar to ProofComplete, we provide the requirement in our evaluation during the infilling to prevent LLMs from deviating from the original verification goal.

Task 6. Proof Generation from Code (abbrev. Code2Proof). In addition to generating formal specifications from natural languages, formal specifications can also be generated from code if the verification goal is the property of a given program. In this paper, we focus mainly on specifications in form of code annotations (Baudin et al., 2021; Hatcliff et al., 2012), expressing specifications (*e.g.,* pre-/post-condition, loop invariants) that help one to verify that (part of) a program satisfies certain properties. The task takes the code with properties to be verified as input and outputs the code with generated annotated formal specifications. Similar task formulation can be found in recent works (Wen et al., 2024; Ma et al., 2024).

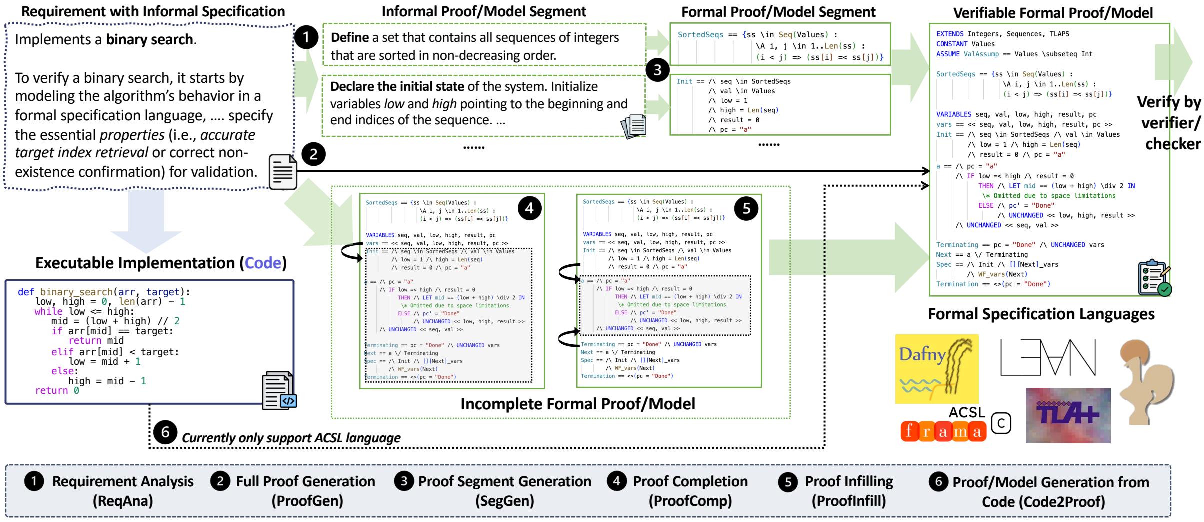

Figure 2: Six tasks towards Informal to Formal Verification

3 Data Construction

3.1 Formal Specification Language Selection

In this study, we consider five formal specification languages that can be used for formal verification, including Coq (Huet, 1986), Dafny (Leino, 2010), Lean4 (Moura and Ullrich, 2021), ACSL (ANSI/ISO C Specification) (Baudin et al., 2021) and TLA+ (Yu et al., 1999; Lamport, 2002). We selected them in order to cover various ***verification paradigms*** (*i.e.,* theorem proving and model checking) and ***verification scenarios*** (*e.g.,* mathematical reasoning and program verification).

First, for ***interactive theorem provers*** which are suitable for developing rigorous mathematical proofs, we consider **Coq** (Huet, 1986) and **Lean4** (Moura and Ullrich, 2021) because Coq has been extensively used in academia and research for proving mathematical theorems and in formal verification of software for a long history, while Lean4 garnered considerable attention from the mathematical community (Wang et al., 2024; Tao et al., 2023; Avigad et al., 2020) recently. Second, for ***programming languages with built-in specification***, we consider Dafny (Leino, 2010) and ACSL (Baudin et al., 2021; Cuoq et al., 2012) because they seamlessly integrate specifications (*e.g.,* pre-/post-conditions, loop invariants) within the code, ensuring the correctness through embedded assertions and conditions. Lastly, for ***model checking*** (Jhala and Majumdar, 2009; Clarke, 1997), we consider **TLA+** (Yu et al., 1999) since it is a representative math-based formal language for modeling algorithms and programs such as concurrent and distributed systems.

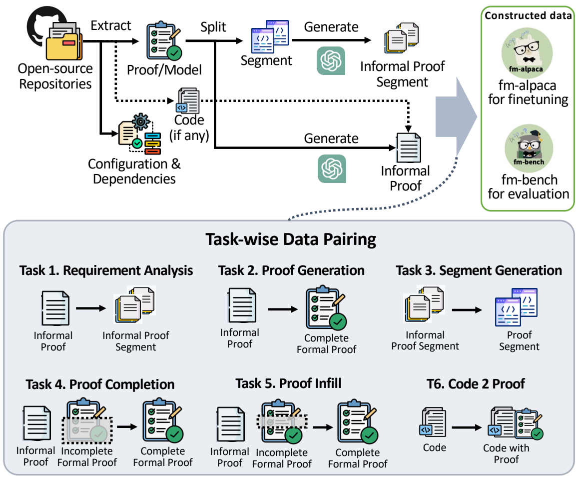

Figure 3: The Illustration of Data Preparation.

3.2 Data Preparation

The workflow of data preparation for FM-ALPACA and FM-BENCH is illustrated in Figure 3. The workflow begins with the data collection, where formal proofs in the desired formal specification languages and related configurations and dependencies are gathered from open-source repositories in Github. Then, formal proofs are **extracted** from the collected repositories. Next, the proofs go through the data quality assurance check by execution, the proofs that cannot be verified successfully are filtered out. The remaining ones are **split** into segments (*e.g.,* definition of functions or conditions).

Given the impracticality of manually writing descriptions for all the collected formal proofs, we leveraged *distill*ed GPT4 (gpt, 2023) to gen-

Spec	Num of Proofs			Num of Segments		
	Total	FM-ALPACA	FM-BENCH	Total	FM-ALPACA	FM-BENCH
Coq	2,126	1,683	443	14,939	11,638	3,301
Lean4	1,163	919	244	1,578	1,261	317
ACSL	544	426	118	765	598	167
Dafny	249	206	43	417	348	69
TLA+	256	199	57	594	476	118
Total	4,338	3,433	905	18,293	14,321	3,972
Average	868	687	181	3,659	2,864	794

Table 1: Formal-Specification-Language-wise Statistics of Formal Verification Data

	Task	Total	FM-ALPACA	FM-BENCH
1	Requirement Analysis	627	496	131
2	Full Proof Generation	700	557	143
3	Segment Proof Generation	14843	11597	3246
4	Proof Complete	658	520	138
5	Proof Infill	1439	1146	293
6	Code2Proof	70	56	14
	Total	18337	14372	3965
	Average	3056	2395	661

Table 2: Task-wise Statistics of Formal Verification Data

erate high-quality informal proof descriptions via meticulous prompting. This alternative is well-established and frequently employed in prior literature (Wang et al., 2022, 2024; Ding et al., 2023; Wang et al., 2023). Specifically, for each formal specification language, we designated the model as an expert in that particular language, equipping it with comprehensive domain knowledge about the language's specifications, essential grammatical cues, and three-shot examples featuring proof segments in the formal language as inputs and natural language descriptions as outputs. This approach ensures that the collected descriptions are of high quality and well-organized. It's important to note that we did not generate descriptions for proof segments shorter than two lines (such as package imports or constant definitions) because their meaning is self-explained, with the exception of ACSL, whose proof segments are typically 1-2 lines.

After the descriptions for both full and segment proofs were prepared, we then prepared the data pairs for each task as shown in **Task-wise Data Pairing** in Figure 3. Note that for Task 4, *i.e.,* Proof Completion, to prepare the incomplete formal proof, we randomly choose a line number and delete the lines in the proof after the line. For Task 5, *i.e.,* Proof Infill, we randomly choose two line numbers and mask the lines between them. In case the remaining lines of proof cannot provide sufficient information for the proof generation, we also provide the informal proof for these two tasks.

After pairing the instruction-response for different tasks, we manually designed five task instructions for each task and randomly assigned one for each paired data to increase instruction diversity and avoid overfitting to certain instructions (Lu et al., 2022; Feng et al., 2023; Sanh et al., 2022).

3.3 Data Statistics

The specification-language-wise and task-wise statistics are shown in Table 1 and Table 2. In particular, Table 1 presents a detailed breakdown of the number of proofs and segments across five specification languages. Note that we split the prepared data for all the tasks and specification languages into an 8:2 ratio, *i.e.,* 80% for fine-tuning, named FM-ALPACA, 20% for benchmarking, named FM-BENCH, and show the separate statistics. In particular, there are 4k+ verifiable proofs in total, with 249 $\sim$ 2k proofs for each language. These proofs were split into 18k+ proof segments, with an average of 3.6k segments for each language. The reason why the ratio of segments in FM-ALPACA and FM-BENCH is slightly less than 8:2 is that the train-test split was applied to proofs, while the number of split segments in each proof varies.

Table 2 shows the task-wise statistics. There are 18k instructions across six tasks, with an average of 3k instructions for each task. After splitting the train-test set, FM-ALPACA contains 14k, and FM-BENCH has nearly 4k instructions. It is clear that the number of instructions for the task Segment Proof Generation (SegGen) is far more than that for Requirement Analysis (ReqAna) and Full Proof Generation (ProofGen) because one full proof can be split into numerous pieces of proof segments, and one proof can contribute to only one instruction for ReqAna and ProofGen. Note that the number of ReqAna (627) and ProofGen (700) is unequal because we filtered out the instructions with more than 2048 tokens considering the context limits.

3.4 Validation Mechanism

For the tasks whose outputs are written in formal specification languages, we verify the full proofs against the corresponding verifiers, *i.e.,* Coq, Dafny, Lean4 use their own proving environment; formal specification written in TLA+ can be checked by TLC (Yu and Kuppe); C programs with ACSL specifications can be checked by Frama-C (Cuoq et al., 2012; Carvalho et al., 2014). Also, for the proof segments that cannot be verified independently, for each extracted segment, we prepared a proof template with a placeholder during data preparation. Whenever a generated segment is to be verified, we replace the placeholder in the template with the segment and verify the completed formal proof.

For ACSL specification verification, we report the results under two modes. First, **Basic checking**. ACSL uses 'frama-c' tool for syntax checking (*i.e.,* whether the syntax is correct), semantics checking (*i.e.,* program behaviors are consistent with the annotated ACSL specifications), and other basic ACSL checking. Second, **Verification**. 'ACSL-WP' uses 'frama-c' with the WP plugin for formal verification. This mode requires not only the basic checking (*i.e.,* syntactically correct, semantically consistent with the program), but also that the properties under verification can be proved.

Note that the invalid outputs (*e.g.,* empty response or responses without formal specifications) are considered incorrect directly and will not go through the verification.

For the task whose outputs are written in natural language (*i.e.,* ReqAna), we calculate the Bleu score (Papineni et al., 2002) between the descriptions in FM-BENCH with the predicted outputs.

4 Experiments

4.1 Experiment Setup

Studied LLMs. We selected ten LLMs as baselines without fine-tuning, including llama3.1-instruct-8B/70B (Meta, 2024), qwen2.5-instruct-7B/72B (Yang et al., 2024a), qwen2.5-coder-instruct-7B/-32B (Hui et al., 2024), starcoder-instruct-15B (Lozhkov et al., 2024), deepseek-coder-instruct-7B-v1.5, deepseek-coder-instruct-33B (Guo et al., 2024), and deepseek-R1 (Guo et al., 2025). Note that we avoid evaluating the GPT-series LLMs by OpenAI because the descriptions in FM-BENCH were generated by GPT-4o, making the evaluation fairer.

Fine-tuning. Instruction fine-tuning (Wei et al., 2021; Sanh et al., 2021; Ding et al., 2023; Ivison et al., 2023) aims to improve a model's ability to effectively respond to human instructions and has shown strong experimental potential in model enhancement. We select llama3.1-8B-instruct (Meta, 2024), qwen2.5-7B-instruct (Yang et al., 2024a), and deepseek-coder-7B-instruct-v1.5 (Guo et al., 2024) as base models for fine-tuning. These three models are chosen due to their promising performance on tasks such as coding, mathematics, and reasoning, and fine-tuning models in their scale is relatively affordable compared with fine-tuning larger scale models. We fine-tuned the three aforementioned models over three epochs using a learning rate 2e-5, a warm-up ratio of 0.04, a batch size of 512, and a cosine learning rate scheduler.

Baseline Fine-tuning Datasets: To distinguish whether the capability improvement is simply because more instruction tuning is applied, we also include two commonly used fine-tuning datasets for comparison. We select UltraChat (Ding et al., 2023) and Tulu-V3 (Lambert et al., 2024) as baseline fine-tuning datasets, and use llama3.1–8B-base (Meta, 2024) as the base model due to their popularity. In particular, UltraChat is a large-scale dataset of instructional conversations that contains 1.5 million high-quality multi-turn dialogues and covers a wide range of topics and instructions. Tulu-v3 (Lambert et al., 2024) embraces new data that is either carefully manu ally curated for quality or generated from GPT models. It is an enhancement of its previous versions (Ivison et al., 2023; Wang et al., 2023), focusing more on core skills of knowledge recall, reasoning, mathematics, coding, instruction following, general chat, and safety.

Benchmarks for Related Capabilities (RQ4). To comprehensively evaluate the model's capabilities, we tested the fine-tuned models on a series of benchmarks: Math (Hendrycks et al., 2021) and GSM-8K (Cobbe et al., 2021) for mathematical reasoning, BBH (Suzgun et al., 2022) for general reasoning, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021b) for coding.

Inference Strategies: We adopt different settings for different RQs. In particular, We use (1) the greedy sampling strategy to generate one single greedy sample with a temperature of 0.0 and calculate Pass@1, and (2) nucleus sampling (Holtzman et al., 2020), where five solution samples were randomly generated with a temperature of 0.2 for RQ1 and RQ2. We also consider different in-context learning strategies, including zero-shot and few-shot (we used 3-shot in the experiment). For RQ3, we use a zero-shot greedy search with a temperature of 0.0 and a few-shot nuclear search with a temperature of 0.2 for a fair comparison.

Experiment Environment. The fine-tuning experiment was conducted on 32 Nvidia A800-40G GPUs, while inference was on Nvidia A100-80G GPUs with vLLM (Kwon et al., 2023).

LLMs	Size	ProofGen						SegGen						ProofComplete				ProofInfill						Cd2Prf		Ave
		TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	Dafny	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	ACSL	ACSL-WP	
W/o Fine-tuning																										
llama3.1-instruct	8B	0.00	4.26	1.69	0.00	14.29	0.00	1.43	1.19	6.06	8.33	0.00	0.00	0.00	3.57	5.56	28.57	0.00	4.85	7.69	13.33	21.43	0.00	57.14	0.00	7.47
llama3.1-instruct	70B	7.69	6.38	10.17	20.00	7.14	0.00	24.29	4.86	13.64	11.11	39.68	3.17	27.27	3.57	13.89	28.57	0.00	7.27	24.62	13.33	21.43	0.00	21.43	0.00	12.90
qwen2.5-instruct	7B	0.00	2.13	5.08	0.00	21.43	0.00	1.43	1.05	7.58	5.56	1.59	1.59	0.00	13.10	19.44	28.57	0.00	5.45	18.46	13.33	21.43	0.00	14.29	0.00	7.56
qwen2.5-instruct	72B	7.69	6.38	18.64	20.00	21.43	7.14	12.86	3.29	12.12	25.00	12.70	4.76	0.00	11.90	33.33	28.57	0.00	9.70	24.62	26.67	35.71	0.00	57.14	0.00	15.82
qwen2.5-coder-instruct	7B	0.00	2.13	6.78	0.00	0.00	0.00	2.86	2.14	12.12	11.11	3.17	0.00	0.00	4.76	11.11	14.29	0.00	3.64	23.08	20.00	0.00	0.00	0.00	0.00	4.88
qwen2.5-coder-instruct	32B	0.00	6.38	11.86	0.00	21.43	0.00	12.86	4.07	9.85	25.00	26.98	4.76	27.27	5.95	36.11	42.86	2.94	7.88	33.85	20.00	71.43	14.29	57.14	14.29	19.05
starcoder-instruct	15B	7.69	2.13	8.47	30.00	28.57	0.00	27.14	1.97	6.06	27.78	0.00	0.00	27.27	11.90	25.00	42.86	0.00	9.70	24.62	20.00	35.71	0.00	28.57	0.00	15.23
deepseek-coder-instruct	7B	0.00	2.13	8.47	0.00	0.00	0.00	4.29	1.73	3.79	5.56	0.00	0.00	9.09	2.38	13.89	28.57	0.00	3.03	13.85	33.33	0.00	0.00	0.00	0.00	5.42
deepseek-coder-instruct	33B	0.00	0.00	1.69	0.00	0.00	0.00	2.86	1.94	4.55	11.11	3.17	0.00	0.00	3.57	13.89	42.86	0.00	2.42	23.08	6.67	7.14	0.00	7.14	0.00	5.50
deepseek-r1	671B	30.77	8.51	30.51	30.00	35.71	7.14	22.86	10.70	21.21	22.22	49.21	3.17	45.45	15.48	33.33	28.57	5.88	9.70	24.62	6.67	42.86	21.43	64.29	14.29	**24.36**
Task-wise average		7.70						9.16						18.31				12.62						16.79		
W/ Fine-tuning																										
llama3.1-**fma**	8B	0.00	6.38	8.47	20.00	0.00	0.00	38.57	29.58	21.97	25.00	88.89	17.46	36.36	8.33	11.11	28.57	0.00	0.00	7.69	0.00	21.43	0.00	21.43	0.00	16.30
llama3.1-ultrachat	8B	0.00	6.38	3.39	0.00	0.00	0.00	0.00	0.00	3.79	0.00	0.00	0.00	0.00	5.95	13.89	14.29	0.00	4.85	9.23	0.00	14.29	7.14	0.00	0.00	3.47
llama3.1-ultrachat-**fma**	8B	0.00	8.51	10.17	30.00	0.00	0.00	41.43	35.79	29.55	33.33	95.24	17.46	18.18	8.33	11.11	28.57	2.94	0.00	7.69	0.00	28.57	0.00	35.71	0.00	18.44
llama3.1-tulu	8B	0.00	2.13	1.69	0.00	0.00	0.00	0.00	0.71	6.06	0.00	0.00	0.00	0.00	5.95	8.33	28.57	0.00	1.82	10.77	0.00	7.14	0.00	14.29	0.00	3.64
llama3.1-tulu-**fma**	8B	0.00	4.26	11.86	30.00	7.14	0.00	42.86	36.64	27.27	36.11	98.41	17.46	18.18	10.71	11.11	42.86	2.94	2.42	6.15	0.00	35.71	0.00	50.00	0.00	20.50
qwen2.5-**fma**	7B	0.00	4.26	11.86	10.00	0.00	0.00	38.57	27.95	27.27	22.22	87.30	17.46	27.27	9.52	13.89	28.57	0.00	0.00	7.69	0.00	21.43	0.00	14.29	0.00	15.40
qwen2.5-coder-**fma**	7B	0.00	6.38	18.64	20.00	7.14	0.00	44.29	36.50	34.09	33.33	98.41	17.46	36.36	9.52	16.67	28.57	5.88	1.82	13.85	13.33	42.86	0.00	42.86	0.00	**22.00**
deepseek-coder-**fma**	7B	0.00	2.13	16.95	0.00	0.00	0.00	34.29	25.30	31.06	25.00	84.13	15.87	36.36	8.33	19.44	28.57	0.00	1.82	7.69	6.67	14.29	7.14	14.29	7.14	16.10
Task-wise average		6.50						39.82 ↑						20.69 ↑				7.22						15.48		

* **-fma**: fine-tuned with FM-ALPACA. * **-ultrachat**: fine-tuned with UltraChat. * **-tulu**: fine-tuned with Tulu3. * **-ultrachat-fma**: fine-tuned with both UltraChat and FM-ALPACA. * **-tulu-fma**: fine-tuned with both Tulu3 and FM-ALPACA.
* **ACSL and ACSL-WP**: 'ACSL' uses 'frama-c' tool for syntax checking, semantics checking, and other basic ACSL checking; 'ACSL-WP' uses 'frama-c' with the WP plugin for specification verification.
The task-wise average of finetuned models is calculated excluding the models (*i.e.,* llama3.1-ultrachat and llama3.1-tulu) that are finetuned without FM-ALPACA to remove the influence of the average value of the control group models' results.

Table 3: **RQ1-3: Pass@1 Accuracy of LLMs' Performance Across Formal Verification Task and Formal Specification Languages** with (w/) and without (w/o) fine-tuning. The greener, the better.

LLMs	Size	ReqSplitter				
		TLA	Coq	Lean	Dafny	ACSL
w/o fine-tune						
llama3.1-instruct	8B	0.33	0.26	0.35	0.36	0.28
llama3.1-instruct	70B	0.33	0.28	0.35	0.41	0.30
qwen2.5-instruct	7B	0.29	0.25	0.32	0.33	0.19
qwen2.5-instruct	72B	0.33	0.28	0.33	0.34	0.22
qwen2.5-coder-instruct	7B	0.29	0.26	0.35	0.42	0.25
qwen2.5-coder-instruct	32B	0.31	0.27	0.33	0.39	0.23
star-coder-instruct	15B	0.21	0.24	0.29	0.30	0.29
deepseek-coder-instruct	7B	0.34	0.30	0.39	0.49	0.36
deepseek-coder-instruct	33B	0.37	0.29	0.47	0.47	0.43
deepseek-r1	671B	0.32	0.27	0.33	0.27	0.28
Average		31.85				
w/ fine-tune						
llama3.1-**fma**	8B	0.33	0.18	0.49	0.49	0.42
llama3.1-ultrachat	8B	0.35	0.30	0.44	0.40	0.46
llama3.1-ultrachat-**fma**	8B	0.37	0.30	0.56	0.57	0.62
llama3.1-tulu	8B	0.41	0.30	0.43	0.42	0.39
llama3.1-tulu-**fma**	8B	0.57	0.33	0.57	0.59	0.71
qwen2.5-**fma**	7B	0.33	0.13	0.53	0.58	0.38
qwen2.5-coder-**fma**	7B	0.51	0.21	0.57	0.64	0.73
deepseek-coder-**fma**	7B	0.43	0.24	0.46	0.54	0.43
Average		44.36 (**39%** ↑)				

* **-fma**: fine-tuned with FM-ALPACA.

Table 4: Evaluation on Requirement Analysis

4.2 RQ1. Basic Performance across Formal Specification Tasks

To understand the current LLMs' performance in six tasks, we evaluate 10 LLMs against FM-BENCH with model size ranges from 7B to 671B. The upper part of Table 3 and the upper part of Table 4 show LLMs' basic performance without fine-tuning.

Task-wise: LLMs perform the best in ProofComplete (18.31%) and Code2Proof (16.79%). In contrast, LLMs fall short in generating both the entire formal proof (7.70%) and the proof segments (9.16%). We analyzed the failures and found that syntax errors account for a large proportion, with 12.43% failures caused by syntax errors (See Appendix B). The observation echoes the motivation of prior work (Wang et al., 2024) and is reasonable due to the grammar difference between most formal specification languages and other programming languages like Python. Regarding requirement analysis, as shown in the upper part of Table 4, the Bleu scores between the ground-truth description and LLM-generated ones range from 0.19 to 0.49.

LLM-wise: Without fine-tuning, DeepSeek-R1 achieved the best average (24.36%), followed by qwen2.5-coder-instruct-32B (19.05%).

Model Size: Larger LLMs generally perform better than smaller LLMs. For example, llama3.1-8B only achieved 1.43% in generating TLA+ segments, while llama3.1-70B boosts to 24.29% in the same task. However, there are several exceptions worth noticing, especially for ProofInfill and Code2Proof. For example, llama-3.1-**8**B achieved 57.14% in Code2Proof (ACSL), yet the performance drops to 21.43% using the llama-3.1-**70**B model. The decrease in performance is inherently due to the fine-tuning strategy of these instruction models: they are trained to excel in generating rather than filling in the blanks (Fried et al., 2022). Also, we conducted a more detailed examination of generated segments and observed that larger LLMs tend to fill in the proof segments that ***not only look***

more plausibly correct and well-organized but also include extra content. The additional content, yet, is either redundant, as it repeats information that appears in the subsequent proof, or is incomplete. Promisingly, recent model developers have noticed such conundrums and refined their fine-tuning strategy for fill-in-the-middle tasks (Guo et al., 2024).

4.3 RQ2. Formal Specification Languages-wise Capability

Table 5 shows the LLMs' performance across formal specification languages in the task of generating proof segments (SegGen). This task accounts for the most instructions and serves as the basic capability for other proof generation tasks. We can see that LLMs perform the best in ACSL (average: 34.92%), followed by Dafny (15.97%) while performing unsatisfactorily in other formal specification languages. The observation is reasonable because the syntax of ACSL is basically an annotation of C language, while Dafny shares similar grammar as C# and Java. Thus, generating proof segments in ACSL and Dafny is generally easier than generating other specification languages.

Note that though LLMs are generally good at generating syntactically correct and semantically consistent (*i.e.,* program behaviors are consistent with the annotated ACSL specifications) according to the 'ACSL' column in Table 3, the specifications are usually insufficient for the verification according to the 'ACSL-WP' column in Table 3. The results in the two ACSL columns indicate that there is a large room to improve for generating not only syntactically/semantically correct specifications, but also that the generated specifications need to be sufficient to verify the properties under verification.

In addition, we explore whether increasing the attempts ($1 \rightarrow 5$) with a higher temperature ($0.0 \rightarrow 0.2$) and in-context learning could bring about improvement. The improvement ratios are shown in red in Table 5. The results of Pass@5 are better than those of Pass@1, with an average score increase from 10.82% (Dafny) to 63.64% (ACSL) in different languages. Moreover, when using 3-shot, the performance increases dramatically, with 51.33% (Dafny) to over five times (ACSL) improvement compared with zero-shot Pass@5. The results indicate the potential of in-context learning in generating correct specification languages.

4.4 RQ3. Improvement by Fine-tuning

We further investigate whether FM-ALPACA could bring about improvement. The lower part of Table 3 and Table 4 shows the results. From Table 3, dramatic improvements can be observed in generating full and segmental proofs after fine-tuning. Note that the model size of fine-tuned models is 7B $\sim$ 8B, while the performance largely outperforms the 70B+ models without fine-tuning. Furthermore, after fine-tuning with formal data, ***the 7 $\sim$ 8B fine-tuned models can achieve comparable or slightly better performance than Deepseek-R1-671B***, with 27.31% achieved by qwen2.5-coder-7B fine-tuned with FM-ALPACA (R1-671B: 27.11%). It may suggest the possibility of distilling domain-specific small models for handier usage.

Task-wise: Improvements in generation tasks (*i.e.,* ProofGen, SegGen, and ProofComplete) are substantial. ProofGen doubles the performance, and SegGen more than triples. The dramatic increases happen in all models fine-tuned with FM-ALPACA in SegGen Task, from nearly all zeros to 29.98% $\sim$ 90.48%. An increase of 41% can also be observed in Table 4. The experimental improvements make evident the effectiveness of fine-tuning in formal verification tasks.

Yet, drops can be observed in fill-in-the-middle tasks (*i.e.,* ProofInfill and Code2Proof). The results echo the observation made in RQ1 (Section 4.2), where the large LLMs perform worse than small LLMs in fill-in-the-middle tasks. The results also indicate the necessity of adopting different fine-tuning strategies other than instruction tuning only.

Fine-tuning Datasets: Take a closer look at the LLMs fine-tuned with general-purpose datasets (*i.e.,* llama3.1-ultrachat and llama3.1-tulu) in Table 3, with them only, no or opposite effects can be observed. The results indicate the ***complementarity*** of FM-ALPACA and existing general-purpose fine-tuning datasets. Additionally, by combining with general-purpose datasets, the performance can be ***further improved*** (*e.g.,* llama3.1-tulu-fma).

Comparison with Few-shot: Compared with the best results in Table 5 achieved by 3-shot, the results after fine-tuning (Table 3) still generally outperform the 3-shot results. The results indicate that although in-context learning can improve LLMs' performance, the enhancement is limited. ***Further significant improvements still require fine-tuning with formal data***. This may also suggest that in-context learning alone cannot adequately address

LLMs	Size	TLA			Coq			Lean			Dafny			ACSL			ACSL-WP		
		Zero-shot		Few-Shot	Zero-shot		Few-Shot	Zero-shot		Few-Shot	Zero-shot		Few-Shot	Zero-shot		Few-Shot	Zero-shot		Few-Shot
		P@1	P@5	P@1	P@1	P@5	P@1	P@1	P@5	P@1	P@1	P@5	P@1	P@1	P@5	P@1	P@1	P@5	P@1
llama3.1-instruct	8B	1.43	4.29	11.43	1.19	1.90	10.12	6.06	8.33	9.85	8.33	11.11	19.44	0.00	3.17	38.10	0.00	0.00	12.70
llama3.1-instruct	70B	24.29	27.14	38.57	4.86	6.01	3.46	13.64	17.42	15.91	11.11	13.89	16.67	39.68	52.38	76.19	3.17	4.76	14.29
qwen2.5-instruct	7B	1.43	2.86	5.71	1.05	1.39	12.80	7.58	9.09	9.85	5.56	5.56	13.89	1.59	3.17	66.67	1.59	3.17	15.87
qwen2.5-instruct	72B	12.86	17.14	24.29	3.29	3.80	21.05	12.12	14.39	19.70	25.00	25.00	25.00	12.70	15.87	90.48	4.76	4.76	15.87
qwen2.5-coder-instruct	7B	2.86	5.71	12.86	2.14	2.99	14.30	12.12	13.64	18.94	11.11	11.11	19.44	3.17	4.76	87.30	0.00	0.00	17.46
qwen2.5-coder-instruct	32B	12.86	15.71	30.00	4.07	4.96	18.40	9.85	15.15	13.64	25.00	27.78	30.56	26.98	33.33	96.83	4.76	4.76	17.46
deepseek-coder-instruct	6.7B	4.29	8.57	4.29	1.73	2.82	8.79	3.79	5.30	17.42	5.56	5.56	19.44	0.00	4.76	60.32	0.00	0.00	9.52
deepseek-coder-instruct	33B	2.86	2.86	18.57	1.94	3.23	11.27	4.55	6.82	22.73	11.11	13.89	22.22	3.17	25.40	92.06	0.00	1.59	15.87
Language-wise Average		12.20			6.15			11.99			15.97			34.92			6.35		
Average		7.86	10.54	18.21	2.53	3.39	12.53	8.71	11.27	16.00	12.85	14.24	20.83	10.91	17.86	75.99	1.79	2.38	14.88
Improvement Ratio			34.09%↑	72.88%↑		33.67%↑	269.80%↑		29.35%↑	42.02%↑		10.81%↑	46.34%↑		63.64%↑	325.56%↑		33.33%↑	525.00%↑

* The improvement ratios shown in "Pass@5" column are calculated by comparing with the scores in Pass@1, and the ratios shown in "Few-shot" column are calculated by comparing with the scores in Pass@5.

* **ACSL and ACSL-WP**: 'ACSL' uses 'frama-c' tool for syntax, semantics checking, and basic ACSL checking; 'ACSL-WP' uses 'frama-c' with the WP plugin for specification verification.

Table 5: **RQ2: Language-wise LLMs' Performance**. Pass@1 and Pass@5 Accuracy in Generating Proof Segments Across Formal Specification Languages Under Zero/Few-shot without fine-tuning.

Fine-tuning Dataset	MATH			Reasoning	Coding			Average
	MATH	GSM-8K	Average	bbh	HumanEval	MBPP	Average	
UltraChat	**17.54**	61.33	39.44	**62.64**	19.51	**36.4**	27.96	39.48
UltraChat + **fma**	16.16 7.87%↓	**62.32** **1.61% ↑**	39.24 0.49%↓	62.14 0.80%↓	**31.71** **62.53% ↑**	35.2 3.30%↓	**33.46** **19.67% ↑**	**41.51** **5.14% ↑**
tulu3	27.36	**75.82**	51.59	62.47	64.02	48.8	56.41	55.01
tulu3 + **fma**	**29.48** **7.75% ↑**	75.44 0.50%↓	52.46 **1.69% ↑**	**63.16** **1.10% ↑**	**64.63** **0.95% ↑**	**49.4** **1.23% ↑**	**57.02** **1.07% ↑**	**55.76** **1.37% ↑**

Table 6: **RQ4: Capability Migration from FM-ALPACA to Math, Reasoning, and Coding**.

capability deficits in formal verification tasks but rather stem from a lack of knowledge.

4.5 RQ4. Capability Migration from Formal Verification to Related Tasks

Finally, we explore whether fine-tuning with FM-ALPACA could benefit related capabilities. Table 6 shows the results. The base model is llama3.1-8B, fine-tuned under two base fine-tuned datasets with and without FM-ALPACA. On average, with FM-ALPACA, an increase of 1.37% to 5.15% can be observed. Interestingly, a dramatic increase (62.53%) can be observed in HumanEval compared with the performance of the model that is only fine-tuned with UltraChat. The experiment may indicate that feeding more formal data may improve LLMs' coding, reasoning, and math capabilities.

5 Conclusion

This paper contributes a comprehensive assessment and formulation to understand LLMs' capability in formal verification. We constructed 18k high-quality instruction-response pairs across five formal specification languages in six tasks. The fine-tuned models, fine-tuning data, and the benchmark are released to facilitate subsequent studies.

Limitations

This paper has two primary limitations that offer avenues for future research. First, the primary limitation of our work is that our benchmark relies on model-generated data. While this approach effectively reduces manual efforts; it may introduce biases and data leakage issues in the dataset towards the models that generated the data. To address this limitation, we use gpt-4o to generate the natural language descriptions, while during the evaluation, we use other LLMs for evaluation. Second, another limitation of our work lies in the validation design. When creating ProofInfill and ProofComplete data, it is possible that the properties to be verified or theorems to be proven are masked. If LLMs happened not to generate these properties/theorems, the generated "proofs/models" could escape the verifier/checker, mistakenly labeling the output as correct. To avoid this scenario, we include the requirement descriptions as part of the input, guiding LLMs to generate the necessary properties or theorems without omission.

6 Acknowledgments

We sincerely thank the reviewers for their insightful comments and valuable suggestions. This work was supported in part by the Hong Kong Research Grants Council/General Research Fund (HKSAR RGC/GRF, Grant No. 16206524), the National Natural Science Foundation of China (Grant Nos. 62372193, 62192734, 62302375, 62472339), CAS Project for Young Scientists in Basic Research (Grant No.YSBR-040) and the Basic Research Program of ISCAS (Grant No. ISCAS-ZD-202402).

References

2023. Gpt-4 and gpt-4-turbo preview. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models#gpt-4-and-gpt-4-turbo-preview.

DeepMind AlphaProof and AlphaGeometry Teams. 2024. Ai achieves silver-medal standard solving international mathematical olympiad problems.'25 july 2024.

Annie I Anton. 1996. Goal-based requirements analysis. In *Proceedings of the second international conference on requirements engineering*, pages 136–144. IEEE.

Andrew W. Appel. 2011. Verified software toolchain. In *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg. Springer Berlin Heidelberg.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021a. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021b. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Jeremy Avigad, Kevin Buzzard, Robert Y Lewis, and Patrick Massot. 2020. Mathematics in lean.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *Preprint*, arXiv:2302.12433.

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2021. Acsl: Ansi/iso c specification.

Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, and Aaron Tomb. 2014. Formal verification of klibc with the wp frama-c plug-in. In *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29–May 1, 2014. Proceedings 6*, pages 343–358. Springer.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

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

Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Stephen A Cook. 2023. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c: A software analysis perspective. In *International conference on software engineering and formal methods*, pages 233–247. Springer.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

Alan M Davis. 1990. *Software requirements: analysis and specification*. Prentice Hall Press.

Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. 2013. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 290–300, New York, NY, USA. Association for Computing Machinery.

Deva Kumar Deeptimahanti and Muhammad Ali Babar. 2009. An automated tool for generating uml models from natural language requirements. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 680–682. IEEE.

Giuseppe Della Penna, Benedetto Intrigila, Daniele Magazzeni, Igor Melatti, and Enrico Tronci. 2013. Cgmurphi: Automatic synthesis of numerical controllers for nonlinear hybrid systems. *European Journal of Control*, 19(1):14–36.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051, Singapore. Association for Computational Linguistics.

Agung Fatwanto. 2012. Translating software requirements from natural language to formal specification. In *2012 IEEE International Conference on Computational Intelligence and Cybernetics (CyberneticsCom)*, pages 148–152. IEEE.

Chun-Mei Feng, Kai Yu, Yong Liu, Salman Khan, and Wangmeng Zuo. 2023. Diverse data augmentation with diffusions for effective test-time prompt tuning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2704–2714.

Deborah Ferreira and André Freitas. 2020. Natural language premise selection: Finding supporting statements for mathematical text. *arXiv preprint arXiv:2004.14959*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Jeffrey O Grady. 2010. *System requirements analysis*. Elsevier.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3).

Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad apps:{End-to-End} security via automated {Full-System} verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Gérard Huet. 1986. *The calculus of constructions*. Ph.D. thesis, INRIA.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2024. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, page 103917.

Hamish Ivison, Yizhong Wang, Valentina Pyatkin, Nathan Lambert, Matthew Peters, Pradeep Dasigi, Joel Jang, David Wadden, Noah A Smith, Iz Beltagy, et al. 2023. Camels in a changing climate: Enhancing lm adaptation with tulu 2. *arXiv preprint arXiv:2311.10702*.

Muhammad Javed and Yuqing Lin. 2021. imer: Iterative process of entity relationship and business process model extraction from the requirements. *Information and Software Technology*, 135:106558.

Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Dongming Jin, Shengxin Zhao, Zhi Jin, Xiaohong Chen, Chunhui Wang, Zheng Fang, and Hongbin Xiao. 2024. An evaluation of requirements modeling for cyber-physical systems via llms. *arXiv preprint arXiv:2408.02450*.

Zhi Jin. 2017. *Environment modeling-based requirements engineering for software intensive systems*. Morgan Kaufmann.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA. Association for Computing Machinery.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. 2024. T\" ulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*.

Leslie Lamport. 2002. Specifying systems: the tla+ language and tools for hardware and software engineers.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

Weixian Waylon Li, Yftah Ziser, Maximin Coavoux, and Shay B Cohen. 2023. Bert is not the count: Learning to match mathematical statements with proofs. *arXiv preprint arXiv:2302.09350*.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*.

Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, Haiming Wang, Wei Ju, Chuanyang Zheng, Yichun Yin, Lin Li, Ming Zhang, and Qun Liu. 2023. Fimo: A challenge formal dataset for automated theorem proving. *Preprint*, arXiv:2309.04295.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Yuning Lu, Jianzhuang Liu, Yonggang Zhang, Yajing Liu, and Xinmei Tian. 2022. Prompt distribution learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5206–5215.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *Preprint*, arXiv:2401.08807.

The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL'20. ACM.

AI Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date. Blog. Online; accessed 15-January-2024.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg. Springer-Verlag.

Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2024. Laurel: Generating dafny assertions using large language models. *Preprint*, arXiv:2405.16792.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Tali Bers, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask prompted training enables zero-shot task generalization. *Preprint*, arXiv:2110.08207.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Towards large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534*.

Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Calibration and correctness of language models for code. *arXiv preprint arXiv:2402.02047*.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-loop verifiable code generation. *Preprint*, arXiv:2310.17807.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2727–2735.

William Taggart Jr and Marvin O Tharp. 1977. A survey of information requirements analysis techniques. *ACM Computing Surveys (CSUR)*, 9(4):273–290.

Terence Tao. 2024. Ai will become mathematicians' "co-pilot". https://www.scientificamerican.com/article/ai-will-become-mathematicians-co-pilot/.

Terence Tao, Yael Dillies, and Bhavik Mehta. 2023. Formalizing the proof of pfr in lean4 using blueprint: a short tour. *Blog post, November*.

Norbert Tihanyi, Ridhi Jain, Yiannis Charalambous, Mohamed Amine Ferrag, Youcheng Sun, and Lucas C Cordeiro. 2023. A new era in software security: Towards self-healing software via large language models and formal verification. *arXiv preprint arXiv:2305.14752*.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482.

Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. 2024. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*.

Ye Wang, JW Chen, Xin Xia, and B Jiang. 2021. Intelligent requirements elicitation and modeling: A literature review. *Journal of Computer Research and Development*, 58(4):683–705.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. 2023. How far can camels go? exploring the state of instruction tuning on open resources. *Advances in Neural Information Processing Systems*, 36:74764–74786.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. 2021. Naturalproofs: Mathematical theorem proving in natural language. *Preprint*, arXiv:2104.01112.

Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. 2022. Naturalprover: Grounded mathematical proof generation with language models. *Advances in Neural Information Processing Systems*, 35:4913–4927.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pages 302–328. Springer.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*, pages 6984–6994. PMLR.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. 2024b. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*.

Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*.

Yuan Yu and Markus Kuppe. Tlc model checker. https://tla.msr-inria.inria.fr/tlatoolbox/doc/model/executing-tlc.html.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking tla+ specifications. In *Advanced research working conference on correct hardware design and verification methods*, pages 54–66. Springer.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*.

Qixiang Zhou, Tong Li, and Yunduo Wang. 2022. Assisting in requirements goal modeling: a hybrid approach based on machine learning and logical reasoning. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 199–209.

A Related Work

The formal specification datasets or benchmarks offer a standard, well-defined set of problems, providing a shared challenge that helps build a community of practice among researchers. According to different verification techniques, the existing benchmarks mainly fall into two categories; we discuss them separately.

A.1 Theorem Proving Datasets

Formal theorem proving represents theorems and proofs in a machine-verifiable format (Cook, 2023), ensuring their correctness using rigorous logical rules. A recent survey (Li et al., 2024) summarized the existing datasets for theorem proving. In particular, the ***informal benchmarks*** craft the proofs from various sources such as ProofWiki, textbooks, and public corpus. NL-PS (Ferreira and Freitas, 2020) first builds a natural language premise selection dataset source from ProofWiki. Similarly, NaturalProofs (Welleck et al., 2021) further incorporates data from Stacks and textbooks, resulting in a dataset with roughly 25k examples. Adapted from it, NaturalProofs-Gen (Welleck et al., 2022) contains around 14.5k theorems for informal proof generation. Moreover, MATcH (Li et al., 2023) constructs over 180k statement-proof pairs for matching using the MREC corpus [2].

For ***formal datasets***, a line of efforts focuses on extracting and cleaning theorems and proofs written in various specification languages (*e.g.,* Coq, Isabelle, Lean) from established formal libraries and verification projects. For example, LeanDojo (Yang et al., 2023) extracts over 98k theorems and proofs with 130k premises from Lean mathlib (mathlib Community, 2020). Besides extracting data from existing projects, several works ***manually*** annotate or formalize the problems in natural language. For example, MiniF2F (Zheng et al., 2021) manually formalizes 488 Olympiad-level problems across 4 proof systems and equally splits them into a validation set and a test set. FIMO (Liu et al., 2023) and ProofNet (Azerbayev et al., 2023) formalize the theorem statements of the International Mathematical Olympiad and undergraduate-level problems in Lean. In addition, datasets for Dafny also attract research contributions because industries like Amazon adopted Dafny to verify cryptographic libraries, authorization protocols, a random number generator, and the Ethereum virtual machine. Dafny datasets such as CloverBench (Sun et al., 2024) and DafnyGym (Mugnier et al., 2024).

A.2 Model checking datasets

Model checking is an automated technique used in computer science and formal methods to verify the correctness of systems, particularly those with finite state spaces. It systematically checks whether a system's model satisfies a given specification, usually expressed in formal specification languages. The basic idea is to explore all possible system states to ensure the desired properties hold in every conceivable scenario.

Model checking benchmarks are less than that for theorem proving. Currently, there are few model-checking benchmarks for proving, while several model-checking subjects are going with specific model-checking languages such as CMurphi (Della Penna et al., 2013) and TLA+ (Yu et al., 1999). In particular, CMurphi is a software tool used to verify concurrent and distributed systems through explicit state enumeration. It implements the Murphi verification language, which allows users to describe finite-state systems in a procedural style. The core principle behind CMurphi is to explore the state space of a system exhaustively to check for violations of specified invariants or properties. Another example is TLA+ (Temporal Logic of Actions), a high-level language for modeling programs and systems suitable for concurrent and distributed systems.

B Proportion of Failures Caused by Syntax Error

We listed the proportions of failures caused by syntax errors for each LLM and each task in Table 8. We used a set of pre-defined keywords (summarized in Table 7 to identify if a verification failure is caused by syntax errors. Specifically, we consider a failure caused by syntax error if its error message contains at least one keyword in Table 7.

	Language	Keywords
1	Coq	“Syntax Error:”
2	Lean4	“unexpected token ”, “unknown identifier”, “type mismatch”
3	ACSL	“unexpected token”
4	Dafny	“type errors detected”
5	TLA+	“***Parse Error***”, “Unknown operator”

Table 7: Keywords for identifying syntax error raised by each language's verifier.

[2] https://mir.fi.muni.cz/MREC/

LLMs	Size	ProofGen					SegGen					ProofComplete				ProofInfill					Cd2Prf
		TLA	Coq	Lean	Dafny	ACSL	TLA	Coq	Lean	Dafny	ACSL	TLA	Coq	Lean	Dafny	TLA	Coq	Lean	Dafny	ACSL	ACSL
llama3.1-instruct	8B	0/13	18/45	34/58	0/10	1/14	43/69	742/2910	78/124	4/33	4/63	4/11	45/81	16/34	1/5	16/34	44/157	27/60	0/13	1/14	0/14
llama3.1-instruct	70B	4/12	10/44	40/53	1/8	1/14	43/53	71/2802	75/114	16/32	5/61	6/8	44/81	10/31	1/5	15/34	39/153	22/49	0/13	2/14	3/14
qwen2.5-instruct	7B	12/13	13/46	33/56	1/10	0/14	26/69	302/2914	53/122	5/34	6/62	9/11	35/73	8/29	1/5	14/34	66/156	31/53	0/13	4/14	3/14
qwen2.5-instruct	72B	10/12	6/44	28/48	1/8	9/13	47/61	40/2848	54/116	11/27	0/60	7/11	35/74	8/24	1/5	13/34	59/149	20/49	0/11	0/14	1/14
qwen2.5-coder-instruct	7B	11/13	18/46	38/55	0/10	0/14	60/68	22/2882	72/116	6/32	53/63	10/11	12/80	8/32	2/6	17/34	34/159	26/50	1/12	0/14	0/14
qwen2.5-coder-instruct	32B	11/13	12/44	29/52	1/10	2/14	50/61	31/2825	68/119	9/27	25/60	5/8	24/79	6/23	3/4	17/33	51/152	16/43	0/12	0/12	0/12
deepseek-coder-instruct	7B	10/13	4/46	31/54	1/10	0/14	35/67	9/2894	62/127	3/34	43/63	9/10	14/82	11/31	2/5	22/34	18/160	23/56	0/10	1/14	1/14
deepseek-coder-instruct	33B	12/13	0/47	46/58	4/10	0/14	66/68	52/2888	81/126	8/32	45/63	10/11	3/81	6/31	2/4	19/34	6/161	23/50	0/14	0/14	1/14
star-coder-instruct	15B	8/12	0/46	27/54	2/7	3/14	36/51	10/2887	67/124	12/26	1/63	6/8	4/74	7/27	1/4	20/34	6/149	23/49	0/12	0/14	1/14

[1] Denominator represents the total number of failures.

Table 8: Proportion of Verification Failures Caused by Syntax Errors

C Example Specifications in FM-ALPACA and FM-BENCH

The examples of the five formal specification languages are shown in Figure 4.

D Collected Repositories

We listed the repositories that were collected for data construction in the following. Note that one can easily add more repositories into FM-ALPACA and FM-BENCH.

For ACSL:

- `https://github.com/manavpatnaik/frama-c-problems`
- `https://github.com/fraunhoferfokus/acsl-by-example`

For TLA+:

- `https://github.com/tlaplus/Examples`

For Lean4:

- `https://github.com/leanprover/lean4`

For Coq:

- `https://github.com/coq/coq`

For Dafny:

- `https://github.com/vladstejeroiu/Dafny-programs`

E Complete Evaluation Result

The Pass@1 and Pass@5 are shown in Table 9. It is a completed version of Table 3.

F Prompt Design

We listed the prompts that are used for data preparation and inference in the following. For **data preparation**, as shown in Figure 5, to generate descriptions for the given proof segments, the prompt template consists of five parts: (1) Role description, (2) Domain knowledge of TLA+, (3) Task description, (4) Few-shot examples (we show one example in the figure, while three-shots were used in RQ2), (5) The proof or proof segment to be summarized.

For the **inference**, for each task, we designed five different instructions to avoid overfitting. The prompts for each task are shown in Figure 6 $\sim$ Figure 10. For each task, we first randomly choose one instruction and concat the inputs.

Coq

```coq
Proof.
  intros EM R H.
  destruct (EM (R true false)).
  - exists (fun _ => true).
    intros []; firstorder.
  - exists (fun b => b).
    intro b. split.
    + reflexivity.
    + destruct b, y; intros HR;
    easy || now symmetry in HR.
Qed.
```

Lean4

```lean
theorem gcd_dvd_left {a b : Int} : (gcd a b : Int) ∣ a := by
  have := Nat.gcd_dvd_left a.natAbs b.natAbs
  rw [← Int.ofNat_dvd] at this
  exact Int.dvd_trans this natAbs_dvd_self

theorem gcd_dvd_right {a b : Int} : (gcd a b : Int) ∣ b := by
  have := Nat.gcd_dvd_right a.natAbs b.natAbs
  rw [← Int.ofNat_dvd] at this
  exact Int.dvd_trans this natAbs_dvd_self
```

ACSL

```c
#include <stdio.h>


/*@
      requires a >= 0 && b >= 0;
      ensures \result == \old(a)*\old(b);
      assigns \nothing;
*/
int mul(int a, int b) {
      int x = a, y = b, prod = 0;
      /*@
            loop invariant prod == (a-x)*y;
            loop assigns prod, x;
      */
      while(x >= 0) {
            prod = prod + y;
            x--;
      }
      return prod;
}


int main() {
      int pdt = mul(2, 5);
      //@ assert pdt == 10;
}
```

Dafny

```dafny
method factorial(n: nat) returns (res: nat)
      ensures res == fact(n)
{
      var i := 1;
      res := 1;
      while i < n + 1
            invariant 0 < i <= n + 1
            invariant res == fact(i - 1)
      {
            res := i * res;
            i := i + 1;
      }
}
```

TLA+

```tla
PickDifferentColor ==
      /\ BeanCount > 1
      /\ can.black >= 1
      /\ can.white >= 1
      /\ can' = [can EXCEPT !.black = @ - 1]

TerminationHypothesis ==
      IF can.white % 2 = 0
      THEN <>(can.black = 1 /\ can.white = 0)
      ELSE <>(can.black = 0 /\ can.white = 1)
```

Figure 4: Formal Specification Languages in FM-bench

Pass @ 1

LLMs	Size	ProofGen						SegGen						ProofComplete				ProofInfill						Cd2Prf	
		TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	Dafny	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	ACSL	ACSL-WP
W/o Fine-tuning																									
llama3.1-instruct	8B	0.00	4.26	1.69	0.00	14.29	0.00	1.43	1.19	6.06	8.33	0.00	0.00	0.00	3.57	5.56	28.57	0.00	4.85	7.69	13.33	21.43	0.00	57.14	0.00
llama3.1-instruct	70B	7.69	6.38	10.17	20.00	7.14	0.00	24.29	4.86	13.64	11.11	39.68	3.17	27.27	3.57	13.89	28.57	0.00	7.27	24.62	13.33	21.43	0.00	21.43	0.00
qwen2.5-instruct	7B	0.00	2.13	5.08	0.00	21.43	0.00	1.43	1.05	7.58	5.56	1.59	1.59	0.00	13.10	19.44	28.57	0.00	5.45	18.46	13.33	21.43	0.00	14.29	0.00
qwen2.5-instruct	72B	7.69	6.38	18.64	20.00	21.43	7.14	12.86	3.29	12.12	25.00	12.70	4.76	0.00	11.90	33.33	28.57	0.00	9.70	24.62	26.67	35.71	0.00	57.14	0.00
qwen2.5-coder-instruct	7B	0.00	2.13	6.78	0.00	0.00	0.00	2.86	2.14	12.12	11.11	3.17	0.00	0.00	4.76	11.11	14.29	0.00	3.64	23.08	20.00	0.00	0.00	0.00	0.00
qwen2.5-coder-instruct	32B	0.00	6.38	11.86	0.00	21.43	0.00	12.86	4.07	9.85	25.00	26.98	4.76	27.27	5.95	36.11	42.86	2.94	7.88	33.85	20.00	71.43	14.29	57.14	14.29
star-coder-instruct	15B	7.69	2.13	8.47	30.00	28.57	0.00	27.14	1.97	6.06	27.78	0.00	0.00	27.27	11.90	25.00	42.86	0.00	9.70	24.62	20.00	35.71	0.00	28.57	0.00
deepseek-coder-instruct	7B	0.00	2.13	8.47	0.00	0.00	0.00	4.29	1.73	3.79	5.56	0.00	0.00	9.09	2.38	13.89	28.57	0.00	3.03	13.85	33.33	0.00	0.00	0.00	0.00
deepseek-coder-instruct	33B	0.00	0.00	1.69	0.00	0.00	0.00	2.86	1.94	4.55	11.11	3.17	0.00	0.00	3.57	13.89	42.86	0.00	2.42	23.08	6.67	7.14	0.00	7.14	0.00
deepseek-r1	xxB	30.77	8.51	30.51	30.00	35.71	7.14	22.86	10.70	21.21	22.22	49.21	3.17	45.45	15.48	33.33	28.57	5.88	9.70	24.62	6.67	42.86	21.43	64.29	14.29
W/ Fine-tuning																									
llama3.1-sft	8B	0.00	6.38	8.47	20.00	0.00	0.00	38.57	29.58	21.97	25.00	88.89	17.46	36.36	8.33	11.11	28.57	0.00	0.00	7.69	0.00	21.43	0.00	21.43	0.00
llama3.1-ultrachat	8B	0.00	6.38	3.39	0.00	0.00	0.00	0.00	0.00	3.79	0.00	0.00	0.00	0.00	5.95	13.89	14.29	0.00	4.85	9.23	0.00	14.29	7.14	0.00	0.00
llama3.1-ultrachat-fm	8B	0.00	8.51	10.17	30.00	0.00	0.00	41.43	35.79	29.55	33.33	95.24	17.46	18.18	8.33	11.11	28.57	2.94	0.00	7.69	0.00	28.57	0.00	35.71	0.00
llama3.1-tulu	8B	0.00	2.13	1.69	0.00	0.00	0.00	0.00	0.71	6.06	0.00	0.00	0.00	0.00	5.95	8.33	28.57	0.00	1.82	10.77	0.00	7.14	0.00	14.29	0.00
llama3.1-tulu-fm	8B	0.00	4.26	11.86	30.00	7.14	0.00	42.86	36.64	27.27	36.11	98.41	17.46	18.18	10.71	11.11	42.86	2.94	2.42	6.15	0.00	35.71	0.00	50.00	0.00
qwen2.5-sft	7B	0.00	4.26	11.86	10.00	0.00	0.00	38.57	27.95	27.27	22.22	87.30	17.46	27.27	9.52	13.89	28.57	0.00	0.00	7.69	0.00	21.43	0.00	14.29	0.00
qwen2.5-coder-sft	7B	0.00	6.38	18.64	20.00	7.14	0.00	44.29	36.50	34.09	33.33	98.41	17.46	36.36	9.52	16.67	28.57	5.88	1.82	13.85	13.33	42.86	0.00	42.86	0.00
deepseek-coder-sft	7B	0.00	2.13	16.95	0.00	0.00	0.00	34.29	25.30	31.06	25.00	84.13	15.87	36.36	8.33	19.44	28.57	0.00	1.82	7.69	6.67	14.29	7.14	14.29	7.14

Pass @ 5

LLMs	Size	ProofGen						SegGen						ProofComplete				ProofInfill						Cd2Prf	
		TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	TLA	Coq	Lean	ACSL-WP	TLA	Coq	Lean	Dafny	ACSL	ACSL-WP	ACSL	ACSL-WP
W/o Fine-tuning																									
llama3.1-instruct	8B	0.00	2.13	5.08	0.00	35.71	0.00	4.29	1.90	8.33	11.11	3.17	0.00	0.00	3.57	8.33	28.57	0.00	7.88	20.00	20.00	42.86	0.00	78.57	7.14
llama3.1-instruct	70B	7.69	8.51	13.56	20.00	35.71	21.43	27.14	6.01	17.42	13.89	52.38	4.76	36.36	11.90	27.78	28.57	0.00	10.30	30.77	20.00	35.71	7.14	50.00	14.29
qwen2.5-instruct	7B	0.00	2.13	5.08	0.00	28.57	0.00	2.86	1.39	9.09	5.56	3.17	3.17	0.00	16.67	22.22	28.57	0.00	5.45	20.00	20.00	42.86	0.00	42.86	0.00
qwen2.5-instruct	72B	15.38	6.38	22.03	30.00	21.43	7.14	17.14	3.80	14.39	25.00	15.87	4.76	9.09	16.67	38.89	28.57	0.00	11.52	26.15	33.33	57.14	14.29	92.86	7.14
qwen2.5-coder-instruct	7B	0.00	4.26	10.17	0.00	0.00	0.00	5.71	2.99	13.64	11.11	4.76	0.00	0.00	11.90	19.44	28.57	0.00	4.24	30.77	20.00	7.14	0.00	0.00	0.00
qwen2.5-coder-instruct	32B	7.69	6.38	15.25	10.00	28.57	0.00	15.71	4.96	15.15	27.78	33.33	4.76	36.36	10.71	47.22	42.86	2.94	10.91	33.85	20.00	92.86	28.57	71.43	28.57
star-coder-instruct	15B	15.38	4.26	10.17	30.00	42.86	0.00	34.29	3.46	7.58	27.78	0.00	0.00	45.45	16.67	41.67	57.14	0.00	16.97	38.46	26.67	42.86	0.00	42.86	7.14
deepseek-coder-instruct	7B	7.69	2.13	10.17	0.00	0.00	0.00	8.57	2.82	5.30	5.56	4.76	0.00	18.18	9.52	16.67	28.57	0.00	4.24	18.46	33.33	21.43	0.00	21.43	0.00
deepseek-coder-instruct	33B	0.00	6.38	6.78	20.00	7.14	0.00	2.86	3.23	6.82	13.89	25.40	1.59	0.00	11.90	19.44	42.86	0.00	10.30	40.00	26.67	14.29	7.14	14.29	0.00
W/ Fine-tuning																									
llama3.1-sft	8B	0.00	6.38	15.25	20.00	0.00	0.00	41.43	34.57	26.52	27.78	90.48	17.46	36.36	9.52	11.11	28.57	2.94	0.00	13.85	0.00	35.71	0.00	57.14	0.00
llama3.1-ultrachat	8B	0.00	8.51	5.08	0.00	0.00	0.00	0.00	0.07	7.58	0.00	1.59	0.00	0.00	4.76	22.22	14.29	0.00	4.85	9.23	6.67	14.29	7.14	0.00	0.00
llama3.1-ultrachat-fm	8B	0.00	10.64	15.25	30.00	7.14	0.00	42.86	39.86	31.06	38.89	96.83	17.46	18.18	9.52	13.89	28.57	11.76	0.00	10.77	0.00	42.86	7.14	64.29	7.14
llama3.1-tulu	8B	0.00	6.38	1.69	0.00	0.00	0.00	0.00	1.43	6.06	0.00	3.17	0.00	0.00	13.10	30.56	28.57	0.00	6.67	21.54	13.33	21.43	7.14	21.43	0.00
llama3.1-tulu-fm	8B	0.00	6.38	11.86	30.00	7.14	0.00	44.29	40.20	32.58	38.89	98.41	17.46	18.18	13.10	11.11	42.86	2.94	4.24	9.23	0.00	71.43	7.14	64.29	0.00
qwen2.5-sft	7B	0.00	4.26	13.56	10.00	7.14	0.00	40.00	32.77	30.30	27.78	88.89	17.46	45.45	10.71	19.44	28.57	2.94	1.21	12.31	0.00	42.86	0.00	35.71	0.00
qwen2.5-coder-sft	7B	0.00	8.51	22.03	30.00	7.14	0.00	44.29	40.81	35.61	36.11	98.41	17.46	54.55	10.71	19.44	42.86	8.82	4.24	20.00	13.33	64.29	0.00	57.14	0.00
deepseek-coder-sft	7B	0.00	4.26	16.95	0.00	0.00	0.00	35.71	30.70	36.36	25.00	85.71	15.87	54.55	13.10	16.67	28.57	2.94	4.24	9.23	6.67	14.29	7.14	28.57	7.14

Table 9: Full Experiment Results on Pass@1 and Pass@5

Prompt for Description Generation. Take TLA+ as an example.

 # Role description

As an expert in TLA+, you are good at understanding and writing TLA+. TLA+ is a formal specification language used for modeling and verifying concurrent and distributed systems.

 # Domain knowledge

1. The logical operators supported by TLA+ include: /\ (and), \/ (or), ~ (not), => (Implication), <=> (Bidirectional implication), `TRUE`, `FALSE`, \A (Universal Quantification), \E (Existential Quantification)
2. The sets operators supported by TLA+ include: = (Equality), # (not equal), \union (Union), \intersect (Intersection), \in (Membership), \notin (Not in), \subseteq (Subset Equal), \(Difference).
3. The temporal Operators supported by TLA+ include `[] x > 0`, which is an example of [] (Always). It means that at all times, the value of variable x is greater than 0. `<> x = 0` is an example of <> (Eventually). It means that at some point in time, the value of variable x becomes 0.
4. Built-in keywords and operators in TLA+ include: 'MODULE, 'EXTENDS', 'CONSTANT', 'INSTANCE', 'VARIABLE', 'ASSUME', 'PROVE', 'INIT', 'NEXT', 'ACTION', 'SPECIFICATION', 'IF', 'ELSE', 'WITH', 'CASE', 'THEN', 'LET', 'IN', 'CHOOSE', 'ENABLED', 'UNCHANGED', 'DOMAIN'.

 # Task description

Given a TLA+ code snippet, you need to summarize the given TLA+ in several sentences in detail.

 # Example Input and Output

Code

```
Return(c,S) ==
/\ S # {} /\ S \subseteq alloc[c]
/\ alloc' = [alloc EXCEPT ![c] = \ S]
/\ UNCHANGED unsat
```

Description:

An operation 'Return(c,S)' that represents the return of a set of resources by a client. It satisfies the following conditions:
- The set 'S' is not empty and 'S' must be a subset of the set of allocated resources to the client 'c'.
- The 'alloc' is updated by assigning the difference between the current set of allocated resources and the set 'S' to the 'c' index of 'alloc' except 'c'.
- The 'unsat' remains unchanged.

 # Code to be described:

<A proof segment>

Figure 5: Prompt for generating TLA+ description. The prompt templates for other formal specification languages are in the same structure.

Prompt for SegGen Task (lang: a placeholder to be replaced by each formal specification language name.)

 # Task Description (SegGen)
1. Translate the given natural language into {lang} syntax.
2. Model the intention written in natural language using {lang}.
3. Express the requirement using {lang}.
4. Model the given natural language into {lang}.
5. Translate the given requirement using {lang}'s syntax and semantics.

(Randomly choose one of the above.)

You only need to return the {lang} formal specification without explanation.

 # Input
<Input goes here>

Figure 6: Prompt for SegGen Task.

Prompt for ProofGen Task (lang: a placeholder to be replaced by each formal specification language name.)

 # Task Description (ProofGen)
1. Translate the given requirements into lang syntax.
2. Model the given requirements written in natural language using lang.
3. Express the requirements using lang.
4. Model the given requirements written in natural language into lang.
5. Translate the given requirements into lang's syntax and semantics.

(Randomly choose one of the above.)

(For ACSL): You only need to return the lang formal specification with the code without explanation.
(For others): You only need to return the lang formal specification without explanation.

 # Input
<Input goes here>

Figure 7: Prompt for ProofGen Task.

Prompt for ProofComplete Task (lang: a placeholder to be replaced by each formal specification language name.)

Task Description (ProofComplete)
1. Please complete the following formal proof in formal specification language lang according to the given requirement.
2. Please complete the following formal proof in lang according to the given requirement.
3. Please complete the given formal proof in lang following the requirement below.
4. Please complete the following formal proof in lang according to the requirement below.
5. Please complete the following formal proof in lang according to the given requirement.

(Randomly choose one of the above.)

You only need to return the completed lang formal specification (together with the provided formal specification) without explanation.

Input
<Input goes here>

Figure 8: Prompt for ProofComplete Task.

Prompt for ProofInfill Task (lang: a placeholder to be replaced by each formal specification language name.)

Task Description (ProofInfill)
1. Please fill in the [MASK] in the following formal proof in formal specification language lang according to the given requirement.
2. Please fill in the [MASK] in the following formal proof in lang according to the given requirement.
3. Please complete the given formal proof in lang following the requirement below by filling in the [MASK].
4. Please fill in the [MASK] in the following formal proof in lang according to the requirements below.
5. Please fill in the [MASK] in the following formal proof in lang according to the given requirement.

(Randomly choose one of the above.)

You only need to return the completed lang formal specification (together with the provided formal specification) without explanation.

Input
<Input goes here>

Figure 9: Prompt for ProofInfill Task.

Prompt for Code2Proof Task

Task Description (Code2Proof)

1. Please fill in the [MASK] in ACSL according to the given requirement and ACSL specification.
2. Please fill in the [MASK] in ACSL according to the given requirement.
3. Please fill in the [MASK] in ACSL according to the given ACSL specification.
4. Please fill in the [MASK] in ACSL according to the given requirement and ACSL specification.
5. Please infill the [MASK] in ACSL according to the given requirement.

(Randomly choose one of the above.)

You only need to return the completed ACSL formal specification (together with the provided formal specifications and C programs) without explanation.

Input

<Input goes here>

Figure 10: Prompt for Code2Proof Task.