Evaluating LLM-Generated ACSL Annotations for Formal Verification

Arshad Beg
Maynooth University
Ireland
arshad.beg@mu.ie

Diarmuid O'Donoghue
Maynooth University
Ireland
diarmuid.odonoghue@mu.ie

Rosemary Monahan
Maynooth University
Ireland
rosemary.monahan@mu.ie

Abstract

Formal specifications are crucial for building verifiable and dependable software systems, yet generating accurate and verifiable specifications for real-world C programs remains challenging. This paper empirically evaluates the extent to which formal-analysis tools can automatically generate and verify ACSL specifications without human or learning-based assistance. We conduct a controlled study on a recently released dataset of 506 C programs, repurposing it from interactive, developer-driven workflows to an automated evaluation setting. Five ACSL generation systems are compared: a rule-based Python script, Frama-C's RTE plugin, and three large language models–DeepSeek-V3.2, GPT-5.2, and OLMo 3.1 32B Instruct. All generated specifications are verified under identical conditions using the Frama-C WP plugin powered by multiple SMT solvers, allowing a direct comparison of annotation quality, solver sensitivity, and proof stability. Our results provide new empirical evidence on the capabilities and limitations of automated ACSL generation, complementing prior survey-based work.

Keywords: Static Program Analysis, ACSL Annotation, Runtime Error Detection, Specification Automation

1 Introduction

Over the past decade, artificial intelligence has advanced rapidly. Large models now perform impressively in language understanding, vision, and decision-making. Yet these systems remain difficult to trust. Their behaviour is often opaque, failures can occur silently, and there are no built-in guarantees of safety or correctness—limitations that are especially problematic in high-assurance domains. Formal methods address this gap by offering mathematically grounded guarantees about system behaviour. They allow developers to express precise requirements and prove that software satisfies them. However, formal methods are costly to apply in practice. Writing and maintaining specifications is challenging, proofs scale poorly, and purely symbolic techniques struggle with modern and evolving codebases.

This paper empirically evaluates the extent to which formal analysis tools can automatically generate and verify ACSL specifications for C programs without human or learning-based assistance. This work differs fundamentally from our prior study in [2] in both dataset and methodology. We evaluate a new dataset released in November 2025, consisting of 506 C programs [6], which is commonly used in interactive settings where developers rely on Gemini-based prompt feedback loops to iteratively correct programs or annotations [10]. In contrast, we re-purpose this dataset and conduct a controlled comparison of five ACSL generation strategies: a rule-based Python script, Frama-C's RTE plugin, and three large language models–DeepSeek-V3.2, GPT-5.2, and OLMo 3.1 32B Instruct. In the paper, we refer to these models as DeepSeek, GPT-5, and OLMo3, respectively, without using their full names. All generated specifications are verified under identical conditions using the Frama-C WP plugin and multiple SMT solvers. This design enables a direct comparison between tool generated and LLM generated ACSL on a modern dataset, isolating the impact of annotation quality, solver sensitivity, and proof stability, and providing new empirical evidence that complements—but does not overlap with—our earlier survey-based work.

The structure of the paper is as follows: Most recent literature and its analysis is presented in Section 2. Section 3 is about the experimental setup and the workflow we adopted to conduct the research. Empirical evaluation of our experiments is presented in Section 4 and Section 5 concludes the paper.

2   Related Work

Our earlier workshop paper [1] examined the challenges of translating informal natural-language requirements into formal specifications. It highlighted key sources of difficulty, including ambiguity, missing domain knowledge, contextual gaps, and instability in large language model outputs. That work introduced the VERIFYAI framework, which integrates LLMs with NLP pipelines, ontology-based modelling, and artefact reuse to support semi-automated formalisation. It also compared early systems such as Req2Spec, SpecGen, AssertLLM, and nl2spec, analysing how they represent requirements and interface with verification tools. An empirical evaluation using Frama-C PathCrawler and several SMT solvers exposed practical limits in verification scalability and solver robustness.

These ideas were substantially extended in [2], a large-scale survey of over one hundred studies on AI-enabled requirements formalisation, traceability, and verification. The survey mapped the broader research landscape across programming languages, specification notations, and verification backends, including both theorem provers and model checkers. It identified recurring methodological patterns, from rule-based extraction to hybrid neuro-symbolic pipelines, and highlighted unresolved ecosystem challenges such as weak toolchain integration, limited traceability, and poor reuse of existing specification artefacts. In addition, the study reported an expanded empirical evaluation of Frama-C's EVA, RTE, and PathCrawler tools, revealing recurring issues related to solver instability, path coverage, and configuration sensitivity. Together, these prior works provide the methodological and empirical foundation for the focused analysis presented in this paper.

PALM, introduced in [8], is a generate-and-repair framework that integrates large language models with symbolic reasoning to enhance formal proof generation in Coq. Analyzing 520 proof-generation errors by GPT-3.5, Minghai et al. observed that while LLMs often capture the overarching structure of proofs, they frequently fail at low-level steps. This insight motivated PALM's iterative repair mechanism. Tested on a dataset exceeding 10,000 theorems, PALM demonstrates a 76.6%-180.4% improvement in success rates, proving an additional 1,270 theorems over previous methods and showing strong generalizability across multiple LLM architectures.

The framework AutoSpec, presented in [13], automates the synthesis of formal specifications to enable end-to-end program verification with minimal manual intervention. Unlike earlier approaches constrained to narrow domains, AutoSpec handles complex program constructs, including arrays, pointers, nested loops, and function calls. The framework employs an iterative synthesis loop that combines static analysis and program verification, progressively refining candidate specifications while ensuring both satisfiability and

proof adequacy. Empirical evaluation indicates robust performance, successfully verifying 79% of benchmark programs, a 1.592 times improvement over prior techniques and further validating its practicality by verifying the real-world X509-parser project.

In [3], the authors assessed GPT-4o's capability to generate verifiable specifications for C programs using VeriFast, a separation-logic-based static verifier. Their experiments, which varied user inputs and prompting strategies, revealed that GPT-4o could preserve functional behavior in specifications; however, the generated specifications often failed formal verification and contained redundancies. Similarly, [16] introduces OntoChat, a conversational agent leveraging LLMs to support the Ontology Requirements Engineering (ORE) process. OntoChat addresses limitations of manual ORE methods by facilitating automated and enhanced requirements elicitation, documentation, and validation. Preliminary findings from the first year indicate that LLM assisted interactions can improve efficiency, consistency, and collaboration in ontology engineering.

HILBERT, detailed in [11], is an integrated agentic framework combining informal mathematical reasoning with formal verification to advance automated theorem proving. The system coordinates four synergistic components: an informal reasoning model for high-level insights of Lean 4, which is an optimised prover, a formal verifier, and a semantic theorem retriever. For unresolved problems, HILBERT decomposes them recursively into smaller subgoals, addressed via either the prover or the informal reasoning agent. Verifier-guided feedback loops iteratively correct and reinforce proofs, maintaining semantic consistency. Experimental results demonstrate state-of-the-art performance: 99.2% accuracy on miniF2F, 6.6 percentage points higher than prior best methods and 462/660 problems (70.0%) solved on PutnamBench, surpassing proprietary systems like SeedProver (50.4%) and improving 422% over the best publicly available baseline. These outcomes establish HILBERT as a significant advancement in bridging informal reasoning and formally verified proof synthesis.

LEMUR, introduced in [14], is a framework that combines automated reasoning with structured synthesis to improve loop invariant generation and program verification. Unlike prior learning-based methods such as Code2Inv, which rely purely on reinforcement learning, LEMUR formalizes a unified calculus integrating symbolic reasoning and guided invariant inference. On the Code2Inv benchmark set of 133 C programs, LEMUR uses the ESBMC k-induction verifier to validate invariants, significantly outperforming ESBMC alone and Code2Inv. Specifically, LEMUR (GPT-4) solves 107 benchmarks within a 10-minute timeout, compared to 68 for ESBMC and fewer for Code2Inv under a one-hour limit. Its adaptive generation strategy typically converges in four iterations, though complex cases may take longer. Comparative results with LEMUR (GPT-3.5) highlight that stronger

symbolic reasoning oracles lead to faster convergence and higher verification success, emphasizing the value of structured reasoning over heuristic generation.

RvLLM, described in [15], provides a runtime verification framework that exploits domain-specific knowledge encoded in a custom specification language (ESL) to systematically detect and correct errors. Operating in two stages, interpretation and reasoning, the framework uses context-driven interpretations to identify inconsistencies and iterative follow-up queries to ensure output consistency. Evaluations across three representative tasks—violation detection under the Singapore Rapid Transit Systems Act, numerical comparison, and inequality solving—tested multiple LLMs including Qwen (max, plus, turbo, 2.5 variants), GPT-4.1 (mini, nano), Gemini 2.0 Flash (Lite), and DeepSeek-V3. In violation detection, RvLLM substantially improved true positive rates, raising Qwen max from 56.2% to 86.1% and GPT-4.1 from 57.7% to 81.1%, while maintaining true negative rates. In numerical comparison, LLMs guided by RvLLM consistently produced correct or inconclusive results, e.g., Qwen 2.5 (32B) yielded 98 correct and 2 inconclusive outputs in 4.98 seconds. In inequality solving tasks involving factorization, interval analysis, and endpoint checking, RvLLM improved true positive rates, achieving 50% for Qwen 2.5 on factorization, demonstrating effective enforcement of domain-specific constraints.

The most recent works of November 2025 have following contribution: [10] introduces VeCoGen, a tool that combines large language models with formal verification to automatically generate correct C programs from formal and natural language specifications along with test cases. VeCoGen generates candidate programs and iteratively refines them until a program satisfies the formal specification, ensuring correctness. The approach is evaluated on 15 Codeforces problems, successfully solving 13, demonstrating the feasibility of integrating LLMs with formal methods for program generation. This work highlights the potential of leveraging LLMs to produce formally verified code suitable for safety-critical applications. [6] addresses the lack of datasets for benchmarking verified C code generation by presenting CASP, a curated dataset of 506 C programs paired with formally verified ACSL specifications. The dataset is constructed through multi-stage filtering, formal verification using Frama-C, LLM-assisted improvements, and manual inspection to ensure correctness. CASP enables systematic evaluation of automated code generation methods against verified specifications. This contribution provides a foundation for research on integrating LLM-based code generation with formal verification.

SpecVerify, presented in [12], integrates LLMs with formal verification tools to automatically extract and verify properties from natural language requirements. By combining Claude 3.5 Sonnet with the ESBMC verifier, SpecVerify

achieves verification accuracy comparable to NASA's CoCoSim, while reducing false positives and extending the expressiveness of assertions beyond traditional logics. Complementing this, [7] explores methods for automatically discovering inductive loop invariants, a long-standing formal verification challenge. Using a benchmark of 1,025 C programs with diverse loops, GPT-4, GPT-3.5, and Code Llama were paired with Frama-C's WP tool and SMT solvers to evaluate invariant soundness. The approach synthesizes candidate invariants through data-driven methods and validates them formally via automated proof obligations.

SV-LLM, introduced in [9], is a multi-agent LLM framework designed for automating security verification of complex system-on-chip (SoC) designs. The framework deploys domain-specialized agents to perform tasks such as asset identification, threat modeling, property generation, vulnerability detection, and simulation-based bug validation. Agents employ diverse learning paradigms—including fine-tuning, in-context learning, and retrieval-augmented generation—to produce precise, verifiable security specifications from design data and documentation, with reasoning capabilities tailored to RTL semantics and design constraints. Experimental evaluation shows substantial gains: the fine-tuned Security Vulnerability Detection Agent (Mistral-7B-Instruct) achieved 84.8% accuracy, a 42.3 percentage-point improvement over its non-fine-tuned baseline, while the Bug Validation Agent reached up to 89% validated testbench generation, outperforming zero-shot prompting by more than fourfold. These results demonstrate SV-LLM as a scalable, explainable, and domain-adapted solution for SoC security verification. Finally, [5] explores integrating Copilot with formal methods through an IDE equipped with language servers. Granberry et al. [4] also studied combining LLMs with symbolic analysis for C program specification generation, enhancing LLM prompts using outputs from PathCrawler and EVA to produce ACSL annotations.

3   Experimental Setup and Workflow

The Frama-C platform is configured and executed within a Linux environment running on Windows Subsystem for Linux (WSL 2), hosted on a Windows 11 system equipped with an Intel Core Ultra 5 125U processor (3.60 GHz) and 32 GB of RAM (31.5 GB usable) operating under a 64-bit architecture. We acquired the dataset available on URL, https://huggingface.co/datasets/nicher92/CASP_source_files from [6]. The C programs are first cleaned to get pure C files. It was found that out of 506 files, there were two stacks: one of 355 C files and other of 151 files. Once filtered out, second stack contained empty or incomplete C files. So, we selected

the first stack for our experimentation. The adopted workflow is depicted in Figure 1, which summarizes the sequential transformation from a curated dataset to pure and annotated C programs, the application of multiple ACSL generation techniques, the execution of weakest-precondition prover tests, and the subsequent analysis of the obtained verification results. The sources and logs of our experiments are available publicly at https://github.com/arshadbeg/FTfJPatECOOP2026.

4 Empirical Evaluation

4.1 EVA Static Analysis

We first performed a static analysis on these pure C files through Frama-C EVA tool. Table 1 provides a summary of results from EVA, augmented with the figure 2. Although EVA successfully analysed only 40.8% of the files, we achieved a median statement coverage of 81%, with alarm generation remaining sparse across the dataset.

Table 1. EVA Dataset Summary

Dataset	Files	Mean coverage (%)	Median coverage (%)	Mean alarms	Max alarms	Kernel warnings
All files	355	81.41	81.0	0.12	3	19
Successful	145	81.41	81.0	0.29	3	19
Failed	210	–	–	0.00	0	0

Figure 2 provides a visual overview of the EVA analysis results. Figure 2a shows the distribution of alarms per file, highlighting that most files generate few or no alarms. Figure 2b compares statement coverage between successful and failed analyses, indicating that failed analyses do not have coverage data. Figure 2c examines the relationship between kernel warnings and statement coverage, illustrating that files with more warnings tend to have lower coverage.

4.2 Analysis based on Mean Values

Table 2 summarises the aggregate WP verification statistics across all ACSL generation strategies, providing a quantitative complement to the visual trends in Fig. 3. Tool-generated specifications, particularly those produced by the Python script and the RTE plugin, achieve the highest mean proof success rates while maintaining low timeout counts, confirming their reliability for automated verification. Among LLM-based approaches, DeepSeek exhibits the strongest overall performance, with mean success and timeout figures approaching those of RTE, whereas GPT and OLMo3 show reduced success rates and substantially higher solver timeouts, especially for Alt-Ergo. Notably, the median proof success reaches 100% for all approaches, indicating that when proofs succeed, they often do so completely; however, the variability captured by the mean values and timeout counts highlights significant differences in specification robustness. Overall, Table 2 reinforces that ACSL generation quality must

Table 2. WP verification results for ACSL annotations generated by different sources. Mean success denotes the average percentage of proved goals over generated goals. Qed time is reported in milliseconds.

Source	Solver	Runs	Mean	TO	Qed
Python Script	AltErgo	207	71.93	278	1.25
	CVC4	205	72.23	27	1.16
	CVC5	203	72.19	33	1.16
	Z3	205	72.57	91	1.22
RTE	AltErgo	115	99.15	6	7.50
	CVC4	113	99.12	0	8.50
	CVC5	111	99.09	0	7.50
	Z3	113	99.12	4	6.50
DeepSeek	AltErgo	149	94.89	130	6.17
	CVC4	147	93.76	13	6.53
	CVC5	145	93.80	23	6.32
	Z3	147	94.56	22	5.04
GPT	AltErgo	177	80.98	272	1.86
	CVC4	175	81.63	8	2.00
	CVC5	173	81.90	10	2.30
	Z3	175	82.07	20	1.97
OLMo3	AltErgo	134	83.14	92	2.92
	CVC4	133	83.14	0	2.81
	CVC5	132	83.14	0	3.05
	Z3	133	83.14	77	2.87

be evaluated not only by proof success but also by solver stability and efficiency.

Figure 3 presents a consolidated comparison of Frama-C WP verification performance across ACSL specifications generated by the Python script, Frama-C RTE, and the three LLMs (DeepSeek, GPT, and OLMo3). As shown in Fig. 3a, tool-generated specifications (Python script and RTE) consistently achieve higher mean proof success rates with lower solver-dependent variability, reflecting more precise and verification friendly contracts. Among LLM-based approaches, DeepSeek demonstrates relatively strong performance with proof success close to tool-generated ACSL, whereas GPT and OLMo3 exhibit noticeably lower mean success and higher dispersion across solvers. This trend indicates that while LLMs can often produce syntactically valid ACSL, the semantic strength and completeness of the generated contracts vary substantially, directly impacting automated proof outcomes.

Solver robustness is further highlighted in Fig. 3b, which reports the total number of solver timeouts aggregated across
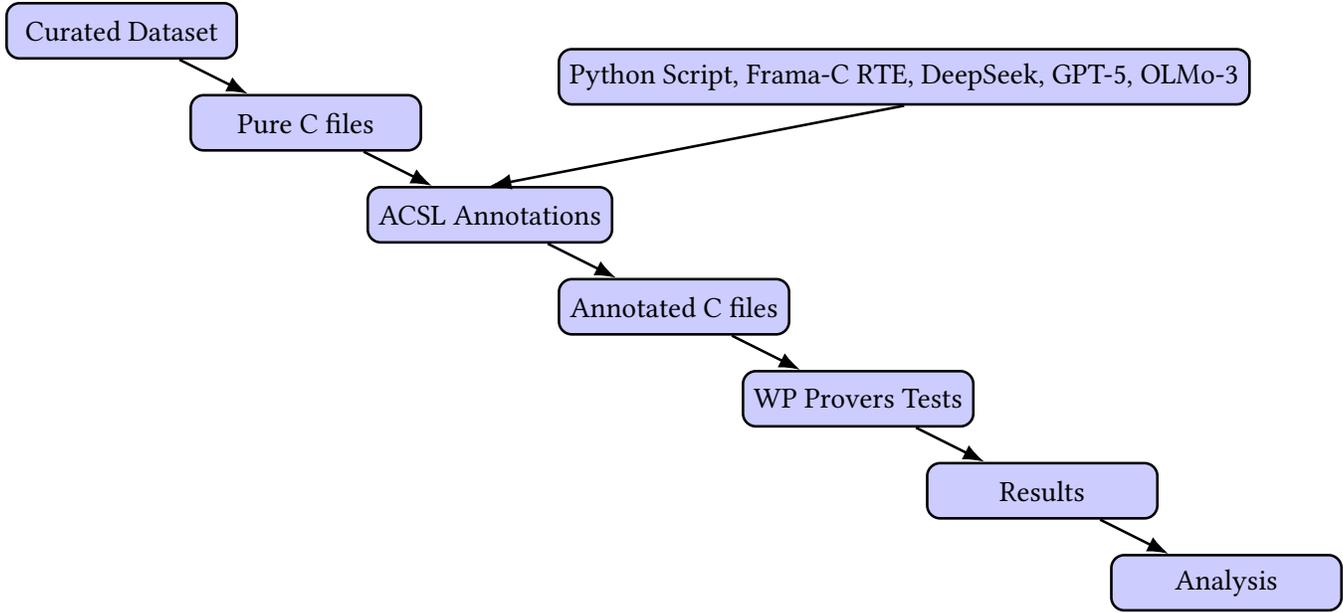
Figure 1. End-to-end research workflow illustrating how a curated dataset is progressively transformed into pure C files, enriched with ACSL annotations generated via a combination of automated scripts, Frama-C RTE, and large language models (DeepSeek-V3.2, GPT-5.2, and OLMo-3.1 32B Instruct), then consolidated into annotated C files that are evaluated using weakest-precondition (WP) prover tests to produce verification results that are finally subjected to systematic analysis.
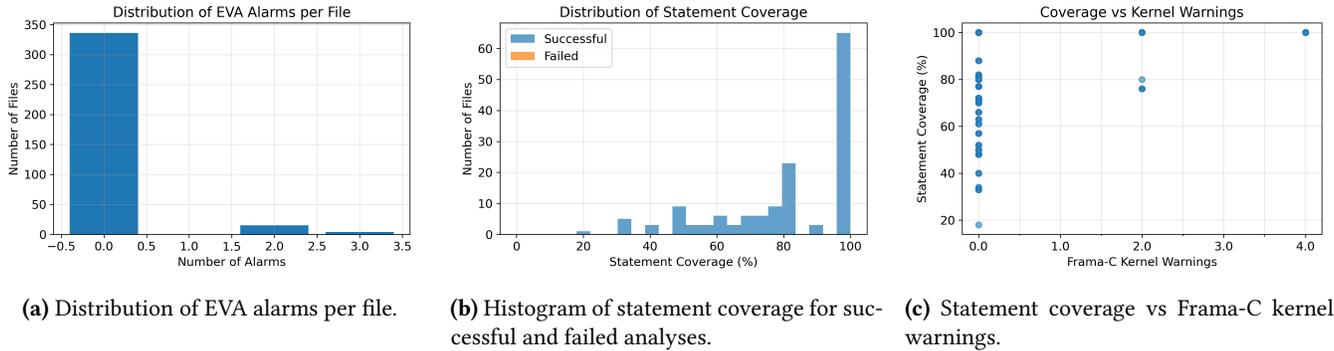

(a) Distribution of EVA alarms per file.

(b) Histogram of statement coverage for successful and failed analyses.

(c) Statement coverage vs Frama-C kernel warnings.

Figure 2. Visual summary of EVA analysis results. (a) Number of alarms per file. (b) Distribution of statement coverage for successful and failed analyses. (c) Relationship between kernel warnings and coverage.

all files. RTE-generated ACSL yields the fewest timeouts overall, confirming the conservative yet stable nature of runtime-error–driven specifications. In contrast, GPT-generated annotations result in the highest number of timeouts, particularly for Alt-Ergo, suggesting that overly weak or ambiguous specifications increase solver search space and proof difficulty. This trade-off is made explicit in Fig. 3c, where a clear inverse relationship emerges between proof success and timeout frequency across ACSL sources. The X-axis of the figure represents the different ACSL generation sources used to produce verification tasks, while the Y-axis denotes the SMT solvers applied to attempt proofs on these tasks.

The Z-axis shows the numerical values, with the lower portion of each bar indicating the mean proof success in percentage and the stacked portion on top representing the total number of timeouts. This arrangement allows a clear comparison of both solver effectiveness and robustness across different sources. Together, these results demonstrate that higher apparent proof success does not necessarily imply solver efficiency, and that specification quality must be assessed jointly through success rates and solver behavior, as summarized holistically in Fig. 3.
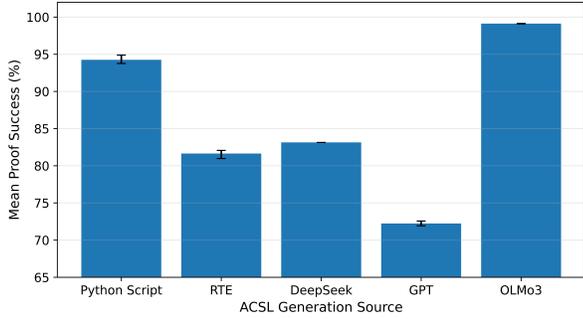
(a) Mean proof success rate with solver-dependent variability.
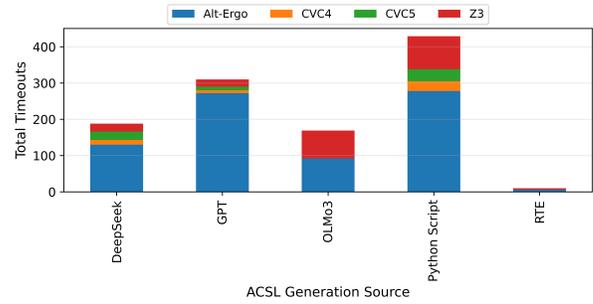
(b) Distribution of solver timeouts per ACSL source.
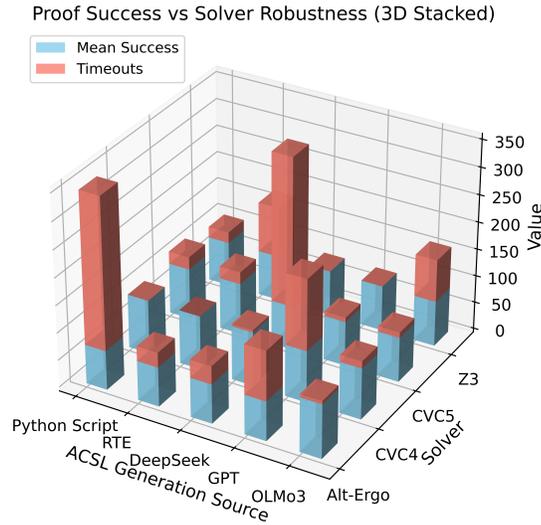
(c) Trade-off between proof success and solver robustness.

Figure 3. Comparison of WP verification outcomes across ACSL generation methods. The figure contrasts proof success, solver robustness, and their interaction for tool-generated and LLM-generated specifications.

4.3 Rule-Based Python Script Generation

Table 3 summarises the results of 452 WP runs distributed across four provers, with each solver executed between 111 and 115 times. For all solvers, the observed proof success values lie between 75% and 100%, indicating that, in every run, at least three quarters of the generated proof obligations were discharged automatically. The dispersion of these values is limited: Alt-Ergo exhibits a standard deviation of 4.56 over 115 runs, CVC4 and Z3 both show a standard deviation of 4.64 over 113 runs, and CVC5 reaches the highest observed variability with a standard deviation of 4.72 over 111 runs. Despite minor differences, the narrow spread and identical minimum and maximum values across solvers suggest that all four provers achieve comparable proof coverage on the evaluated benchmarks, with no solver consistently failing or excelling across the full set of verification tasks.

Differences between solvers become more apparent when considering timeout behavior and proof discharge times, as reported in Table 3. Alt-Ergo incurred 6 timeouts over 115 runs, while Z3 experienced 4 timeouts over 113 runs; in contrast, both CVC4 and CVC5 completed all runs without any observed timeouts. Mean Qed times range from 6.50ms for Z3 to 8.50ms for CVC4, with Alt-Ergo and CVC5 both averaging 7.50ms. Variability in discharge time also differs across solvers: Alt-Ergo and CVC5 show low standard deviations of 0.71ms, indicating stable performance across proof obligations, whereas CVC4 and Z3 exhibit higher variability with standard deviations of 2.12ms. These results indicate that, while proof success levels are similar across solvers, robustness and timing stability provide a clearer basis for differentiating prover behavior in practice.

Table 3. The table reports the number of runs, observed proof success range, standard deviation of proof success, timeout counts, and Qed-time statistics of ACSL augmented C files by rule-based Python Script

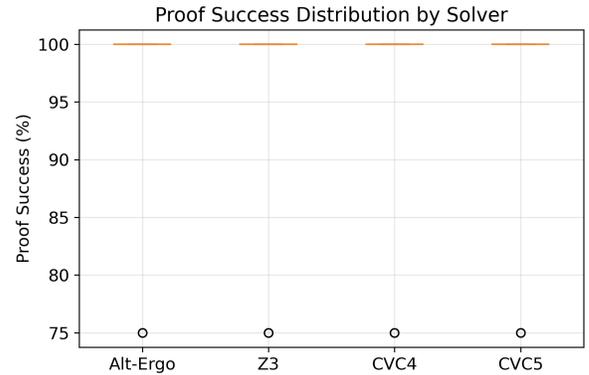| Solver	Runs	Min	Max	SD	TO	Qed	SD Qed
Alt-Ergo	115	75.0	100.0	4.56	6	7.50	0.71
CVC4	113	75.0	100.0	4.64	0	8.50	2.12
CVC5	111	75.0	100.0	4.72	0	7.50	0.71
Z3	113	75.0	100.0	4.64	4	6.50	2.12

Figure 4a presents a comparative analysis of Qed time variability, where the reported values correspond to the standard deviation of Qed times (in milliseconds) across four solvers, while Figure 4b presents the proof success distribution. Alt-Ergo and CVC5 exhibit similar variability profiles, with tightly clustered distributions and median standard deviation values around 7.5 ms, indicating relatively stable and consistent solver behavior. Z3 shows the lowest median standard deviation (approximately 6.5 ms), suggesting lower average variability; however, its wider interquartile range and extended whiskers indicate that variability fluctuates more significantly across instances, ranging from about 5 ms to 8 ms. In contrast, CVC4 demonstrates the highest median standard deviation (around 8.5 ms) and the largest overall spread, with values reaching up to roughly 10 ms, reflecting both higher variability and less predictable performance. Overall, the results highlight a trade-off between stability and dispersion across solvers: Z3 tends to exhibit lower central variability but with greater fluctuations, Alt-Ergo and CVC5 provide more consistent variability characteristics, and CVC4 shows the highest and most dispersed Qed time variability.

4.4 Frama-C RTE Generation

Table 4 summarises 820 WP runs performed on ACSL specifications generated by Frama-C RTE, with each solver executed between 203 and 207 times. For all provers, proof success ranges from 30% to 100%, indicating that some verification tasks remain only partially discharged, while others are fully proved automatically. The dispersion of proof success is nearly identical across tools, with standard deviations tightly grouped around 25.8 (from 25.80 for Alt-Ergo to 25.89 for Z3), reflecting a comparable variability over the benchmark set. More substantial differences arise in timeout counts: Alt-Ergo records 366 timeouts, considerably more than Z3 (142), CVC5 (48), and CVC4 (42). In contrast, Qed-related metrics remain close, with mean values between 1.16 and 1.25 and standard deviations below 1.2. Overall, while proof coverage and variability appear consistent across solvers, timeout frequency constitutes the primary differentiating factor in these experiments.
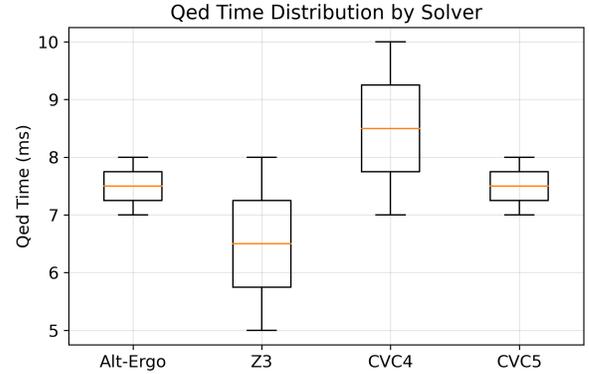
(a) Qed Time Distribution
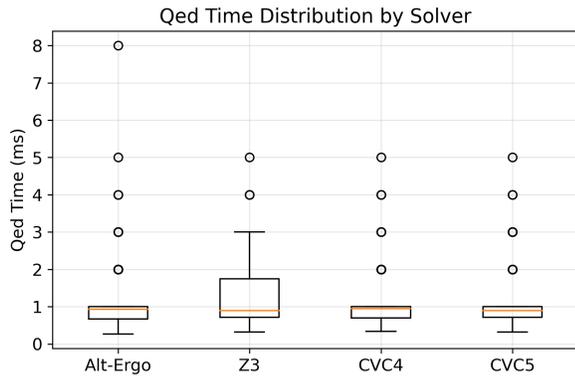

(b) Proof Success Distribution

Figure 4. Solver Performance for Rule-based Python Script

| Solver	Runs	Min	Max	SD	TO	Qed	SD Qed
Alt-Ergo	207	30.0	100.0	25.80	366	1.25	1.17
CVC4	205	30.0	100.0	25.85	42	1.16	0.87
CVC5	203	30.0	100.0	25.86	48	1.16	0.90
Z3	205	30.0	100.0	25.89	142	1.22	0.89

Table 4. WP verification results for ACSL specifications generated via Frama-C RTE

Figure 5a presents a comparative analysis of Qed time variability for ACSL specifications generated using the Frama-C RTE tool, where the reported values correspond to the standard deviation of Qed times (in milliseconds) across the four solvers, while Figure 5b presents the proof success distribution. Overall, all solvers exhibit very low Qed time variability, reflecting the lightweight nature of proof obligations produced by RTE instrumentation. CVC4 and CVC5 show the most stable behavior, with median standard deviation values slightly below 1 ms and tightly clustered distributions, indicating highly consistent proof discharge times across

verification runs. Z3 displays a comparable median variability (around 0.9 ms), though with a marginally wider interquartile range, suggesting slightly greater fluctuation across instances. Alt-Ergo exhibits the highest median standard deviation (approximately 1.17 ms) and a broader spread relative to the other solvers, indicating comparatively less stable Qed time behavior despite remaining within a low absolute range. Overall, the results highlight uniformly fast and stable Qed performance across solvers for RTE-generated ACSL, with CVC4 and CVC5 providing the most consistent profiles, Z3 showing moderate dispersion, and Alt-Ergo exhibiting the highest relative variability.

(a) Qed Time Distribution
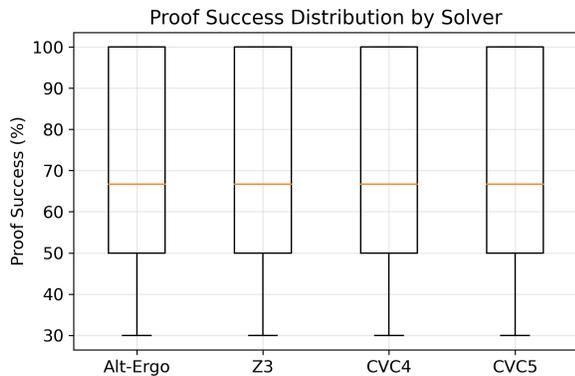
(b) Proof Success Distribution

Figure 5. Solver Performance for RTE-Generated ACSL

4.5 GPT-5.2 Generation

Table 5 presents 700 WP executions over ACSL specifications generated by GPT-5.2, with each solver run between 173 and 177 times. Proof success spans from 16.67% to 100% for all provers, revealing a broader range of partial discharges compared to fully successful runs. The variability of results is again similar across tools, with standard deviations between 27.34 and 28.15, indicating a consistently wide dispersion over the benchmark set. Timeout behaviour, however,

differs markedly: Alt-Ergo reports 272 timeouts, whereas Z3 records 20 and both CVC4 and CVC5 exhibit very few (8 and 10, respectively). Qed statistics show slightly higher averages than in the previous setting, ranging from 1.86 to 2.30, with relatively large standard deviations (up to 3.18), suggesting greater fluctuation in simplification effects. Overall, while proof coverage and dispersion remain comparable, timeout frequency and Qed variability distinguish solver performance in this configuration.

Solver	Runs	Min	Max	SD	TO	Qed	SD Qed
Alt-Ergo	177	16.67	100.0	28.15	272	1.86	2.88
CVC4	175	16.67	100.0	27.64	8	2.00	2.88
CVC5	173	16.67	100.0	27.34	10	2.30	3.18
Z3	175	16.67	100.0	27.42	20	1.97	2.60

**Table 5.** WP verification results for ACSL specifications generated by GPT-5.2

Figure 6a presents a comparative analysis of Qed time variability for ACSL specifications generated by the GPT-5 language model, where the reported values correspond to the standard deviation of Qed times (in milliseconds) across the four solvers, while Figure 6b presents the proof success distribution. In contrast to RTE-generated specifications, all solvers exhibit substantially higher variability, reflecting the increased complexity and heterogeneity of LLM-generated ACSL annotations. Z3 shows the lowest median standard deviation (approximately 2.6 ms), indicating comparatively more stable Qed performance, although its distribution still spans a wide range of values, suggesting non-uniform behavior across verification instances. Alt-Ergo and CVC4 exhibit similar median variability levels (around 2.9 ms), with broader interquartile ranges that point to less predictable proof discharge times. CVC5 demonstrates the highest median standard deviation (approximately 3.2 ms) and the widest overall spread, indicating the most unstable Qed time behavior among the solvers. Overall, the results highlight a clear increase in Qed time dispersion for GPT-5−generated ACSL specifications: while Z3 offers the lowest central variability, Alt-Ergo and CVC4 show moderate instability, and CVC5 exhibits the highest and most dispersed variability profile.

4.6 DeepSeek Generation

Table 6 aggregates 588 WP runs conducted on ACSL specifications generated by DeepSeek, with each solver executed between 145 and 149 times. For all provers, proof success ranges uniformly from 25% to 100%, indicating that while some benchmarks are only partially discharged, complete automation is frequently achieved. In contrast to the previous configurations, dispersion is notably lower, with standard deviations confined to a narrow interval between 13.97 and 14.72, suggesting more stable proof coverage across tasks. Timeout counts, however, remain uneven: Alt-Ergo records
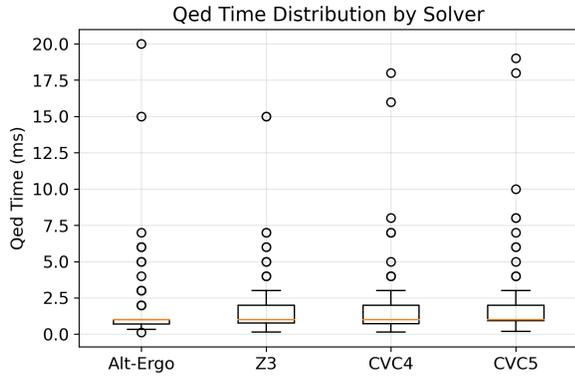
(a) Qed Time Distribution
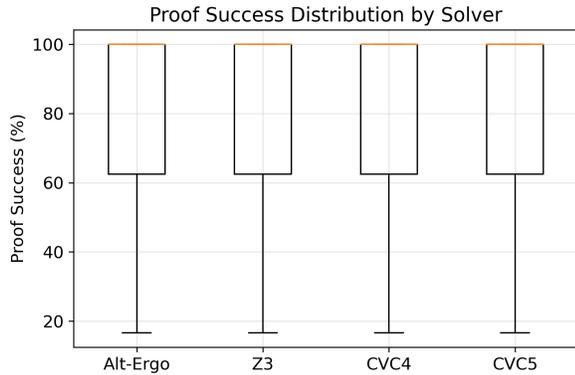
(b) Proof Success Distribution

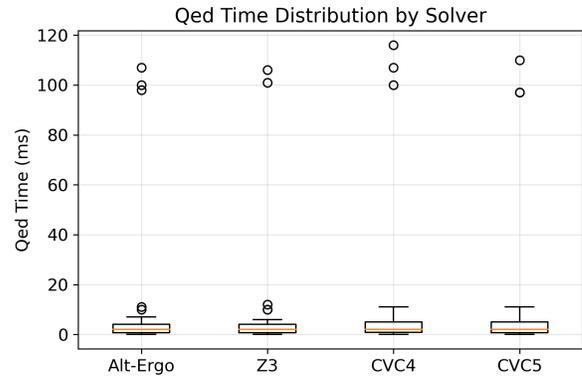**Figure 6.** Solver Performance for GPT-5–Generated ACSL

130 timeouts, substantially exceeding Z3 (22), CVC5 (23), and especially CVC4 (13). Qed averages are considerably higher than in earlier settings, lying between 5.04 and 6.53, and are accompanied by large standard deviations (up to 20.67), reflecting significant variability in simplification impact. Overall, proof success appears more consistent, while timeout frequency and Qed dispersion differentiate solver behaviour.

| Solver	Runs	Min	Max	SD	TO	Qed	SD Qed
Alt-Ergo	149	25.0	100.0	13.97	130	6.17	19.25
CVC4	147	25.0	100.0	14.72	13	6.53	20.67
CVC5	145	25.0	100.0	14.72	23	6.32	19.44
Z3	147	25.0	100.0	14.30	22	5.04	16.27

Table 6. WP verification results for ACSL specifications generated by DeepSeek

Figure 7a presents a comparative analysis of Qed time variability for ACSL specifications generated by the DeepSeek model, where the reported values correspond to the standard deviation of Qed times (in milliseconds) across the four solvers, while Figure 7b presents the proof success distribution. Compared to both RTE-generated and GPT-5–generated ACSL, DeepSeek specifications result in substantially higher Qed time dispersion, indicating significantly less predictable solver behavior. Z3 exhibits the lowest median standard deviation (approximately 16.3 ms), suggesting comparatively more stable proof discharge times, in this more challenging verification condition. Alt-Ergo, CVC4, and CVC5 show markedly higher variability, with median standard deviation values clustered around 19–21 ms and wide interquartile ranges. Among these, CVC4 demonstrates the highest median variability and the broadest spread, indicating the most unstable Qed time behavior, while Alt-Ergo and CVC5 display similarly high dispersion with slightly lower central values. Overall, the results highlight a pronounced increase in Qed time variability for DeepSeek-generated ACSL specifications: Z3 offers the lowest central variability, whereas Alt-Ergo and CVC5 show high but comparable instability, and CVC4 exhibits the highest and most dispersed variability profile among the solvers.

(a) Qed Time Distribution
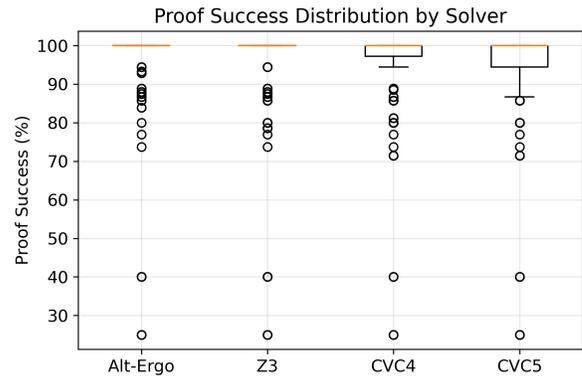
(b) Proof Success Distribution

Figure 7. Solver Performance for DeepSeek–Generated ACSL

4.7 OLMo3 Generation

Table 7 summarises 532 WP runs on ACSL specifications generated by OLMo3, with each solver invoked between 132 and 134 times. All provers share identical proof-success bounds, ranging from 33.33% to 100%, and exhibit exactly the same standard deviation (25.56), indicating an indistinguishable dispersion of results across the benchmark suite. Thus, in terms of coverage and variability, the four solvers behave almost identically in this setting. Differences arise primarily in timeout counts: CVC4 and CVC5 complete all runs without timeouts, whereas Alt-Ergo and Z3 report 92 and 77 timeouts, respectively. Qed averages remain close, between 2.81 and 3.05, with modest standard deviations (1.24–1.56), suggesting limited variation in simplification effects. Overall, while proof success statistics are perfectly aligned across solvers, timeout behaviour constitutes the principal factor separating their performance on these specifications.

Solver	Runs	Min	Max	SD	TO	Qed	SD Qed
Alt-Ergo	134	33.33	100.0	25.56	92	2.92	1.24
CVC4	133	33.33	100.0	25.56	0	2.81	1.25
CVC5	132	33.33	100.0	25.56	0	3.05	1.47
Z3	133	33.33	100.0	25.56	77	2.87	1.56

Table 7. WP verification results for ACSL specifications generated by OLMo3

Figure 8a presents a comparative analysis of Qed time variability for ACSL specifications generated by the OLMo3 model, where the reported values correspond to the standard deviation of Qed times (in milliseconds) across the four solvers, while the Figure 8b presents the proof success distribution. In contrast to DeepSeek- and GPT-5–generated specifications, OLMo3 exhibits markedly lower Qed time dispersion, indicating more stable solver behavior. Alt-Ergo and CVC4 show very similar median standard deviation values (around 1.2–1.3 ms) with tightly clustered distributions, reflecting consistent proof discharge performance across verification runs. CVC5 displays a slightly higher median variability (approximately 1.5 ms) and a moderately wider interquartile range, suggesting limited additional fluctuation. Z3 exhibits the highest median standard deviation among the solvers (around 1.6 ms), accompanied by a broader spread, indicating comparatively less stable behavior, though still within a relatively low absolute range. Overall, the results highlight that ACSL specifications generated by OLMo3 lead to stable and predictable Qed time behavior across solvers: Alt-Ergo and CVC4 provide the most consistent variability profiles, CVC5 shows moderate dispersion, and Z3 exhibits the highest, yet still limited, variability.

Across the five ACSL generation techniques, a clear gradient in verification stability emerges. Rule-based and RTE-generated ACSL consistently produce the lowest Qed time
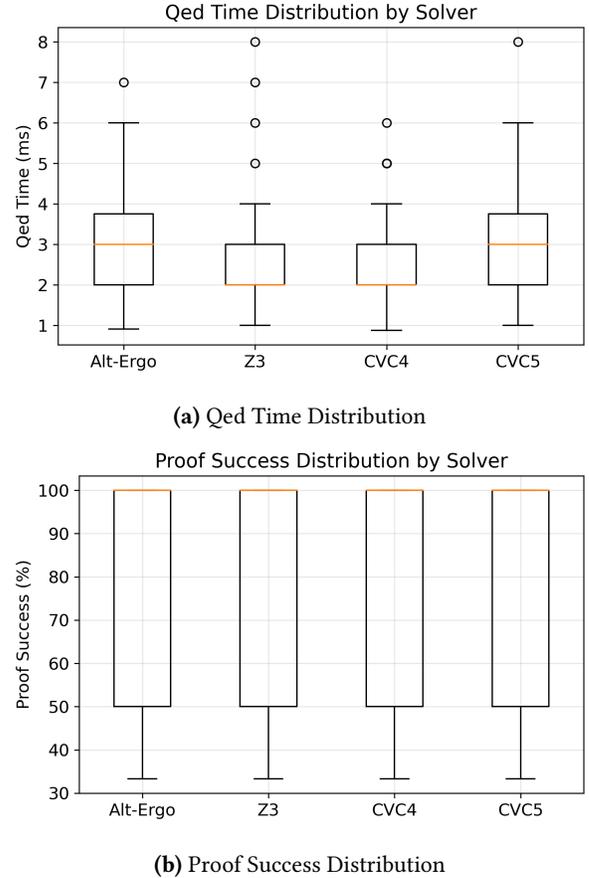

(a) Qed Time Distribution

(b) Proof Success Distribution

Figure 8. Solver Performance for OLMo3–Generated ACSL

variability across all solvers, reflecting the uniform and predictable structure of the generated proof obligations. OLMo3 occupies a middle ground, maintaining low variability while introducing moderately richer specifications that remain solver friendly. In contrast, GPT-5 and DeepSeek generate substantially more heterogeneous ACSL, leading to sharply increased Qed time dispersion and less predictable verification behavior, particularly evident in the wide spreads observed across solvers.

From a solver perspective, Z3 generally maintains consistent runtime performance on complex, LLM-generated ACSL but exhibits broader dispersion, indicating sensitivity to specification irregularities. Alt-Ergo and CVC5 remain stable on structured techniques but degrade under more expressive LLM outputs, while CVC4 shows the strongest consistency on simpler inputs and the most pronounced variability on complex ones. Overall, the comparison highlights a fundamental trade-off: increased expressiveness in ACSL generation correlates with reduced solver stability, whereas disciplined, structure-driven techniques yield more reliable and predictable verification outcomes.

Overall, our results reveal a clear trade-off between expressiveness and verification stability. Tool-generated ACSL (rule-based and RTE) delivers fast, predictable, and highly consistent solver performance, with low mean and median Qed times and minimal dispersion. OLMo3-generated specifications maintain this stability while introducing modest complexity. In contrast, GPT-5 and DeepSeek-generated ACSL produce more heterogeneous proof obligations, leading to higher and more variable Qed times, wider distributions, and increased solver sensitivity. Among solvers, CVC4 and CVC5 provide the most consistent behavior, Z3 achieves lower central variability but exhibits broader fluctuations on complex inputs, and Alt-Ergo experiences the highest timeout rates. These findings emphasize that achieving reliable large-scale verification requires balancing specification richness with controlled variability to ensure predictable solver performance across diverse ACSL generation techniques.

5   Conclusions

This paper presented a systematic empirical study of automated ACSL specification generation and verification for C programs using Frama-C. By analysing a recent November 2025 open-source dataset and evaluating hundreds of repeated verification runs, we established a concrete baseline for tool-driven formalisation using EVA and RTE, as well as for LLM-generated specifications produced by DeepSeek, GPT-5, and OLMo3. The results demonstrate that automated approaches can produce verifiable specifications, but also expose recurring limitations related to parser robustness, annotation completeness, and solver sensitivity. Our comparative evaluation across SMT solvers shows clear differences in verification behavior. RTE-generated ACSL consistently achieves almost 100% proof success, reflecting the reliability of tool-generated runtime contracts. In contrast, LLM-generated annotations exhibit lower and more variable success rates, with performance strongly dependent on both the solver and the model.

Among solvers, CVC4 and CVC5 generally show greater stability, while Alt-Ergo experiences significantly higher timeout rates in several settings. These findings indicate that verification success is shaped not only by annotation quality but also by solver-specific characteristics. Tool-generated specifications provide a dependable foundation, while LLM-generated ACSL shows promise but requires further refinement to improve consistency and solver compatibility.

Further insight is provided by the analysis of Qed times using mean, median, and standard deviation, complemented by boxplot–based variability assessment. Rule-based and RTE-generated ACSL exhibit low mean Qed times, tightly clustered medians, and minimal dispersion, indicating both fast and predictable solver performance. OLMo3 specifications show slightly higher central values but maintain limited spread, suggesting controlled increases in specification complexity.

GPT-5 and DeepSeek generated ACSL present higher mean and median Qed times along with markedly larger standard deviations and wider distributions, revealing substantial variability across verification tasks. These patterns highlight that while LLM generated specifications enhance expressiveness, stable large-scale verification remains reliant on disciplined generation techniques that minimize variability and ensure consistent solver behavior. By grounding our evaluation in a modern dataset and uniform verification pipeline, this study provides practical evidence to guide future research on learning assisted specification synthesis and hybrid formal workflows that balance automation, robustness, and scalability. It also highlights that the existing rule-based generators are starting to be challenged by LLM-based specification generation.

Use of AI-Assisted Tools

We acknowledge the use of free version of GPT-5.2 for refining the textual presentation of the paper. The model was applied to improve clarity and coherence. Followed by this, the text has been thoroughly reviewed and discussed by all authors to ensure accuracy and integrity.

Acknowledgments

References

[1] Arshad Beg, Diarmuid O'Donoghue, and Rosemary Monahan. 2025. Leveraging LLMs for Formal Software Requirements: Challenges and Prospects. In *Proceedings of OVERLAY 2025: Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (CEUR Workshop Proceedings, Vol. 4142).* CEUR-WS.org, 95–105. https://ceur-ws.org/Vol-4142/paper11.pdf

[2] Arshad Beg, Diarmuid O'Donoghue, and Rosemary Monahan. 2025. Traceable and Verifiable Software Requirements: A Synthesis of AI-Enabled Formal Methods. *Journal of Software Testing, Verification and Reliability* (2025). doi:10.5281/zenodo.17772835 Submitted; Version v1 available on Zenodo.

[3] Wen Fan, Marilyn Rego, Xin Hu, Sanya Dod, Zhaorui Ni, Danning Xie, Jenna DiVincenzo, and Lin Tan. 2025. Evaluating the Ability of Large Language Models to Generate Verifiable Specifications in VeriFast. arXiv:2411.02318 [cs.SE] https://arxiv.org/abs/2411.02318

[4] George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2025. Specify What? Enhancing Neural Specification Synthesis by Symbolic Methods. In *Integrated Formal Methods*, Nikolai Kosmatov and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 307–325.

[5] George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2025. Towards Integrating Copiloting and Formal Methods. In *Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Nature Switzerland, Cham, 144–158.

[6] Niclas Hertzberg, Merlijn Sevenhuijsen, Liv Kåreborn, and Anna Lokrantz. 2025. CASP: An Evaluation Dataset for Formal Verification

of C Code. In *Bridging the Gap Between AI and Reality - Third International Conference on Bridging the Gap between AI and Reality, AISoLA 2025, Rhodes, Greece, November 1-5, 2025, Selected Papers (Lecture Notes in Computer Science, Vol. 16220)*, Bernhard Steffen (Ed.). Springer, 63–82. doi:10.1007/978-3-032-07132-3_5

[7] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *CoRR* abs/2311.07948 (2023). arXiv:2311.07948 doi:10.48550/ARXIV.2311.07948

[8] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1509–1520. doi:10.1145/3691620.3695521

[9] Dipayan Saha, Shams Tarek, Hasan Al Shaikh, Khan Thamid Hasan, Pavan Sai Nalluri, Md. Ajoad Hasan, Nashmin Alam, Jingbo Zhou, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. 2025. SV-LLM: An Agentic Approach for SoC Security Verification using Large Language Models. (2025). arXiv:2506.20415 [cs.CR] https://arxiv.org/abs/2506.20415

[10] Merlijn Sevenhuijsen, Khashayar Etemadi, and Mattias Nyberg. 2025. VeCoGen: Automating Generation of Formally Verified C Code With Large Language Models. In *13th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*. IEEE, 101–112. doi:10.1109/FORMALISE66629.2025.00017

[11] Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. 2025. Hilbert: Recursively Building Formal Proofs with Informal Reasoning. *CoRR* abs/2509.22819 (2025).

arXiv:2509.22819 doi:10.48550/ARXIV.2509.22819

[12] Weiqi Wang, Marie Farrell, Lucas C. Cordeiro, and Liping Zhao. 2025. Supporting Software Formal Verification with Large Language Models: An Experimental Study. In *33rd IEEE International Requirements Engineering Conference, RE 2025, Valencia, Spain, September 1-5, 2025*. IEEE, 423–431. doi:10.1109/RE63999.2025.00049

[13] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14682)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[14] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. arXiv:2310.04870 [cs.FL] https://arxiv.org/abs/2310.04870

[15] Yedi Zhang, Sun Yi Emma, Annabelle Lee Jia En, and Jin Song Dong. 2025. RvLLM: LLM Runtime Verification with Domain Knowledge. *CoRR* abs/2505.18585 (2025). arXiv:2505.18585 doi:10.48550/ARXIV.2505.18585

[16] Yihang Zhao. 2025. Leveraging Large Language Models for Ontology Requirements Engineering. In *The Semantic Web: ESWC 2025 Satellite Events - Portoroz, Slovenia, June 1-5, 2025, Proceedings (Lecture Notes in Computer Science, Vol. 15832)*, Edward Curry, Valentina Presutti, John P. McCrae, Mehwish Alam, Pieter Colpaert, Josiane Xavier Parreira, Diego Collarana, Marta Sabou, Andreas Harth, and Pasquale Lisena (Eds.). Springer, 254–264. doi:10.1007/978-3-031-99554-5_40