# Preguss: It Analyzes, It Specifies, It Verifies

Zhongyi Wang*
Zhejiang University
Hangzhou, China
zhongyi.wang@zju.edu.cn

Tengjie Lin*
Zhejiang University
Hangzhou, China
tengjie.lin@zju.edu.cn

Mingshuai Chen†
Zhejiang University
Hangzhou, China
m.chen@zju.edu.cn

Mingqi Yang
Zhejiang University
Hangzhou, China
mingqiyang@zju.edu.cn

Haokun Li
Peking University
Beijing, China
ker@pm.me

Xiao Yi
The Chinese University of Hong Kong
Hong Kong, China
yixiao5428@link.cuhk.edu.hk

Shengchao Qin
Xidian University
Xi'an, China
shengchao.qin@gmail.com

Jianwei Yin
Zhejiang University
Hangzhou, China
zjuyjw@zju.edu.cn

## Abstract

Fully automated verification of large-scale software and hardware systems is arguably the holy grail of formal methods. Large language models (LLMs) have recently demonstrated their potential for enhancing the degree of automation in formal verification by, e.g., generating formal specifications as essential to deductive verification, yet exhibit poor scalability due to context-length limitations and, more importantly, the difficulty of inferring complex, interprocedural specifications. This paper outlines Preguss – a modular, fine-grained framework for automating the generation and refinement of formal specifications. Preguss synergizes between static analysis and deductive verification by orchestrating two components: (i) potential runtime error (RTE)-guided construction and prioritization of verification units, and (ii) LLM-aided synthesis of interprocedural specifications at the unit level. We envisage that Preguss paves a compelling path towards the automated verification of large-scale programs.

***CCS Concepts:*** • **Theory of computation → Program verification**; **Program specifications**; *Program analysis*; • **Software and its engineering → Formal software verification**; *Automated static analysis.*

***Keywords:*** Abstract interpretation, Large language models, Minimal contract, Undefined behaviors

*Both authors contributed equally to this research.
†Corresponding author.



*LMPL '25, Singapore, Singapore*

ACM ISBN 979-8-4007-2148-9/25/10
https://doi.org/10.1145/3759425.3763394

**ACM Reference Format:**
Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Mingqi Yang, Haokun Li, Xiao Yi, Shengchao Qin, and Jianwei Yin. 2025. Preguss: It Analyzes, It Specifies, It Verifies. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759425.3763394

## 1 Introduction

Runtime errors (RTEs), e.g., division by zero, buffer/numeric overflows, and null pointer dereference, are a common cause of undefined behaviors (UBs) exhibited during the execution of C/C++ programs [16, Sect. 3.5.3]. These UBs can trigger catastrophic failures, rendering them critical considerations for safety-critical applications [1, 4, 22, 27]. Consequently, in conventional program verification methodologies, establishing *RTE-freeness* (i.e., conformance to the C standard specification) constitutes a necessary precondition for verifying *functional correctness* (i.e., adherence to intended behaviors) [24]. State-of-the-art *abstract interpretation*-based static analyzers, such as Astrée [18], Frama-C/Eva [8], and Mopsa [17], aim to reliably detect all potential UBs in C programs, thereby formally certifying the absence of RTEs. However, due to the inherent abstraction mechanism that soundly approximates concrete program semantics [10], these tools often emit numerous false positives. Manually identifying such false alarms or tuning analyzer configurations for better accuracy remains notoriously difficult [26].

Program verification tools employing *deductive verification* [2] provide a rigorous methodology for ensuring critical program properties, including both RTE absence and functional correctness. The verification process typically involves two stages: (i) constructing *specifications* to formalize intended program behaviors, and (ii) proving that the program adheres to the specifications. While modern verifiers such as

Frama-C/Wp [7] and Dafny [19] automate the latter stage, the former stage still relies heavily on human expertise.

Recent studies [20, 21, 28–30] have explored large language models (LLMs) for *automated specification synthesis*, demonstrating substantial improvements over conventional techniques. Nevertheless, these methods exhibit significant *scalability limitations* when applied to large-scale programs. Current evaluations [21, 28–30] remain confined to small-scale benchmarks (e.g., SV-COMP test suites [6], Code2Inv benchmark set [23], SyGuS competition [3]), each comprising individual or just a few functions. Although SpecGen [20, Sect. VI-A] explores specification synthesis for substantial real-world Java programs, its focus remains restricted to specification *verifiability* (i.e., syntactic and semantic correctness) rather than *holistic program verification* – emcompassing properties like RTE-freeness and functional correctness. This limitation is primarily due to two constraints: First, the *context-length limitation* of LLMs [13] prohibits them from processing large-scale programs as a whole; Second, verifying complex programs necessitates the synthesis of *a diverse set of interprocedural specifications*, e.g., preconditions, postconditions, invariants, etc. The latter is beyond the capability of existing approaches: (i) some focus exclusively on specific categories (e.g., invariants [21, 29, 30]), and (ii) others generate *function contracts* holistically without modeling intrinsic differences between preconditions and postconditions (as part of the contracts) [20, 28], which are critical for validating potential RTEs (as demonstrated in Section 2).

In response to the aforementioned challenges, we propose Preguss – an LLM-aided framework for synthesizing fine-grained formal specifications. Preguss first employs a static analyzer to emit RTE assertions (signifying all possible RTEs), then constructs a queue of verification units (contexts of program slices and relevant assertions) according to the RTE assertions, and finally prompts an LLM to generate and refine interprocedural specifications for every verification unit. We envisage that Preguss facilitates (i) the *synergy between static analysis and deductive verification*: The RTE assertions reported by static analysis are used to either construct necessary specifications certifying RTE-freeness or locate root causes triggering RTEs; and (ii) *modular synthesis of interprocedural specifications* and thereby a viable approach to the automated verification of large-scale programs.

## 2 Background and Motivation

### 2.1 Potential RTE-Guided Verification

Abstract interpretation-based static analyzers [8, 17, 18] conduct sound value analysis of target programs to warn about potential RTEs. These tools exhibit significant advantages in automatically scaling to large codebases. During the analysis, they embed *RTE (guard) assertions* at program locations susceptible to undefined behaviors – as exemplified by the

```c
#include <limits.h>
int abs(int x) {
  if (x < 0)
    /*@ assert overflow:
    ↪  -2147483647 <= x; */
    return -x;
  else
    return x;
}
void main() {
  int a = abs(42);
  int b = abs(INT_MIN);
}
```

**(a)** analysis result of `abs.c`

```c
#include <limits.h>
/*@ requires INT_MIN < x; */
int abs(int x) {
  if (x < 0)
    return -x;
  else
    return x;
}
void main() {
  int a = abs(42);
  /*@ preconditions of abs */
  int b = abs(INT_MIN);
}
```

**(b)** verification of specified `abs.c`

**Figure 1.** Identifying potential RTEs in a C program via the abstract interpretation-based analyzer Frama-C/Eva [8] and the deductive verifier Frama-C/Wp [7].

assertion `-2147483647 <= x` in Fig. 1a. The analyzers maintain *abstract program states* – e.g., overapproximating possible values $\{-3, 0, 3\}$ via an interval abstraction $[-3, 3]$ – and validate satisfiability of the RTE assertions against these abstract states. Unsatisfied assertions are subsequently reported as RTE alarms. Fig. 1a illustrates the analysis result for the commonly used code snippet `abs.c` generated by Frama-C/Eva [8], which signifies a potential RTE of signed integer overflow expressed in the ANSI/ISO C Specification Language (ACSL) [5].

Tracing the RTE in Fig. 1a is a classic *source-to-sink* problem [25] in static (taint) analysis, namely, to identify that `x = INT_MIN` (source) exclusively triggers the overflow RTE (sink) in the `abs` function. Although taint analysis tools can track data-flow paths from untrusted sources to sinks, validating these paths requires analysts to monitor how the data flows through the program and interacts with different components. This is a prohibitively challenging task in large-scale programs due to intricate call hierarchies and data dependencies. Deductive verification addresses this challenge through an advanced mechanism that propagates guard assertions from sinks upward along caller-callee chains, enabling violation checks at potential source locations. This approach facilitates either precise RTE source tracing or formal establishment of RTE absence [12, 14].

Specifically, by augmenting functions with necessary specifications, e.g., the ACSL precondition `requires INT_MIN < x` in Fig. 1b, verifiers like Frama-C/Wp [7] (based on weakest-precondition reasoning [11]) formally certify RTE-free execution under the specified preconditions. *These preconditions subsequently serve as guard assertions at call sites of caller functions.* Fig. 1b demonstrates this mechanism: The argument `INT_MIN` violates `abs`'s (weakest) precondition, thus leading the verifier to report a *definitive* RTE at call site

```c
int id(int x) {return x;}

void one() {
  int x = id(1);
  /*@ assert division_by_0:
  ↪  x != 0; */
  1/x;
}
void zero() {
  id(0);
}
void main() {
  one(); zero();
}
```

**(a)** analysis result of `id.c`

```c
/*@ requires x != 0; */
/*@ ensures \result == x; */
int id(int x) {return x;}

void one() {
  int x = id(1); 1/x;
}
void zero() {
  /*@ preconditions of id */
  id(0);
}
void main() {
  one(); zero();
}
```

**(b)** verification of specified `id.c`

**Figure 2.** Demonstrating the dual role of interprocedural specifications: postconditions can eliminate false RTEs while over-constrained preconditions can induce false alarms.

`abs(INT_MIN)`; In other words, RTE-freeness can be guaranteed in case no caller of `abs` violates its precondition.

This gives rise to our conceptual idea of *potential RTE-guided verification*, which aims to synergize between static analysis and deductive verification. The core paradigm is to exploit analyzer-reported RTE assertions to either (i) construct necessary specifications certifying RTE-freeness, or (ii) locate root causes triggering RTEs. We will show in Section 3 how this paradigm can be refined to a pipeline that facilitates automated verification of large-scale programs.

### 2.2 Interprocedural Specifications

The example in Fig. 1 illustrates the simple case where an RTE assertion (`-2147483647 <= x`) can be validated through specifications (`requires INT_MIN < x`) *localized to its host function* (`abs`). There are, however, common cases where *interprocedural specifications* are necessary: As demonstrated by Fig. 2, function `one` contains a potential division-by-zero UB flagged by assertion `x != 0` (Fig. 2a), which is in fact a false alarm that can be eliminated solely through the postcondition `ensures \result == x` of function `id` (Fig. 2b). Concretely, the value of `x` is determined by argument `1` and the postcondition of `id` at call site `id(1)`, thereby guaranteeing assertion satisfaction irrespective of the precondition of its host function `one`. Both examples in Figs. 1 and 2 reveal a core forward verification principle: Validating control-flow downstream properties (e.g., RTE assertions in Figs. 1a and 2a) requires correct upstream specifications (e.g., precondition in Fig. 1b or postcondition in Fig. 2b). This observation motivates specialized mechanisms for generating interprocedural specifications in programs with complex call hierarchies.

Nevertheless, existing approaches [20, 28] mostly assume RTE-freeness of a target program for verifying functional correctness, employing simplistic interprocedural-specification synthesis strategies that ask LLMs to generate *holistic* function contracts across the entire program context in a uniform, monolithic manner. This methodology risks inducing false positives during RTE-freeness validation, as illustrated by Fig. 2. An LLM may be misled by the context "`x != 0` and `1/x`" in `one` and thus forge an *over-constrained* precondition `requires x != 0` for `id`, which triggers spurious alarms at call sites like `id(0)` in function `zero`. Although discarding over-constrained preconditions while retaining postconditions may resolve immediate false alarms (as is the case for Fig. 2b), determining whether a precondition is over-constrained is per se a nontrivial task (cf. Fig. 1b vs. Fig. 2b). For Fig. 1b, simply discarding the precondition compromises soundness (as true RTEs are missed) and thus undermines verification integrity. Therefore, interprocedural-specification synthesis mechanisms should make efforts to prevent the generation of over-constrained preconditions.

## 3 Methodology

This section presents the design principles behind PREGUSS – our framework for Potential Runtime Error-GUided Specification Synthesis. As depicted in Fig. 3, PREGUSS is comprised of two synergistic components: (i) *Potential RTE-guided construction and prioritization of verification units*: a divide-and-conquer strategy [31] for decomposing the monolithic RTE-freeness verification into prioritized units; (ii) *Fine-grained interprocedural specification synthesis*: a tactic for inferring necessary specifications via LLMs along caller-callee chains to validate target assertions per verification unit. Below, we show how these two components cooperate to facilitate PREGUSS's scalability to large-scale programs.

### 3.1 Decomposing the Monolithic Verification

To address the context-length limitation of LLMs [13], we decompose the monolithic RTE-freeness verification problem into *a sequence of subproblems for validating individual potential RTEs associated with necessary program contexts*. As shown in Fig. 3, we first employ static analysis to generate RTE assertions for all possible UBs in the target program (❶). We then construct the program's call graph while recording all the call sites (❷), yielding a complete set of RTE guard assertions. These assertions are classified into two distinct categories: (i) UB assertions generated during the initial analysis, and (ii) call-site preconditions (initially defaulting to tautological `true`) to be synthesized/updated by LLMs per Section 3.2. For each assertion, we construct a *verification unit* (V-Unit) containing both the assertion and its necessary contextual program slices. These V-Units are prioritized in a queue (❸) exactly matching the sequence of their corresponding RTE assertions produced by a post-order traversal of the call graph. This deliberate ordering implements the bottom-up verification mechanism by progressing systematically from leaf functions toward the root function [28].

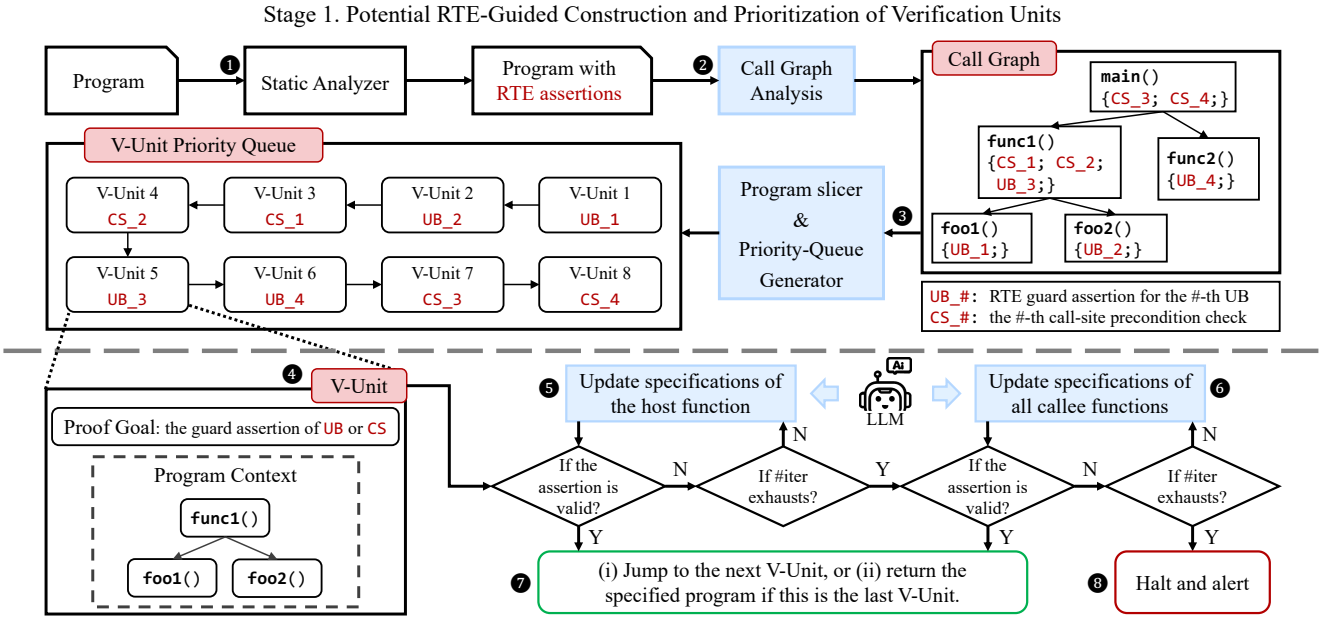


**Figure 3.** Architecture of the PREGUSS framework.

PREGUSS leverages the *modularization principle of deductive verification* [2, Chap. 9], wherein functions are verified in isolation using their contracts. Rather than feeding LLMs with the entire program in an end-to-end manner, validating individual RTEs requires a constrained context comprising two components: the guard assertion and a two-layer program slice (encompassing both the assertion's host function and its callees), collectively encapsulated within a V-Unit (e.g., ❹). *This design ensures that the context does not expand significantly during the bottom-up verification progression, thereby enabling scalability to large-scale programs.*

### 3.2 Generating Interprocedural Specifications

As motivated in Section 2, PREGUSS implements a fine-grained strategy for synthesizing interprocedural specifications. This approach fundamentally differs from existing methods by avoiding the holistic function-contract generation from uniform contexts. PREGUSS processes each V-Unit through two distinct phases: specification generation for the assertion's host function (❺), followed by specification generation for callee functions (❻). Both phases employ an iterative refinement strategy where (i) an LLM generates candidate specifications, (ii) a verifier validates the target assertion using these specifications, and (iii) an LLM refines the (possibly) failed candidates based on the verifier's feedback. This procedure terminates until either the validation succeeds or the iteration limit (#iter) is exhausted. In the initial phase (❺), the LLM is prompted to infer preconditions and auxiliary specifications (e.g., loop contracts) through backward reasoning from the guard assertion – as informally analogous to the process of weakest-precondition reasoning [11]. When the assertion depends on callees' return values (recall Fig. 2a), the verification requires callees' postconditions, thus triggering the transition to the subsequent phase (❻). Crucially, to prevent over-constrained preconditions (as exemplified in Fig. 2b), the callees' precondition synthesis is expressly prohibited during this stage. Upon successful verification of the assertion alongside all generated specifications (excluding host preconditions), PREGUSS either proceeds to the next V-Unit or returns the specified program if the priority queue becomes empty (❼). Unverifiable assertions trigger immediate termination with high-risk RTE alerts (❽).

PREGUSS directly operationalizes our insights presented in Section 2.2: By leveraging the V-Unit's constrained context and implementing a two-stage interprocedural-specification synthesis strategy (thereby mirroring the forward verification principle), PREGUSS systematically avoids generating over-constrained preconditions while enabling automated, scalable RTE-freeness verification for large-scale programs.

## 4 Conclusion and Future Directions

We have conceived PREGUSS – a modular, fine-grained framework for inferring formal specifications that synergize between static analysis and deductive verification. We envisage that PREGUSS offers a promising paradigm towards the automated verification of large-scale programs (subject to an extensive experimental evaluation in future work).

While initially devised to certify RTE-freeness and identify genuine RTEs, the framework can be extended to cater for the verification of other vulnerability classes and functional

correctness. The key is to substitute RTE-assertion generation (via static analysis) with tailored formal annotations for the target properties, including ones with complex data structures. The subsequent stages, commencing from ❷ in Fig. 3, remain fully generic to handle these annotations.

Although Preguss presents a promising paradigm for automated large-scale program verification, establishing a formal proof of its *soundness* remains non-trivial – It requires a rigorous justification that verified properties hold under all execution paths. This necessitates foundational work in two directions: (i) precise formalization of target verification properties, and (ii) rigorous proof establishing semantic equivalence between synthesized specifications and program behaviors. This foundational work must bridge the gap between practical LLM-based synthesis and formal method guarantees. Additionally, language-specific complexities pose significant soundness threats. The pervasive use of function pointers in C programs may induce incomplete call graphs during static analysis, potentially causing undetected function dependencies, verification context omissions, and unsound RTE guard propagation. Ensuring call graph reliability – through advanced pointer analysis techniques like flow-sensitive analysis – becomes imperative for any robust implementation.

Conceived for scalability in programs with lengthy chains of function calls, Preguss exhibits limitations when applied to: (i) Mutually recursive functions, which induce loopy structures in the call graph disrupting the bottom-up verification mechanism, causing potential failures of the interprocedural specification synthesis; (ii) Star-structured call graphs (characterized by few callers invoking numerous callees), where V-Units containing an inflated context of such functions may exceed the LLM's context limit. For the latter, a potential solution is to employ program slicing to extract the minimal subset of statements exhibiting dependencies with the target assertion, thus omitting nonessential callees.

Given LLMs' susceptibility to *hallucination* [15], they frequently generate specifications exhibiting critical flaws – ranging from *syntactically illegal* constructs that violate specification language grammars to *semantically unsatisfiable* predicates that contradict actual program behaviors. While verifiers provide rich feedback (e.g., proof obligations and diagnostic logs) on such errors, translating these formal outputs into effective LLM refinement prompts presents a significant research challenge due to the formal-to-informal semantic gap (cf. [9]). Future work should therefore focus on developing principled mechanisms to bridge this gap.

## Acknowledgments

This work was funded by the Fundamental Research Funds for the Central Universities of China (No. 226-2024-00140), by the ZJNSF Major Program (No. LD24F020013), by the CCF-Huawei Populus Grove Fund (No. CCF-HuaweiSY202503), by the Open Fund of the High-Reliability Embedded Software Engineering Technology Laboratory (No. LHCESET202502), and by the Huawei Technical Collaboration Project (No. TC20250422031).

## References

[1] 2000. *The Explosion of the Ariane 5.* https://www-users.cse.umn.edu/~arnold/disasters/ariane.html

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification – The KeY Book – From Theory to Practice.* Lecture Notes in Computer Science, Vol. 10001. Springer. doi:10.1007/978-3-319-49812-6

[3] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. *CoRR* abs/1904.07146 (2019).

[4] America's Cyber Defense Agency. 2025. *Secure by Design Alert: Eliminating Buffer Overflow Vulnerabilities.* https://www.cisa.gov/resources-tools/resources/secure-design-alert-eliminating-buffer-overflow-vulnerabilities

[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2025. *ANSI/ISO C Specification LanguageVersion 1.22.* https://www.frama-c.com/download/frama-c-acsl-implementation.pdf

[6] Dirk Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *TACAS (3) (Lecture Notes in Computer Science, Vol. 14572).* Springer, 299–329. doi:10.1007/978-3-031-57256-2_15

[7] Allan Blanchard. 2020. *Introduction to C program proof with Frama-C and its WP plugin.* https://allan-blanchard.fr/frama-c-wp-tutorial.html

[8] David Bühler, Pascal Cuoq, and Boris Yakobowski. 2025. *The Eva plug-in.* https://www.frama-c.com/download/frama-c-eva-manual.pdf

[9] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, Shing-Chi Cheung, and Cong Tian. 2025. From Informal to Formal - Incorporating and Evaluating LLMs on Natural Language Requirements to Verifiable Formal Proofs. In *ACL (1).* Association for Computational Linguistics, 26984–27003. doi:10.18653/v1/2025.acl-long.1310

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL.* ACM, 238–252. doi:10.1145/512950.512973

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. doi:10.1145/360933.360975

[12] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a RTE-Free X.509 Parser. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2019).*

[13] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In *USENIX Security Symposium.* USENIX Association. doi:10.5555/3698900.3698947

[14] Jens Gerlach. 2019. *Minimal contract Hoare-style verification versus abstract interpretation.* Technical Report.

[15] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2 (2025), 42:1–42:55. doi:10.1145/3703155

[16] ISO/IEC JTC 1/SC 22. 2024. *ISO/IEC 9899:2024 Information technology — Programming languages — C.* ISO. https://www.iso.org/standard/

82075.html

[17] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *VSTTE (Lecture Notes in Computer Science, Vol. 12031)*. Springer, 1–18. doi:10.1007/978-3-030-41600-3_1

[18] Daniel Kästner, Reinhard Wilhelm, and Christian Ferdinand. 2023. Abstract Interpretation in Industry - Experience and Lessons Learned. In *SAS (Lecture Notes in Computer Science, Vol. 14284)*. Springer, 10–27. doi:10.1007/978-3-031-44245-2_2

[19] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (Dakar) (Lecture Notes in Computer Science, Vol. 6355)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[20] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. (2025), 16–28. doi:10.1109/ICSE55347.2025.00129

[21] Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C. Cordeiro. 2024. LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling. In *ASE*. ACM, 1395–1407. doi:10.1145/3691620.3695512

[22] Narges Shadab, Pritam M. Gharat, Shrey Tiwari, Michael D. Ernst, Martin Kellogg, Shuvendu K. Lahiri, Akash Lal, and Manu Sridharan. 2025. Lightweight and modular resource leak checking (extended version). *Int. J. Softw. Tools Technol. Transf.* 27, 2 (2025), 267–288. doi:10.1007/s10009-025-00804-2

[23] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *CAV (2) (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 151–164. doi:10.1007/978-3-030-53291-8_9

[24] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. 2009. Formal Verification of Avionics Software Products. In *FM (Lecture Notes in Computer Science, Vol. 5850)*. Springer, 532–546. doi:10.1007/978-3-642-05089-3_34

[25] Chengpeng Wang. 2025. *Advances in AI-Powered Code Security: Next-Level Bug Detection.* Technical Report.

[26] Zhongyi Wang, Linyu Yang, Mingshuai Chen, Yixuan Bu, Zhiyang Li, Qiuye Wang, Shengchao Qin, Xiao Yi, and Jianwei Yin. 2024. Parf: Adaptive Parameter Refining for Abstract Interpretation. In *ASE*. ACM, 1082–1093. doi:3691620.3695487

[27] Westley Weimer and George C. Necula. 2004. Finding and preventing run-time error handling mistakes. In *OOPSLA*. ACM, 419–431. doi:10.1145/1028976.1029011

[28] 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 *CAV (2) (Lecture Notes in Computer Science, Vol. 14682)*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[29] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-Symbolic Loop Invariant Inference. In *ASE*. ACM, 406–417. doi:10.1145/3691620.3695014

[30] Haoze Wu, Clark W. Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. In *ICLR*. OpenReview.net.

[31] Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. 2025. Validating Network Protocol Parsers with Traceable RFC Document Interpretation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA078 (June 2025), 23 pages. doi:10.1145/3728955

Received 2025-07-06; accepted 2025-08-08