# A Tale of 1001 LoC: Potential Runtime Error-Guided Specification Synthesis for Verifying Large-Scale Programs

ZHONGYI WANG*, Zhejiang University, China
TENGJIE LIN*, Zhejiang University, China
MINGSHUAI CHEN†, Zhejiang University, China
HAOKUN LI, Peking University, China
MINGQI YANG, Zhejiang University, China
XIAO YI, The Chinese University of Hong Kong, China
SHENGCHAO QIN, Xidian University, China
YIXING LUO, Beijing Institute of Control Engineering, China
XIAOFENG LI, Beijing Institute of Control Engineering, China
BIN GU, Beijing Institute of Control Engineering, China
LIQIANG LU, Zhejiang University, China
JIANWEI YIN, Zhejiang University, China

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 long-context reasoning limitations and, more importantly, the difficulty of inferring complex, interprocedural specifications. This paper presents Preguss – a modular, fine-grained framework for automating the generation and refinement of formal specifications. Preguss synergizes between static analysis and deductive verification by steering two components in a divide-and-conquer fashion: (i) potential runtime error-guided construction and prioritization of verification units, and (ii) LLM-aided synthesis of interprocedural specifications at the unit level. We show that Preguss substantially outperforms state-of-the-art LLM-based approaches and, in particular, it enables highly automated RTE-freeness verification for real-world programs with over a thousand LoC, with a reduction of 80.6%∼88.9% human verification effort.

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

Additional Key Words and Phrases: Abstract interpretation, Deductive verification, Large language models

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

Authors' Contact Information: 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; Haokun Li, Peking University, Beijing, China, ker@pm.me; Mingqi Yang, Zhejiang University, Hangzhou, China, mingqiyang@zju.edu.cn; 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; Yixing Luo, Beijing Institute of Control Engineering, Beijing, China, luoyi_xing@126.com; Xiaofeng Li, Beijing Institute of Control Engineering, Beijing, China, li_x_feng@126.com; Bin Gu, Beijing Institute of Control Engineering, Beijing, China, gubin@ios.ac.cn; Liqiang Lu, Zhejiang University, Hangzhou, China, liqianglu@zju.edu.cn; Jianwei Yin, Zhejiang University, Hangzhou, China, zjuyjw@zju.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART160
https://doi.org/10.1145/3798268

**ACM Reference Format:**
Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Haokun Li, Mingqi Yang, Xiao Yi, Shengchao Qin, Yixing Luo, Xiaofeng Li, Bin Gu, Liqiang Lu, and Jianwei Yin. 2026. A Tale of 1001 LoC: Potential Runtime Error-Guided Specification Synthesis for Verifying Large-Scale Programs. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 160 (April 2026), 29 pages. https://doi.org/10.1145/3798268

## 1 Introduction

Runtime errors (RTEs), such as 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 [ISO/IEC JTC 1/SC 22 2024, Sect. 3.5.3]. These UBs can trigger catastrophic failures, rendering them critical considerations for safety-critical applications [ari 2000; America's Cyber Defense Agency 2025; Shadab et al. 2025]. 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) [Souyris et al. 2009]. State-of-the-art static analyzers based on *abstract interpretation* [Cousot and Cousot 1977], such as Astrée [Kästner et al. 2023] and Frama-C/Eva [Bühler et al. 2025], aim to reliably detect all potential UBs in large-scale C programs, thereby formally certifying the absence of RTEs. However, due to the inherent abstraction mechanism that soundly approximates concrete program semantics [Cousot and Cousot 1977], these tools often emit numerous false positives. Manually identifying such false alarms or tuning analyzer configurations for better accuracy remains notoriously difficult [Ourghanlian 2015; Wang et al. 2024].

Program verification tools, such as *(automated) deductive verifiers* [Ahrendt et al. 2016], provide a rigorous means for ensuring critical properties, including RTE-freeness 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 [Blanchard 2020] and Dafny [Leino 2010] automate the latter stage, the former – known as *specification synthesis* [Ammons et al. 2002] – still relies heavily on human expertise, especially for large-scale programs. [Djoudi et al. 2021; Ebalard et al. 2019].

Recent studies [Ma et al. 2025; Pirzada et al. 2024; Wen et al. 2024; Wu et al. 2024b,a] 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 real-world programs with, e.g., thousands of lines of code (LoC). The reasons are primarily twofold. First, the *long-context reasoning limitation* of LLMs prohibits them from processing large-scale programs as a whole: (i) the entire program may exceed the maximum context window length [Fang et al. 2024]; and (ii) even when the window is sufficiently long, LLMs often struggle to effectively locate and utilize the precise information needed to solve the task in one shot [Lekssays et al. 2025; Zhang et al. 2024]. Second, verifying large-scale programs – which typically feature complex call hierarchies – necessitates the synthesis of *a diverse set of interprocedural specifications*, e.g., *contracts* (pre- and postconditions), loop invariants, etc. The latter is beyond the capability of existing approaches: (i) some focus exclusively on intraprocedural specifications of specific categories, e.g., invariants [Pirzada et al. 2024; Wu et al. 2024b,a] and assertions [Mugnier et al. 2025], and (ii) others generate interprocedural specifications holistically without modeling intrinsic differences between preconditions and postconditions [Ma et al. 2025; Wen et al. 2024], which are critical for identifying or validating the absence of RTEs (as demonstrated in Section 3). A more detailed review of related work is provided in Section 8.

In response to the aforementioned challenges, we present Preguss – *an LLM-assisted framework for synthesizing fine-grained formal specifications that enable the effective verification of large-scale programs.* Preguss adopts a *divide-and-conquer* strategy orchestrating two synergistic phases: Phase

1 (*Divide*) decomposes the monolithic task of RTE-freeness verification into manageable, prioritized units, guided by RTE assertions emitted by a static analyzer; Phase 2 (*Conquer*) employs an LLM to infer interprocedural specifications with fine-grained strategies tailored to different program contexts pertaining to the units. We show that this divide-and-conquer methodology 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 help locate root causes triggering RTEs (à la the *minimal contract* paradigm [Gerlach 2019]); and (ii) *modular synthesis of interprocedural specifications* (at the granularity of verification units) and thereby a viable approach to the automated verification of large-scale programs.

We have implemented Preguss on top of the deductive verifier Frama-C/Wp and abstract interpretation-based analyzers Frama-C/Eva and Frama-C/Rte [Herrmann and Signoles 2025]. Experimental results show that Preguss substantially outperforms state-of-the-art LLM-based approaches in terms of the rate of successfully verified benchmarks. In particular, Preguss *achieves highly automated RTE-freeness verification for real-world programs with over a thousand LoC, with a reduction of 80.6%~88.9% human verification effort* (measured by the number of human-intervened specifications). Moreover, Preguss facilitates the identification of 6 confirmed RTEs in a practical spacecraft control system (1,280 LoC, 48 functions). These results demonstrate Preguss's effectiveness in synthesizing high-quality specifications and scalability to large projects.

**Contributions.** The main contributions of this paper are summarized as follows.

- We propose and formalize the concept of *potential RTE-guided specification synthesis* as an effective means for LLM-aided specification synthesis towards scalable program verification.
- We present Preguss – *a modular, fine-grained framework for inferring formal specifications* that integrates static analysis and deductive verification in a sound and terminating manner. Preguss is, to the best of our knowledge, the first automated method capable of proving RTE-freeness of real-world programs with over 1, 000 LoC (with marginal human effort).
- We construct an open-source dataset consisting of real-world C programs annotated with specifications generated by Preguss and crafted by experts for RTE-freeness verification.
- We implement Preguss and demonstrate its effectiveness and scalability on the benchmark dataset. We show that Preguss outperforms state-of-the-art LLM-based approaches, and achieves highly automated RTE-freeness verification of large-scale, real-world programs.

*Paper Structure.* Section 2 recaps preliminaries on program verification. Section 3 motivates our conceptual idea of potential RTE-guided verification. Section 4 formalizes the problem of potential RTE-guided specification synthesis. Section 5 presents the technical underpinnings of our Preguss framework. Section 6 reports the evaluation results. Section 7 addresses several limitations and future directions. A review of related work is given in Section 8. The paper is concluded in Section 9.

## 2 Preliminaries

Program verification seeks to determine whether a program conforms to a given formal specification. This section recaps several basic concepts in program verification. The discussion is not confined to specific programming or specification languages; rather, we focus on general imperative programs (with instruction-based execution) assessed against predicate-based specifications.

*Program States and Configurations.* A *program state* $\sigma$ maps every program variable in $\texttt{vars}$ to its value in $\texttt{vals}$. We denote the set of program states by $\Sigma \triangleq \{\sigma \mid \sigma\colon \texttt{vars} \to \texttt{vals}\}$. A *program configuration* $c \triangleq \langle \sigma, \iota \rangle$ pairs a state $\sigma \in \Sigma$ with the next instruction $\iota$ to be executed, representing the program's instantaneous state immediately before executing $\iota$.

**Example 1.** Consider a program snippet in C: "`int x = 1; x++;`". The configuration $\langle \{\texttt{x} \mapsto 1\}, \texttt{x++} \rangle$ captures the program state where variable `x` has value 1 immediately before executing `x++`. ◁

*Properties.* A *property* is a tuple $p \triangleq \langle \phi, \iota \rangle$, where $\phi$ is a first-order logic *predicate* (possibly with quantifiers) representing a subset of program states in $\Sigma$ immediately before executing instruction $\iota$. We say that state $\sigma$ satisfies predicate $\phi$, written as $\sigma \models \phi$, iff $\sigma$ is in the set specified by $\phi$, i.e., $\phi(\sigma) = \texttt{true}$; and $\sigma \not\models \phi$ otherwise. Moreover, a configuration $c = \langle \sigma, \iota' \rangle$ satisfies property $p = \langle \phi, \iota \rangle$, denoted by $c \models p$, iff $\sigma \models \phi$ and $\iota' = \iota$; otherwise, it is denoted by $c \not\models p$.

In this paper, the terms *specifications*, *hypotheses*, and *assertions* all refer to instances of properties.

**Example 2.** Consider the program `abs.c` given in Fig. 1 (a), which defines a function `abs` for computing the absolute value of an integer input `x`. The `main` function calls `abs` twice, respectively with argument `-42` and `INT_MIN`. A potential integer-overflow UB may occur when `x` evaluates to `INT_MIN` immediately before executing instruction $\iota_1$ (the return statement). As depicted in Fig. 1 (b), an abstract interpretation-based static analyzer (e.g., Frama-C/Eva) detects this risk and inserts an assertion $p_1 = \langle \phi_1, \iota_1 \rangle$ at the potential UB point, using the ANSI/ISO C Specification Language (ACSL) [Baudin et al. 2025]. The predicate $\phi_1 = \texttt{INT_MIN < x}$ encodes the necessary constraint to prevent integer overflow during the execution of $\iota_1$, thereby serving as an RTE guard. ◁

**Definition 3 (Trace and Subtrace).** *A* trace $\tau$ *is a finite sequence of program configurations, i.e.,*

$$\tau \;\triangleq\; (c_0, \ldots, c_t) \;=\; (\langle \sigma_0, \iota_0 \rangle, \ldots, \langle \sigma_t, \iota_t \rangle)\ . \tag{1}$$

*Here, $\sigma_0$ is the initial state; and the state transition at each step $i \geq 0$ is governed by $\sigma_{i+1} = [\![ \iota_i ]\!](\sigma_i)$, where $[\![ \iota_i ]\!]$ denotes the language-specific, single-step operational semantics of instruction $\iota_i$* [Floyd 1993; McCarthy 1961]. *The* length *of trace $\tau$ in* (1) *is $t+1$. A trace $\tau' = (c'_0, \ldots, c'_s)$ is termed a* subtrace *of $\tau$, denoted by $\tau' \prec \tau$, iff it forms a proper prefix of $\tau$, i.e., $s < t$ and $c'_i = c_i$ for any $0 \leq i \leq s$.*

**Example 4.** The program fragment in Example 1 admits, in principle, infinitely many traces since its initial state is unconstrained. One such trace is the sequence $\tau_1 = (\langle \texttt{x} \mapsto 0, \texttt{int x = 1} \rangle, \langle \{\texttt{x} \mapsto 1\}, \texttt{x++} \rangle, \langle \{\texttt{x} \mapsto 2\}, \iota_{\text{null}} \rangle)$, where $\iota_{\text{null}}$ denotes a null instruction signifying termination. ◁

Next, we relate traces and properties via the notions of reachability and satisfaction:

**Definition 5 (Trace Reachability and Satisfaction).** *Given a property $p = \langle \phi, \iota \rangle$ and a trace $\tau = (c_0, \ldots, c_t) = (\langle \sigma_0, \iota_0 \rangle, \ldots, \langle \sigma_t, \iota_t \rangle)$, we say that $\tau$* reaches *$p$, denoted by $\tau \rightsquigarrow p$, iff $\iota_t = \iota$, i.e., the last instruction of $\tau$ coincides with $\iota$. Furthermore, we extend the satisfaction relation $\models$ to traces and properties: $\tau$* satisfies *$p$, written as $\tau \models p$, if and only if $\tau \rightsquigarrow p$ and $c_t \models p$.*

**Example 6.** Recall Fig. 1 (b) as discussed in Example 2. Consider two traces $\tau_2$ and $\tau_3$, which originate from instructions $\iota_2$ and $\iota_3$, respectively, and both terminate at $\iota_1$ – the instruction associated with assertion $p_1$ that signifies a potential integer-overflow UB. We have $\tau_2 \rightsquigarrow p_1$, $\tau_3 \rightsquigarrow p_1$, $\tau_2 \models p_1$, yet $\tau_3 \not\models p_1$. The latter is due to the argument `INT_MIN` of $\iota_3$ which triggers a genuine RTE at $\iota_1$. ◁

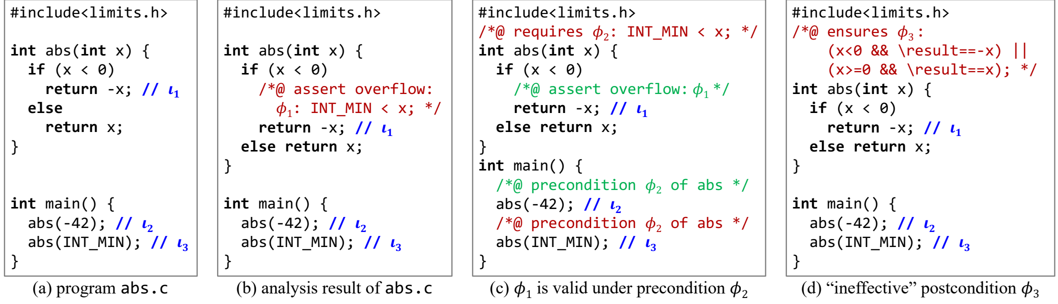
The following definition captures interdependencies between specifications:

**Definition 7 (Property Validity under Hypotheses).** *Given two properties $p$ and $q$, we abuse the notation and write $p \models q$ to denote that $q$ is* valid under hypothesis *$p$, if and only if for every trace $\tau$ that reaches $q$, i.e., $\tau \rightsquigarrow q$, the following condition holds:*

$$\underbrace{\forall \tau' \colon\; (\tau' \prec \tau \wedge \tau' \rightsquigarrow p \implies \tau' \models p)}_{\text{every subtrace of } \tau \text{ that reaches } p \text{ also satisfies } p} \quad \text{implies} \quad \underbrace{\tau \models q}_{\tau \text{ satisfies } q}\ . \tag{2}$$

*More generally, $q$ is* valid under a set of hypotheses *$\mathcal{H}$, written as $\mathcal{H} \models q$, iff for any $\tau \rightsquigarrow q$,*

$$\forall h \in \mathcal{H}.\, \forall \tau' \colon\; (\tau' \prec \tau \wedge \tau' \rightsquigarrow h \implies \tau' \models h) \quad \text{implies} \quad \tau \models q\ . \tag{3}$$




(a) program `abs.c` (b) analysis result of `abs.c` (c) $\phi_1$ is valid under precondition $\phi_2$ (d) "ineffective" postcondition $\phi_3$

Fig. 1. Potential RTE-guided verification process (a)(b)(c) v.s. unguided verification (a)(d).

**Example 8.** Fig. 1 (c) demonstrates a well-designed precondition $p_2 = \langle \phi_2, \iota^{\mathsf{abs}}_{\mathrm{pre}} \rangle$ above function `abs`, where predicate $\phi_2$ is identical to $\phi_1$ and $\iota^{\mathsf{abs}}_{\mathrm{pre}}$ denotes the first instruction following the entry point of function `abs`.[1] This precondition ensures the validity of assertion $p_1$, namely $p_2 \models p_1$, and thus function `abs` is certified RTE-free under hypothesis $p_2$. An example of validity under *multiple* hypotheses can be found in Section 3.3. ◁

Now, we have all the ingredients to formally describe a sound program verifier:

**Definition 9 (Sound Verifier).** *Given a program prog, a hypothesis set* $\mathcal{H}$*, and a target property q, a (deductive)* verifier *is a function mapping these inputs to a certain verification result:*

$$\mathit{Verify}\colon\ (\mathit{prog}, \mathcal{H}, q) \ \mapsto\ \mathit{status}\,, \tag{4}$$

*where the output status is either (i) true signifying that q is valid under* $\mathcal{H}$*; or (ii) unknown indicating invalidity or verification failure due to resource exhaustion and/or solver limitations (e.g., employing over-approximations of program semantics rather than exact representations)* [Kroening and Strichman 2016]*. In particular, the verifier in* (4) *is* sound *if it satisfies the following soundness condition:*

$$\mathit{Verify}\,(\mathit{prog}, \mathcal{H}, q) \ = \ \mathit{true} \quad \text{implies} \quad \mathcal{H} \ \models\ q\,. \tag{5}$$

*This condition ensures that a true verdict guarantees the validity of q under* $\mathcal{H}$*, while* $\mathcal{H} \not\models q$ *necessarily results in unknown. It then follows from the consequence rule of Floyd-Hoare Logic* [Hoare 1969] *that*

$$\mathit{Verify}\,(\mathit{prog}, \mathcal{H}, q) \ = \ \mathit{true} \ \text{and} \ \mathcal{H} \subseteq \mathcal{H}' \quad \text{implies} \quad \mathit{Verify}\,(\mathit{prog}, \mathcal{H}', q) \ = \ \mathit{true}\,, \tag{6}$$

*i.e., a property valid under hypotheses* $\mathcal{H}$ *must also be valid under logically stronger hypotheses* $\mathcal{H}'$*.*

The verifier-reported statuses (*true* or *unknown*) for all properties in Figs. 1 to 3 are visually indicated using their corresponding colors. Moreover, in practice, a sound verifier can take an extended form

$$\mathit{Verify}\colon\ (\mathit{prog}, \mathcal{H}, q) \ \mapsto\ \langle \mathit{status}, \mathit{feedback} \rangle\ , \tag{7}$$

which augments (4) with auxiliary *feedback* information. Such feedback typically includes *proof obligations* (aka, *verification conditions*) – logical formulas that, when being valid, suffice to guarantee program adherence to specifications. A status of *true* indicates that all proof obligations are valid, whereas a status of *unknown* signifies that the verifier could neither prove nor refute the obligations. We show in Section 3.3 that such feedback can be leveraged to refine specifications.

## 3 Motivation

### 3.1 Potential RTE-Guided Verification

We use the simple example in Fig. 1 to demonstrate our idea of *potential RTE-guided verification* [Wang et al. 2025b], which aims to synergize between static analysis and deductive verification (in line with the *minimal contract* paradigm [Gerlach 2019]): We first apply a static analyzer to the source program in Fig. 1 (a); if any RTE alarm is reported in the form of an assertion, e.g., $p_1 = \langle \phi_1, \iota_1 \rangle$ in Fig. 1 (b) (as explained in Example 2), we construct necessary specifications, e.g., the precondition $p_2 = \langle \phi_2, \iota_{\text{pre}}^{\text{abs}} \rangle$ in Fig. 1 (c) (as elaborated in Example 8), which assist a verifier in certifying the assertion. Both the assertion and the precondition then serve as *guard assertions* to ensure RTE-freeness of the source program. Fig. 1 (c) demonstrates this mechanism: By verifying whether the (weakest) precondition $p_2$ is violated at call sites of function `abs` (i.e., $\iota_2$ and $\iota_3$), we can distinguish safe traces (e.g., $\tau_2$) from unsafe ones (e.g., $\tau_3$) that induce a *definite* RTE.

*Can LLMs generate such specifications?* Taking `abs.c` as an example, most advanced LLMs – when fed with the source program context *and* the RTE assertion $p_1$ – exhibit strong code comprehension capabilities and can readily generate weakest preconditions semantically equivalent to $p_2$. However, state-of-the-art approaches to LLM-based specification synthesis [Ma et al. 2025; Pirzada et al. 2024; Wen et al. 2024; Wu et al. 2024b,a] do not exploit RTE assertions emitted by established static analysis techniques (most pertinently, abstract interpretation [Cousot and Cousot 1977]), thus significantly impairing their performance in generating necessary specifications for RTE-freeness verification. For instance, it is common for LLMs to produce specifications such as the postcondition $p_3 = \langle \phi_3, \iota_{\text{post}}^{\text{abs}} \rangle$ depicted in Fig. 1 (d), given *only* the source program `abs.c` in Fig. 1 (a). The predicate $\phi_3$ = `(x<0 && \result ==-x) || (x>=0 && \result ==x)` appears to correctly describe the behavior of `abs`, except for the case `x = INT_MIN`, leading verifiers to mark it as *unknown*. Moreover, postcondition $p_3$ is "ineffective" for detecting the actual RTE source at $\iota_3$. Hence, we have

**Insight 1.** One can leverage RTE assertions from abstract interpretation-based analyzers – as targets for RTE-freeness verification – to guide LLMs to synthesize specifications effectively.

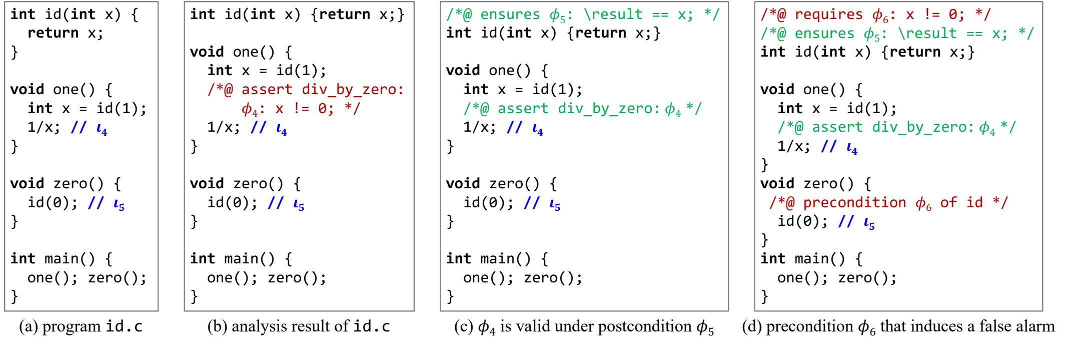
Then, the following question naturally arises: *Does it suffice to extend existing LLM-based synthesis approaches by simply feeding the LLM therein with RTE assertions?* The answer is unfortunately negative: We discuss the corresponding technical challenges throughout the rest of this section and report the practical insufficiency of such a naïve extension in Section 6 (RQ1).

### 3.2 Interprocedural Specifications

The example in Fig. 1 illustrates the simple case where an RTE assertion (i.e., $p_1$) can be validated through specifications (i.e., $p_2$) *localized to its host function* (`abs`). This approach with localized specifications is, however, insufficient for ensuring RTE-freeness in real-world programs featuring *complex call hierarchies*. Consider the program `id.c` in Fig. 2 (a), which defines a function `id` that returns the identical value of its input $x$, alongside functions `one` and `zero` that invoke `id` with arguments `1` and `0`, respectively. The `main` function calls both `one` and `zero`. Static analysis reveals a potential division-by-zero UB in function `one`, flagged by the assertion $p_4 = \langle \phi_4, \iota_4 \rangle$ in Fig. 2 (b) (with $\phi_4$ = `x != 0`). Unlike the assertion $p_1$ in Fig. 1, constructing any precondition for `one` does not suffice to validate $p_4$, as `one` lacks parameters and thus imposes no constraints on its inputs.

To eliminate such false alarms (e.g., $p_4$), which frequently arise in practice, it is necessary to employ *interprocedural specifications* – specifications that extend beyond the host functions of

[1] Given a function `f`, we use $\iota_{\text{pre}}^{\text{f}}$ and $\iota_{\text{post}}^{\text{f}}$ to denote the first instruction after the entry point of `f` and a null instruction representing the termination of `f`, respectively. The call-site preconditions in Fig. 1 (c) are explained in Section 3.1.




(a) program `id.c` (b) analysis result of `id.c` (c) $\phi_4$ is valid under postcondition $\phi_5$ (d) precondition $\phi_6$ that induces a false alarm

Fig. 2. The dual role of interprocedural specifications: postconditions can help discharge false RTEs (c) while over-constrained preconditions can induce false alarms (d).

target assertions. For instance, the postcondition $p_5 = \langle \phi_5, \iota_{\text{post}}^{\text{id}} \rangle$ for function `id` in Fig. 2 (c) (with $\phi_5$ = `\result == x`) serves precisely this role. Specifically, the value of `x` at $\iota_4$ is determined by argument `1` and the postcondition of `id` at the call site `id(1)`, thereby validating $p_4$ , i.e., $p_5 \models p_4$.

The synthesis of interprocedural specifications, as exemplified by postcondition $p_5$, depends on LLMs comprehending interprocedural program contexts that span across multiple functions. Specifically, LLMs must realize that discharging assertion $p_4$ in function `one` necessitates constructing a postcondition for function `id`. Nonetheless, current approaches to interprocedural specification synthesis, e.g., [Ma et al. 2025; Wen et al. 2024], exhibit two key limitations: (i) They provide LLMs with the entire program as context, which is constrained by the *finite context window* of LLMs and thus do not scale to large programs; (ii) Even for small programs with full context, LLMs may generate *over-constrained preconditions* due to *hallucination* [Huang et al. 2025], thus yielding spurious false alarms. As illustrated in Fig. 2 (d), an LLM might be misled by the context "`x != 0` ($\phi_4$) and `1/x`" at $\iota_4$, and thus forge an over-constrained precondition $p_6 = \langle \phi_6, \iota_{\text{pre}}^{\text{id}} \rangle$ for `id` (with $\phi_6$ = `x != 0`), which triggers a false alarm at the call site $\iota_5$ (`id(0)`). Although discarding over-constrained preconditions while retaining postconditions may resolve immediate false alarms (as in Fig. 2 (c)), determining whether a precondition is over-constrained is per se a nontrivial task (cf. Fig. 1 (c) vs. Fig. 2 (d)) [Cousot et al. 2013, 2011]: In Fig. 1 (c), indiscriminately discarding the precondition compromises soundness (as true RTEs are missed). In a nutshell, we observe

**Challenge 1.** In the presence of long-context reasoning limitations, how to instruct LLMs to access particular program contexts required for generating interprocedural specifications for large-scale programs with complex call hierarchies?

**Challenge 2.** How to design a fine-grained interprocedural-specification synthesis mechanism – e.g., determining which specifications to generate, at which program locations, and in what sequence – to alleviate the synthesis of over-constrained preconditions[2]?

### 3.3 Feedback-Driven Specification Refinement

Section 3.2 has demonstrated the challenges in constructing interprocedural specifications; however, validating RTE assertions for individual functions can also be difficult. Consider, for example, the

[2]While over-constrained specifications beyond preconditions (e.g., postconditions and loop invariants) can also lead to issues such as false negatives, these can be addressed through simpler mechanisms, as discussed in Section 5.2.3.

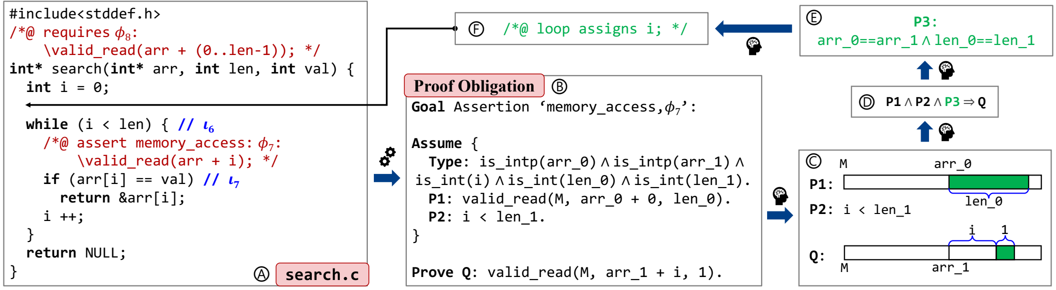



Fig. 3. Handcrafting missing specifications via verifier feedback (e.g., proof obligations) for validating $p_7$.

program `search.c` in Fig. 3 Ⓐ, which defines a function `search` that iterates through an integer array `arr` of length `len` and returns either the address of the first element matching the target value `val`, or `NULL` if no such element exists. Static analysis reveals a potential invalid-memory-access UB, flagged by the assertion $p_7 = \langle \phi_7, \iota_7 \rangle$ in Ⓐ (where $\phi_7 =$ `\valid_read(arr + i)`. Guided by $p_7$, LLMs can generate a precondition $p_8 = \langle \phi_8, \iota_{\text{pre}}^{\text{search}} \rangle$, where $\phi_8 =$ `\valid_read(arr + (0..len-1))` asserts read validity for the entire array `arr` with length `len`. Counterintuitively, the verifier fails to certify the validity of accessing `a[i]` (i.e., marking $p_7$ at $\iota_7$ as *unknown*) under the hypothesis $p_8$.

This behavior stems from the fact that modern deductive verifiers employ elaborate mechanisms to handle complex C semantics (e.g., memory access at $\iota_7$ in Ⓐ). This necessitates *understanding the verifier's internal mechanisms and verification details* when validating RTE assertions involving such semantic features. Fig. 3 Ⓑ-Ⓕ demonstrates a step-by-step process by a verification expert to construct missing specifications for discharging $p_7$ leveraging *verifier feedback*, including the proof obligation of $p_7$ in Ⓑ. Specifically, the obligation comprises hypotheses **P1** (precondition $p_8$), **P2** (loop condition `i < len` at $\iota_6$), and type declarations of integer pointers (`arr_0`, `arr_1`) and integer variables (`i`, `len_0`, `len_1`), associated with the goal **Q** to be verified. Here, **Q** is a transformed version of $p_7$ that is used internally by Frama-C/Wp. In **Q**, `M` denotes a simplified "typed" memory model (for integers in this case), as outlined in [Kosmatov et al. 2024, Sect. 4.3.4], where each memory block has the size of an `int` (i.e., 4 bytes), forming a region storing integer arrays such as `arr_0` and `arr_1`. The variable `arr_1` represents the base address of array `arr` at $\iota_7$. It is distinguished from `arr_0` – the base address of `arr` in precondition $p_8$ – because `arr` is mutable and Frama-C/Wp assumes by default that it may be modified inside loops. At this stage, the assertion $p_7$ remains unproven because **P1** ∧ **P2** ⇒ **Q** cannot be established by external solvers.

The verification expert can model the semantics of **P1** and **Q** through memory representations as depicted in Ⓒ. **P1**: `valid_read(M,arr_0+0,len_0)` denotes read validity for a memory block starting at `arr_0` spanning `len_0` elements, whilst **Q**: `valid_rd(M,arr_1+i,1)` asserts read validity for a single element at `arr_1+i`. Although no solver proves **P1** ∧ **P2** ⇒ **Q**, experienced practitioners may strengthen the *antecedent* by conjoining **P3** and evaluating **P1** ∧ **P2** ∧ **P3** ⇒ **Q** (see Ⓓ). The missing predicate **P3** is immediately derived (based on verification expertise) as `arr_0 == arr_1 ∧ len_0 == len_1` (Ⓔ). This reveals the verifier's over-approximation of side effects – assuming that variables defined outside the loop, such as array address `arr` and length `len`, could be modified inside the loop scope. Consequently, adding the specification `loop assigns i` (denoted by $p_9$) becomes essential to restrict reassignment exclusively to variable `i` (Ⓕ). Inserting this specification

above the loop condition ($\iota_6$) enables verifiers to eliminate the false invalid-memory-access alarm, i.e., $\{p_8, p_9\} \models p_7$ as per (3). This gives rise to the idea of *feedback-driven specification refinement*:

**Insight 2.** The feedback from a verifier, in particular, the proof obligations that remain to be discharged, can be leveraged to refine the specifications generated by LLMs effectively.

## 4 Problem Formulation

This section formalizes the problem of potential RTE-guided specification synthesis. To this end, we first introduce a formal characterization of RTE-freeness:

**Definition 10 (RTE-Freeness).** *Given a program prog, a set of RTE assertions $\mathcal{A}$, and a set of specifications $\mathcal{S}$, we say that prog is* free-of-RTE *if and only if*

$$\forall p \in \mathcal{A} \cup \mathcal{S} \colon \; \mathcal{A} \cup \mathcal{S} \setminus \{p\} \;\models\; p\,, \tag{8}$$

*namely, the properties in $\mathcal{A} \cup \mathcal{S}$ are mutually consistent – every property in the set is valid under the rest. More generally, prog is said to be* free-of-RTE under hypotheses $\mathcal{H} \subseteq \mathcal{A} \cup \mathcal{S}$ *if and only if*

$$\forall p \in \mathcal{A} \cup \mathcal{S} \setminus \mathcal{H} \colon \; \mathcal{A} \cup \mathcal{S} \setminus \{p\} \;\models\; p\,. \tag{9}$$

*Note that* (8) *is a special case of* (9) *with $\mathcal{H} = \emptyset$.*

It follows that the characterization (9) boils down to (8) if the hypotheses $\mathcal{H}$ are discharged:

**Theorem 11 (RTE-Freeness under Valid Hypotheses).** *Suppose prog is free-of-RTE under hypotheses $\mathcal{H}$ as per* (9). *If every $h \in \mathcal{H}$ is valid, i.e., $\models h$, then prog is free-of-RTE in the sense of* (8).

**Example 12.** Recall the three real-world scenarios of RTE-freeness verification in Figs. 1 to 3:

The program `abs.c` in Fig. 1 is free-of-RTE under hypothesis $\{p_2\}$. The precondition $p_2$ facilitates detecting a real RTE in trace $\tau_3$ while safeguarding trace $\tau_2$. With $p_2$, one can correct the program by removing the instruction `abs(INT_MIN)` ($\iota_3$) or revising the implementation of `abs`. This exemplifies a classic *source-to-sink* problem [Wang 2025] in static (taint) analysis, where the goal is to identify that `abs(INT_MIN)` at $\iota_3$ (source) triggers an overflow RTE at $\iota_1$ (sink). Potential RTE-guided specification synthesis provides a deductive verification mechanism for solving such problems.

The program `id.c` in Fig. 2 is free-of-RTE. The postcondition $p_5$ suffices to establish $\models p_5 \models p_4$. This represents the ideal case for RTE-freeness verification, where all alarms raised by static analyzers are false positives and can be eliminated by deductive verifiers with appropriate specifications.

The program `search.c` in Fig. 3 is free-of-RTE under hypotheses $\{p_8, p_9\}$. Unlike `abs.c`, precondition $p_8$ cannot be used to locate RTEs directly, as no caller function to `search` exists in the program. This case depicts the scenario of constructing *summaries* for modules in large projects, known as *modular static program analysis* [Cousot and Cousot 2002]. Preconditions like $p_8$ provide assurances that modules such as `search` are free-of-RTE if all callers adhere to these preconditions. ◁

The *potential RTE-guided specification synthesis problem* concerned in this paper reads as follows:

**Problem Statement.** Given a program *prog* with a set of RTE assertions $\mathcal{A}$ annotated by an abstract interpretation-based static analyzer, automatically generate a set of specifications $\mathcal{S}$ by leveraging a large language model *LLM*, and prove with a deductive verifier *Verify* that *prog* is either (i) free-of-RTE, or (ii) free-of-RTE under hypotheses $\mathcal{H} \subseteq \mathcal{A} \cup \mathcal{S}$.

Note that case (ii) predominates in the verification of large-scale, real-world programs. The presence of a hypothesis $h \in \mathcal{H}$ typically stems from one of the four scenarios: (a) $h$ signifies a genuine RTE, which inherently cannot – and should not – be validated by any sound verifier; (b) $h$ is a false alarm

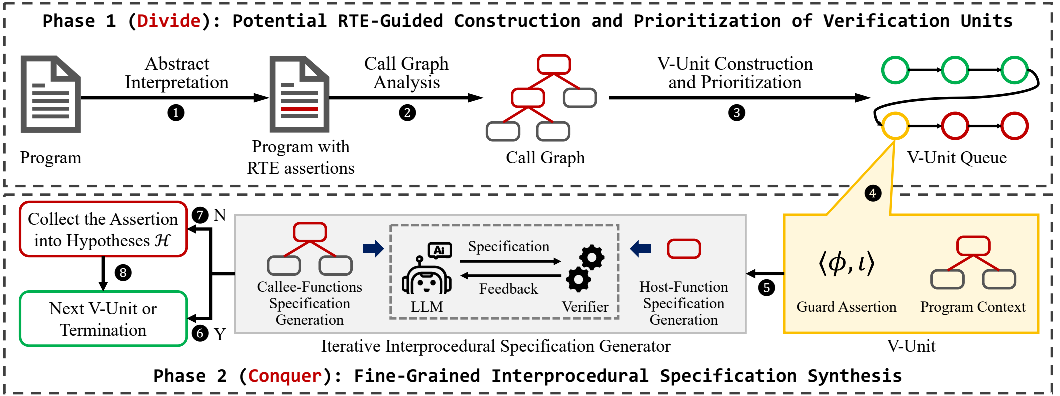



Fig. 4. Divide-and-conquer architecture of the PREGUSS framework.

that fails to be discharged due to resource exhaustion or inherent solver limitations; (c) $h$ is a false alarm resulting from an over-constrained precondition generated by the approach; or (d) $h$ is a false alarm, but the specification-synthesis framework does not suffice to eliminate it – for instance, generating precise specifications for behaviors involving complex data structures and/or intricate memory models remains notoriously hard; see, e.g., [Djoudi et al. 2021] for dedicated research.

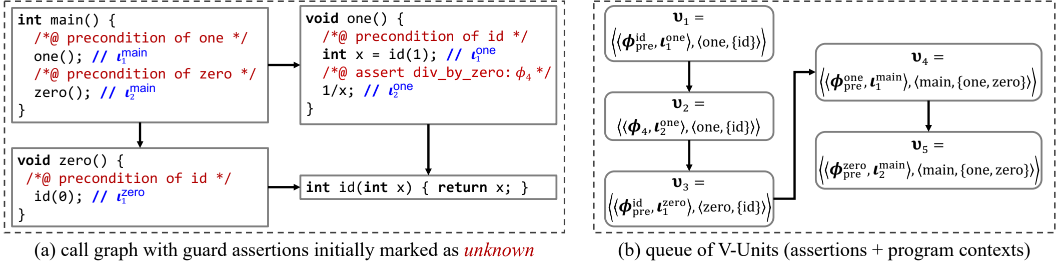
A practical strategy for addressing these scenarios involves manual validation of the hypotheses in $\mathcal{H}$ (as justified by Theorem 11), supplemented by revising generated specifications (particularly for scenario (c)) and constructing additional specifications (particularly for scenario (d)). This approach proves substantially more feasible than attempting to verify RTE-freeness against the entire set of RTE assertions $\mathcal{A}$ produced by the static analyzer, as demonstrated in Section 6 (RQ2).

## 5 Methodology

This section presents PREGUSS – our framework for Potential Runtime Error-GUided Specification Synthesis. As sketched in Fig. 4, PREGUSS adopts a *divide-and-conquer* strategy (in a fashion similar to [Zheng et al. 2025]) which operates through two synergistic phases: Phase 1 (*Divide*) decomposes the monolithic task of RTE-freeness verification into manageable, prioritized units. Each unit contains a guard assertion together with its necessary program context for validation; Phase 2 (*Conquer*) employs an LLM to infer interprocedural specifications along caller-callee chains, thereby certifying the target assertion in each verification unit. Below, we show how these two phases cooperate to enable RTE-freeness verification of real-world programs with complex call hierarchies.

### 5.1 Phase 1: Potential RTE-Guided Construction and Prioritization of Verification Units

To address Challenge 1 (long-context reasoning limitation of LLMs), Phase 1 decomposes the monolithic RTE-freeness verification task into a sequence of manageable *verification units* (V-Units), as illustrated in Fig. 4 (upper part). The process begins by employing abstract interpretation conducted by static analyzers (e.g., FRAMA-C/RTE [Herrmann and Signoles 2025] and FRAMA-C/EVA) to generate RTE assertions signifying *all* potential UBs in the source program (❶). Subsequently, the program's *call graph* is constructed while recording all of its call sites (❷), yielding a graph structure with a comprehensive set of *guard assertions*. For each assertion, a *V-Unit* is created, encapsulating both the assertion and its essential contextual program slices. These V-Units are then prioritized into a queue to streamline the verification workflow (❸). Below, we first elaborate on the generation of guard assertions, which serve as sub-goals for the holistic RTE-freeness verification (Section 5.1.1) and then introduce the V-Unit structure along with prioritization principles (Section 5.1.2).




(a) call graph with guard assertions initially marked as *unknown* (b) queue of V-Units (assertions + program contexts)

Fig. 5. The call graph (a) and V-Unit queue (b) of program `id.c` in Fig. 2.

*5.1.1 Guard Assertion Generation.* As discussed in Section 3.1, both the RTE assertions emitted by static analyzers and the synthesized preconditions act as *guard assertions* to ensure RTE-freeness of the source program. Whereas RTE assertions can be generated via abstract interpretation (❶), preconditions must be constructed on-the-fly during specification synthesis. However, we observe that validating a precondition ($p = \langle \phi, \iota_{\text{pre}}^{\text{f}} \rangle$) involves two key components: (i) the predicate $\phi$ describing expected states of the function, and (ii) all the call-site locations where the function is invoked. Thus, we statically extract all precondition check locations through call-graph analysis (❷) without generating actual specifications, initializing each predicate to the tautological `true` as a placeholder. The actual preconditions are generated and refined during Phase 2 (see Section 5.2). During Phase 1, the *location* of a precondition check is more critical than its predicate content as it determines the structure of the V-Unit queue and thus the order of subsequent verification.

**Remark.** The selection of *preconditions* (checked at call sites) as guard assertions, rather than postconditions or loop invariants, is motivated by their intrinsic role in RTE-freeness verification. Preconditions act as natural guards at function entry points, specifying requisite constraints on input parameters (e.g., value ranges, memory accessibility) (see Fig. 1 (c)). In contrast, postconditions and loop invariants capture semantic behaviors of programs or loops (cf. Fig. 1 (d) and Fig. 3). For RTE-freeness, the goal is to derive (weakest) preconditions that encompass all valid program traces while excluding erroneous ones. Consequently, a violated precondition signals a potential RTE, whereas a violated postcondition or invariant is possibly invalid. This focus on preconditions also contributes to the termination guarantee of Preguss, as established in Section 5.3. ◁

Preguss deals with acyclic call graphs formally described as follows:

**Definition 13 (Call Graph).** *A* call graph *is a directed acyclic graph* $\mathcal{G} \triangleq \langle F, E \rangle$, *where*

- $F = \{f_1, \ldots, f_n\}$ *is a finite set of* function nodes,
- $E \subseteq F \times F$ *is a set of* call edges, *with* $\langle f_i, f_j \rangle \in E$ *indicating* $f_i$ *calls* $f_j$.

*The* callees *of a function* $f \in F$ *are collected into*

$$callee(f) \triangleq \{ f' \mid \langle f, f' \rangle \in E \} .$$

*Moreover, the* ancestor *relation is defined as*

$$ancestor \triangleq \Big\{ \langle f_{anc}, f_{des} \rangle \in F \times F \mid \exists f_i, \ldots, f_j \colon f_i \in callee(f_{anc}) \wedge f_{i+1} \in callee(f_i) \wedge \ldots \wedge f_j \in callee(f_{j-1}) \wedge f_{des} \in callee(f_j) \Big\} .$$

*Intuitively,* $\langle f_{anc}, f_{des} \rangle \in ancestor$ *means that there exists a finite call chain from* $f_{anc}$ *to* $f_{des}$.

**Example 14.** Fig. 5 (a) illustrates the call graph of program `id.c` in Fig. 2. The division-by-zero RTE assertion $p_4 = \langle \phi_4, \iota_2^{\text{one}} \rangle$ as well as all the call-site precondition checks (as placeholders) –

namely $\langle \phi^{\mathrm{id}}_{\mathrm{pre}}, \iota^{\mathrm{one}}_1 \rangle$, $\langle \phi^{\mathrm{id}}_{\mathrm{pre}}, \iota^{\mathrm{zero}}_1 \rangle$, $\langle \phi^{\mathrm{one}}_{\mathrm{pre}}, \iota^{\mathrm{main}}_1 \rangle$, and $\langle \phi^{\mathrm{zero}}_{\mathrm{pre}}, \iota^{\mathrm{main}}_2 \rangle$ – are initially marked as *unknown*. Note that each precondition predicate is initialized to tautological `true`, which represents the weakest possible constraint, i.e., no semantic restrictions at the initial stage. ◁

**Remark.** PREGUSS admits, in its current implementation, merely *acyclic* call graphs. This is because the underlying deductive verifier FRAMA-C/WP we use lacks comprehensive support for verifying recursive and mutually recursive functions [Blanchard 2020, Sect. 4.4.2]. Crucially, this acyclicity assumption greatly simplifies our methodological design of the V-Unit ordering (see Definition 16) and the interprocedural specification-synthesis mechanism (see Section 5.2), as it circumvents the complex dependencies in (mutual) recursions and their specifications. We foresee that PREGUSS could be extended to cope with cyclic call graphs by interfacing with verifiers that support recursion while adapting the methodological design, though such adaptations may increase the risk of generating over-constrained preconditions for recursive functions (as outlined in Section 3.2). ◁

*5.1.2 V-Unit Initialization and Prioritization.* PREGUSS leverages the *modularization principle of deductive verification* [Ahrendt et al. 2016, Chap. 9], whereby each function is verified separately. Specifically, to verify that a function adheres to all its specifications except preconditions[3], one needs *not* the entire program but essentially two elements: (i) the function's own implementation, and (ii) the contracts of all its callees (which are recursively derived from further callees along the call chains in a *bottom-up* manner). Consequently, to validate an individual RTE assertion $\alpha$, it suffices to extract a *minimally constrained program context* consisting of the host function of $\alpha$ and all its callees. *This design prevents the verification context from growing substantially during the bottom-up verification, thereby enabling scalability to large-scale programs.*

PREGUSS encapsulates these minimally constrained program contexts into dedicated V-Units:

**Definition 15 (V-Unit).** *A* V-Unit *is a pair $\langle \alpha, P_{context} \rangle$, where $\alpha = \langle \phi, \iota \rangle$ is a guard assertion and $P_{context} \triangleq \langle f, callee(f) \rangle$ is a two-layer program slice containing the host function $f$ of $\alpha$ and its callees.*

PREGUSS prioritizes all V-Units into a queue (❸ of Fig. 4) according to a total order $\sqsubseteq$:

**Definition 16 (Order over V-Units).** *Given two V-Units $v_i = \langle \langle \phi_i, \iota_i \rangle, \langle f_i, callee(f_i) \rangle \rangle$ and $v_j = \langle \langle \phi_j, \iota_j \rangle, \langle f_j, callee(f_j) \rangle \rangle$, we have $v_i \sqsubseteq v_j$ if and only if*

$$\underbrace{\langle f_j, f_i \rangle \in ancestor}_{\text{Condition 1: bottom-up verification principle}} \vee \underbrace{\left( \langle f_j, f_i \rangle \notin ancestor \wedge \langle f_i, f_j \rangle \notin ancestor \wedge loc(\iota_i) \le loc(\iota_j) \right)}_{\text{Condition 2: total order enforcement}}, \quad (10)$$

*where $loc(\iota)$ denotes the location (i.e., line number) of instruction $\iota$.*

The design principles underneath the total order $\sqsubseteq$ are embodied in Conditions 1 and 2 of (10). Condition 1 captures the bottom-up verification progression [Wen et al. 2024]: Since verifying a function requires the contracts of its callees, functions higher in the call hierarchy (ancestors) must be verified after their callees. Condition 2 serves two primary purposes. First, for guard assertions $\alpha_i$ and $\alpha_j$ within the same function, it is common that one assertion acts as a hypothesis for the other, e.g., $\mathcal{H} \models \alpha_j$ with $\alpha_i \in \mathcal{H}$. In this case, validating $\alpha_i$ first may generate specifications that facilitate the verification of $\alpha_j$. These hypothesis assertions typically appear at earlier program locations if `goto` statements and loop structures are disregarded (all properties within loops are mutually dependent and thus receive equal priority). Second, for properties from functions in disjoint call chains (i.e., $\langle f_j, f_i \rangle \notin ancestor \wedge \langle f_i, f_j \rangle \notin ancestor \wedge f_i \neq f_j$), their relative priority is arbitrary. The clause $loc(\iota_i) \le loc(\iota_j)$ in Condition 2 ensures that $\sqsubseteq$ is a total order, simplifying its implementation. Fig. 5 (b) demonstrates the construction of five V-Units derived from the guard assertions in Fig. 5 (a) and their prioritization into a verification queue as per the order $\sqsubseteq$.

[3] A function's preconditions are validated at its *call sites*, as part of the verification obligation for its callers (see Section 5.1.1).

**Algorithm 1:** Fine-Grained Interprocedural Specification Synthesis

**Input:** $Q_v$: V-Unit queue, *prog*: source program.
**Output:** $\mathcal{S}$: synthesized specifications, $\mathcal{V}$: verified properties, $\mathcal{H}$: *unknown* hypotheses.
**Parameters:** Verify: an extended sound verifier (as per (7)), *iter*: maximum iterations.

1 $\mathcal{S} \leftarrow \emptyset,\ \mathcal{V} \leftarrow \emptyset,\ \mathcal{H} \leftarrow \emptyset$ ; ▷ initialization
2 **while** $Q_v$ is not empty **do**
3   $\langle \alpha, \langle f, callee(f) \rangle \rangle \leftarrow \text{pop}(Q_v),\ \mathcal{S}' \leftarrow \emptyset$ ; ▷ pop the next V-Unit on the queue
4   **for** $i$ from 1 to *iter* **do**
5     $\langle \text{status}, \text{feedback} \rangle \leftarrow \text{Verify}(prog, \mathcal{V} \cup \mathcal{H} \cup \mathcal{S}', \alpha)$ ; ▷ apply sound verifier
6     **if** status = *unknown* **then** ▷ refined specification synthesis
7       $\mathcal{S}' \leftarrow \mathcal{S}' \cup \text{generate_host_spec}(\alpha, \{f\}, \mathcal{V} \cup \mathcal{H}, \text{feedback})$;
8       $\mathcal{S}' \leftarrow \mathcal{S}' \cup \text{generate_callees_spec}(\alpha, \{f\} \cup callee(f), \mathcal{V} \cup \mathcal{H}, \text{feedback})$;
9       $\mathcal{S}' \leftarrow \text{syntax_and_semantics_check}(\mathcal{S}', prog)$;
10     **else**
11       **break**;
12   **if** $\text{Verify}(prog, \mathcal{V} \cup \mathcal{H} \cup \mathcal{S}', \alpha) = \langle true, _ \rangle$ **then** ▷ $\alpha$ is validated
13     $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}',\ \mathcal{P} \leftarrow \text{extract_preconditions}(\mathcal{S})$ ; ▷ collect specifications
14     $Q_v \leftarrow \text{update}(Q_v, \mathcal{P}),\ \mathcal{V} \leftarrow \mathcal{V} \cup \{\alpha\} \cup (\mathcal{S}' \setminus \mathcal{P})$ ; ▷ update $Q_v$ and $\mathcal{V}$ accordingly
15   **else**
16     $\mathcal{H} \leftarrow \mathcal{H} \cup \{\alpha\}$ ; ▷ augment the hypothesis set with $\alpha$
17 **return** $\langle \mathcal{S}, \mathcal{V}, \mathcal{H} \rangle$;

### 5.2 Phase 2: Fine-Grained Interprocedural Specification Synthesis

To address Challenge 2 (fine-grained synthesis), Phase 2 employs a fine-grained strategy, for each V-Unit, to generate interprocedural specifications, as depicted in Fig. 4 (lower part). For each V-Unit $v = \langle \alpha, \langle f, callee(f) \rangle \rangle$, Preguss utilizes an iterative approach (❺) comprising two core components: (i) an LLM agent producing specification candidates driven by verifier feedback on guard assertion $\alpha$, and (ii) a verifier validating these candidates both syntactically and semantically, thereby ensuring the soundness of Preguss. Each iteration generates interprocedural specifications tailored to the program context $\langle f, callee(f) \rangle$ in a refined manner (see details below). Upon successful verification of $\alpha$, Preguss advances to the next V-Unit in the queue or terminates if $v$ is the final unit (❻). If Preguss fails to generate specifications sufficient to validate $\alpha$, the guard assertion is designated as a hypothesis requiring manual review upon termination (❼). Then, by assuming the validity of $\alpha$, Preguss continues processing subsequent V-Units until the queue is exhausted (❽).

Algorithm 1 elaborates the workflow of Phase 2. It takes as input the V-Unit queue $Q_v$ from Phase 1, along with the source program *prog*, and yields as output the synthesized specifications $\mathcal{S}$, alongside the verified properties $\mathcal{V}$ and the *unknown* hypotheses $\mathcal{H}$ that require manual review. Key parameters to the algorithm include an extended sound verifier (see (7)) $\text{Verify}\colon (prog, \mathcal{H}, q) \mapsto \langle \text{status}, \text{feedback} \rangle$ – which returns both verification status and auxiliary feedback – and the hyper-parameter *iter* that controls the maximum number of refinement iterations. The algorithm initializes $\mathcal{S}$, $\mathcal{V}$ and $\mathcal{H}$ to empty sets (Algorithm 1) and processes each V-Unit in $Q_v$ in sequence (Lines 2–16). For each V-Unit $v = \langle \alpha, \langle f, callee(f) \rangle \rangle$, a temporary specification set $\mathcal{S}'$ is initialized to accumulate newly generated properties (Algorithm 1). Within the refinement loop (Lines 4–11), Preguss first applies a sound verifier to obtain the status and feedback for the guard assertion $\alpha$ (Algorithm 1). If $\alpha$ is *unknown* (Algorithm 1), Preguss leverages the feedback to (i) *generate preconditions and loop invariants for the host function $f$* (Algorithm 1, detailed in Section 5.2.1), and (ii) *generate*

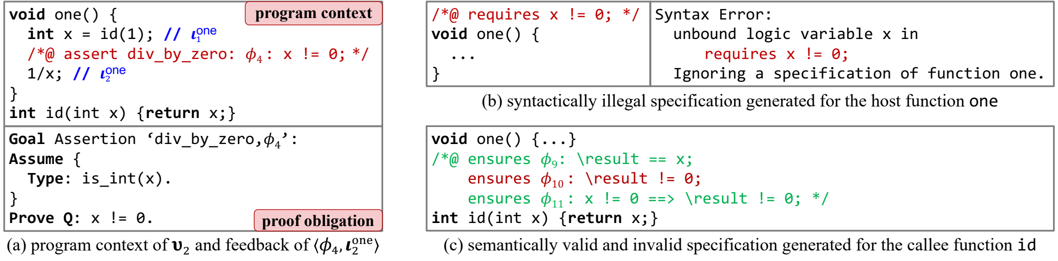



Fig. 6. Fine-grained interprocedural specification synthesis for V-Unit $\upsilon_2$ in Fig. 5.

*postconditions for the callee functions* $callee(f)$ (Algorithm 1, detailed in Section 5.2.2). For all these freshly generated specifications, PREGUSS employs a correction and refinement mechanism to enhance their quality and ensure syntactic and semantic validity (Algorithm 1, detailed in Section 5.2.3). PREGUSS exits the loop once $\alpha$ is proven (Algorithm 1) or the iteration limit *iter* is reached. Finally, if $\alpha$ is certified as *true*, PREGUSS appends the generated specifications $\mathcal{S}'$ to $\mathcal{S}$, and extracts the preconditions $\mathcal{P}$ from the $\mathcal{S}'$ (Algorithm 1). As $\mathcal{P}$ will be validated at $f$'s call sites, PREGUSS then updates the corresponding guard assertions – which are precondition checks at these call sites – in $Q_v$ with $\mathcal{P}$, and incorporates all the newly verified properties – $\alpha$ and generated specifications $\mathcal{S}'$ except for preconditions $\mathcal{P}$ – into $\mathcal{V}$ (Algorithm 1). Otherwise, assertion $\alpha$ is added to the hypothesis set $\mathcal{H}$ (Algorithm 1).

*5.2.1 Host-Function Specification Generation.* When the verifier fails to validate a guard assertion $\alpha$, PREGUSS leverages LLMs to generate additional specifications by incorporating verifier feedback and the relevant program context, including function $f$ and existing specifications $\mathcal{V} \cup \mathcal{H}$. A simplified version of the prompt we use to instruct the LLM is as follows:

**Simplified Prompt for Host-Function Specification Generation**

You are an expert in Frama-C. I will provide a C program specified with ACSL, including an unproven guard assertion, alongside feedback in the form of proof obligations from the FRAMA-C/WP verifier. **Your tasks:** (i) Diagnose the root cause of FRAMA-C/WP's failure to verify the assertion based on the provided feedback; (ii) Generate necessary specifications – such as preconditions, `assigns` clauses, and loop invariants – to enable successful verification.

- **Program context:** $\{f, \mathcal{V} \cup \mathcal{H}\}$
- **Target guard assertion:** $\{a\}$
- **Proof obligation for the assertion:** {feedback}

PREGUSS employs a *chain-of-thought* prompting strategy [Wei et al. 2022], which instructs the LLM to first reason about the verification failure (à la Insight 2 on verifier's feedback) before proposing new specifications. Besides, PREGUSS integrates a few specified programs in the prompt as few-shot examples to improve the quality of generated specifications [Brown et al. 2020]. Notably, the synthesis is strictly confined to the local context of function $f$, excluding its call-site information to *prevent over-constrained preconditions* – a critical safeguard as illustrated in Section 3.2.

**Example 17.** Fig. 6 (a) displays the program context of V-Unit $\upsilon_2$ introduced in Fig. 5, along with the proof obligation for guard assertion $\langle \phi_4, \iota_2^{one} \rangle$. Inspired by the proof goal **Q**: `x != 0`, PREGUSS hypothesizes a precondition `requires x != 0` for the host function one, as shown in Fig. 6 (b). Although this candidate is syntactically illegal initially (as variable x is unbound in the scope of function one), it undergoes an automated correction via the mechanism detailed in Section 5.2.3. ◁

*5.2.2 Callee-Functions Specification Generation.* In this stage, PREGUSS prompts the LLM using a similar approach to that described in Section 5.2.1, with two key distinctions: (i) the LLM is provided with the complete program context $\langle f, callee(f)\rangle$ (as opposed to only $f$ in the host-function stage), and (ii) the LLM is instructed to generate *postconditions* (and loop invariants) exclusively for the callee functions $callee(f)$. This design addresses the scenarios illustrated in Section 3.2, where validating a guard assertion in function $f$ requires interprocedural specifications – particularly postconditions – from its callees. Furthermore, PREGUSS *explicitly prohibits the generation of preconditions* for $callee(f)$ to align with the host-function specification mechanism (detailed in Section 5.2.1), thereby minimizing the risk of synthesizing over-constrained preconditions.

**Example 18.** Consider the program in Fig. 6 (c), where the LLM produces three postcondition candidates at this stage: $\langle \phi_9, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$, $\langle \phi_{10}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$, and $\langle \phi_{11}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$. Here, $\langle \phi_9, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ accurately captures the behavior of function `id`, while $\langle \phi_{10}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ and $\langle \phi_{11}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ are derived from the proof goal $\mathbf{Q}$ in Fig. 6 (a). Although all three candidates suffice to validate the assertion $\langle \phi_4, \iota^{\mathtt{one}}_2\rangle$, the postcondition $\langle \phi_{10}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ is semantically invalid, which will be screened out as discussed in Section 5.2.3. ◁

*5.2.3 Syntactic and Semantic Validity Check of Specifications.* For all the specification candidates generated in Sections 5.2.1 and 5.2.2, PREGUSS employs the verifier to (i) correct syntactically illegal specifications, and (ii) filter out semantically unsatisfiable properties.

When a candidate is syntactically invalid, PREGUSS queries the LLM with verifier-supplied error messages to iteratively correct its syntax. If the candidate remains illegal after several correction attempts, it is discarded. For instance, Fig. 6 (b) shows that the precondition `requires x != 0` triggers a syntax error due to variable unboundedness. Indeed, any predicate containing variables is invalid for this precondition, as `one` has no input parameters. Consequently, PREGUSS fails to correct this candidate and hence removes it.

For an *unknown* specification that cannot be verified – and is not a precondition (see Section 5.1.2) – PREGUSS immediately removes it to avoid inducing false negatives. Consider a pathological example: a loop invariant with predicate `false` would spuriously validate any property $p$ within the loop, since `false` $\models p$ trivially holds. To prevent such false negatives, all *unknown* non-precondition specifications are discarded. In contrast, an *unknown* precondition represents a guard assertion indicating a potential RTE (as discussed in Sections 3.1 and 3.2); it is thus retained in the V-Unit queue to be checked later at the corresponding call sites in caller functions.

**Example 19.** We illustrate the overall Phase 2 of PREGUSS using the V-Unit queue in Fig. 5 (b): The first V-Unit $v_1$ is verified immediately without additional specifications, as its predicate $\phi^{\mathtt{id}}_{\mathrm{pre}}$ is initialized to `true`. PREGUSS then processes $v_2$, validating it via the postconditions $\langle \phi_9, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ and $\langle \phi_{11}, \iota^{\mathtt{id}}_{\mathrm{post}}\rangle$ generated via steps illustrated in Examples 17 and 18. The subsequent V-Units $v_3$, $v_4$, and $v_5$ (pertaining to call sites of `id`, `one`, and `zero`, respectively) are verified immediately since none of these functions require preconditions, and thus no additional specifications are generated. ◁

### 5.3 Soundness and Termination of PREGUSS

This section establishes formal guarantees of PREGUSS, including *soundness* and *termination*.

**Theorem 20 (Soundness).** *Given a program prog, suppose* PREGUSS *annotates prog with a set of RTE assertions $\mathcal{A}$ and returns synthesized specifications $\mathcal{S}$ together with verified properties $\mathcal{V}$ and unknown hypotheses $\mathcal{H}$. Then, prog is free-of-RTE under hypotheses $\mathcal{H}$ (in the sense of (9)).*

A formal proof of Theorem 20 is provided in [Wang et al. 2025a, Appendix A]. Intuitively, if (i) $\mathcal{H}$ is empty or all hypotheses in $\mathcal{H}$ are validated (via manual reviews or auxiliary verification tools), and (ii) the underlying abstract interpretation-based static analyzer is sound (i.e., it misses

no potential RTEs), then Preguss guarantees that the source program *prog* contains no runtime errors (by Theorem 11). In that case, all the RTE alarms in $\mathcal{A}$ emitted by the sound analyzer are false positives.

With this soundness assurance, Preguss reduces the task of RTE-freeness verification from *reviewing every analyzer-emitted alarm* to *validating each hypothesis generated by* Preguss, which, in practice, demands significantly less manual effort; see experimental results in Section 6.

In addition to soundness, Preguss exhibits termination – another essential property for practical verification systems. The following termination statement rests on two reasonable assumptions: (i) both abstract interpretation and call-graph analysis terminate successfully, and (ii) every LLM query terminates either naturally or upon reaching a prescribed time limit.

**Theorem 21 (Termination).** *The* Preguss *procedure always terminates.*

The following design principles constitute a proof sketch of Theorem 21:

First, the *V-Unit queue is constructed statically* during Phase 1, before the on-the-fly specification synthesis in Phase 2. The queue length is determined solely by the number of RTE assertions generated through abstract interpretation plus the number of call sites identified via call-graph analysis. Although guard assertions in V-Units corresponding to call-site precondition checks may be updated during Phase 2, the number of the V-Units remains invariant.

Second, the verification process exhibits a *one-way progression along the V-Unit queue*: Once a V-Unit is verified or marked as a hypothesis, its validity remains unchanged. As elaborated in Sections 5.2.1 and 5.2.2, for a V-Unit $v = \langle \alpha, \langle f, callee(f) \rangle \rangle$, Preguss prohibits the generation of preconditions for callee functions $callee(f)$ but only for the host function $f$. Such preconditions for $f$ are then used to update guard assertions in V-Units associated with callers of $f$. By Condition 1 of the ordering $\sqsubseteq$ (see Definition 16), these updated V-Units are positioned behind $v$ in the queue, thereby preserving the validity of all previously verified V-Units.

Third, *the verification of each individual V-Unit is guaranteed to terminate*. This is enforced by the hyper-parameter *iter*, which bounds the maximum number of feedback-driven refinement iterations. Moreover, each constituent step – host-function specification generation, callee-functions specification generation, and syntactic/semantic checking – is per se designed to terminate.

### 5.4 Generality of Preguss

Our Preguss framework can be integrated with other sound deductive verifiers supporting formal specification languages, such as OpenJML with JML for Java [Cok 2011] and the Move Prover with MSL for Move [Fu et al. 2025]. Although Preguss is initially designed for RTE-freeness verification and RTE identification, it can be extended to cope with other vulnerability classes and functional correctness. The key is to substitute RTE assertions with formal annotations flagging the target properties. The subsequent stages, commencing from ❷ in Fig. 4, remain fully generic to process these annotations. We prioritize RTE-freeness over full functional correctness because abstract interpretation-based static analysis for generating RTE assertions is mature and sound, while automatically constructing precise functional properties remains challenging [Djoudi et al. 2021].

## 6 Evaluation

The experimental evaluation is designed to address the following research questions (RQ):

**RQ1:** How does Preguss compare against state-of-the-art baseline approaches?
**RQ2:** How does Preguss perform on large-scale, real-world programs?
**RQ3:** To what extent does each component of Preguss contribute to its overall effectiveness?
**RQ4:** How sensitive is Preguss to the hyper-parameter *iter* and the underlying LLM?

Below, we first report the evaluation setup (Section 6.1), then address **RQ1**–**RQ4** (Sections 6.2 to 6.5) and conduct case studies (Section 6.6), and finally discuss threats to validity (Section 6.7).

### 6.1 Evaluation Setup

**Implementation.** We have implemented PREGUSS[4] in about 18,000 lines of Python code, which orchestrates the two-phase divide-and-conquer pipeline, plus 3,000 lines of C++ code for static analysis (e.g., call-graph construction and compilation-context extraction) using Clang.

We integrate PREGUSS with the deductive verifier FRAMA-C/WP [Blanchard 2020] and two abstract interpretation-based static analyzers: FRAMA-C/EVA [Bühler et al. 2025] and FRAMA-C/RTE [Herrmann and Signoles 2025]. These analyzers exhibit distinct characteristics: FRAMA-C/EVA is a fine-grained analyzer equipped with advanced abstract domains and data-flow analysis techniques, but it requires the source project to have a unique entry function accompanied by a *generalization mechanism*, e.g., manual assignment of over-approximation value intervals to external variables. In contrast, FRAMA-C/RTE is a coarse-grained analyzer capable of simultaneously analyzing multiple modules (each with its own entry function), yet often yields more RTE alarms than FRAMA-C/EVA.

**Configurations.** We repeat all experiments three times and report the averaged results to cope with the inherent uncertainty of LLMs. We use the LLM Claude 4 [Anthropic 2025] due to its exceptional performance in program comprehension (API: `claude-sonnet-4-20250514`; temperature: 0.7; max token: 4,096)[5] and set the hyper-parameter *iter* (i.e., maximum number of refinement iterations) to 2. For **RQ4** on sensitivity, we further examine GPT-5 [OpenAI 2025] (API: `gpt-5-20250807`; temperature: 0.7; max token: 4,096) and vary *iter* from 1 to 5. All the experiments are performed on a MacBook with an Apple M2 chip and 16 GiB RAM, except those for **RQ1**, which are conducted on a Ubuntu 24.04.2 LTS server with an AMD EPYC 7542 32-core processor and 16 GiB RAM. This is because the baseline approach AUTOSPEC (see below) relies on dependencies in Ubuntu.

**Baselines.** We evaluate PREGUSS against two baseline approaches: (i) AUTOSPEC [Wen et al. 2024], a state-of-the-art LLM-based method for automated specification generation; and (ii) an enhanced variant, $\text{AUTOSPEC}_{\text{RTE}}$, which integrates the abstract interpretation-based analyzer FRAMA-C/RTE to generate RTE assertions for the source program before invoking AUTOSPEC. Additional rationale for excluding other related works as baselines is provided in Section 8.

**Benchmarks.** We conduct evaluations on one C benchmark suite and four real-world C projects:

- Frama-C-Problems [Fra 2020; Wen et al. 2024] is an open-source benchmark suite used by AUTOSPEC, containing 51 small programs with an average of 17.43 lines of code. Each program has an individual function alongside a `main` entry function. All the programs have been previously verified using FRAMA-C/WP with 1∼3 ACSL specifications per program. The rationale for excluding other benchmarks employed by AUTOSPEC is provided in [Wang et al. 2025a, Appendix B].
- Contiki [Peyrard et al. 2018] is an open-source operating system for Internet of Things. We evaluate its encryption-decryption module called AES-CCM*, which consists of 544 LoC and 10 functions. Although this module has been manually verified using FRAMA-C/WP, the specifications used for its verification are not publicly available.
- X509-parser [Ebalard et al. 2019] is an open-source, RTE-free parser for X.509 format certificates. We evaluate its certificate-policy parsing module, which contains 1,199 LoC and 20 functions. This module has been manually verified with 251 ACSL specifications.

[4] The anonymized artifact for replicating the experimental results is available at https://zenodo.org/records/17296158.
[5] These settings match those of AUTOSPEC, with AUTOSPEC configured to use Claude 4 instead of its original model.

Table 1. Experimental result of PREGUSS v.s. baselines on Frama-C-Problems.

| Approach | Frama-C-Problems categorized per program features | | | | | | | | | | | | Overall (51) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | General (12) | | Pointers (8) | | Loops (8) | | Arrays (13) | | Arrays&Loops (5) | | Miscellaneous (5) | | | | | |
| | Num | Rate | Num | Rate | Num | Rate | Num | Rate | Num | Rate | Num | Rate | Num | Rate | Time (m) | Cost ($) |
| AUTOSPEC | 6 | 47.2% | 6 | 75.0% | 2 | 25.0% | 0 | 0.0% | 1 | 20.0% | 2 | 40.0% | 17 | 32.7% | 287.23 | **0.448** |
| AUTOSPEC$_{\text{RTE}}$ | 9 | 69.4% | **8** | **100.0%** | 1 | 12.5% | 0 | 0.0% | 2 | 40.0% | 2 | 40.0% | 22 | 41.8% | 284.40 | 0.455 |
| PREGUSS | **12** | **91.7%** | **8** | **100.0%** | **5** | **37.5%** | **11** | **69.2%** | **5** | **93.3%** | **5** | **100.0%** | **46** | **79.7%** | **65.59** | 2.721 |

- SAMCODE is a spacecraft sun-seeking control system containing 1,280 LoC and 48 functions (not open-sourced due to confidentiality constraints). It has not been previously verified.
- Atomthreads [Lawson 2015] is a real-time scheduler for embedded systems. We evaluate its kernel module, which consists of 1,451 LoC and 40 functions. To the best of our knowledge, this module has not been previously verified using FRAMA-C/WP.

**Remark.** To align with industrial verification practices, we adopt the following tactics for large-scale projects in **RQ2–RQ4**: (i) For SAMCODE, the complete software system with a unique entry function, we manually construct a generalization mechanism and analyze it using FRAMA-C/EVA; (ii) For Contiki, X509-parser, and Atomthreads, which are system modules requiring substantial manual effort for generalization, we directly apply FRAMA-C/RTE. For small programs in Frama-C-Problems used for **RQ1**, FRAMA-C/EVA alone suffices to establish RTE-freeness without any auxiliary specification or verifier. Therefore, we employ FRAMA-C/RTE to generate RTE assertions (all of which are false alarms) as verification targets, enabling a comparative evaluation of specification generation and RTE-freeness verification effectiveness between AUTOSPEC and PREGUSS. ◁

### 6.2 RQ1: Comparison against Baselines

Table 1 reports the experimental results of PREGUSS against AUTOSPEC and AUTOSPEC$_{\text{RTE}}$. Here, *Num* indicates the number of programs verified as RTE-free (marked as success) in at least one trial across repeated experiments, and *Rate* represents the average success rate per benchmark category. Additionally, we report the average execution time (in minutes) and LLM inference cost (in dollars) for evaluating each approach on the entire Frama-C-Problems benchmark suite. In general, we observe two remarkable disparities between PREGUSS and the baselines:

**Verification Success Rate.** *Overall,* PREGUSS *successfully verifies 46 out of 51 programs, achieving an average success rate of 79.7%, which substantially outperforms both* AUTOSPEC *(17/51, 32.7%) and* AUTOSPEC$_{\text{RTE}}$ *(22/51, 41.8%).* Notably, PREGUSS resolves all verification tasks except for a few cases in categories Loops and Arrays, which require complex (nested) loop invariants. These results suggest a promising future improvement of PREGUSS by incorporating advanced invariant synthesis techniques, e.g., [Pirzada et al. 2024; Wu et al. 2024b].

It is noteworthy that AUTOSPEC underperforms in RTE-freeness verification compared to its functional correctness results (31/51 programs as reported in [Wen et al. 2024, RQ1]) on the same benchmark suite. Besides LLM uncertainty, the primary performance gap lies in the Arrays[6] and Arrays&Loops categories. Here, AUTOSPEC verifies functional correctness for 12/18 cases [Wen et al. 2024] but succeeds in RTE-freeness for only 1/18 program (see Table 1). This discrepancy is exemplified by program `max.c` from the Arrays category, as depicted in Fig. 7. For `max.c`, which computes the maximum value in an array `a` of length `n`, AUTOSPEC generates four loop invariants prior to instruction $\iota_9$. Under the assumption that `max.c` is RTE-free (i.e., $\models \langle \phi_{12}, \iota_8 \rangle$ and $\models \langle \phi_{13}, \iota_{10} \rangle$),

[6] For simplicity, we consolidate the “immutable_arrays”, “mutable_arrays”, and “more_arrays” categories from [Wen et al. 2024] into the single Arrays category.

```
int amax(int *a, int n) {
  /*@ assert mem_access: φ12: \valid_read(a + 0); */
  int max = *(a + 0); // ι8
  /*@ loop invariant i <= n;
      loop invariant \forall integer k; 0 <= k < i ==> max >= a[k];
      loop invariant \exists integer k; 0 <= k < i && max == a[k];
      loop assigns max, i; */
  for (int i = 1; i < n; i++) // ι9
    /*@ assert mem_access: φ13:\valid_read(a + i); */
    if (max < *(a + i)) // ι10
      max = *(a + i);
  return max;
}
void main(void){
  int arr[5] = {1, 2, 3, 4, 5};
  int sum = amax(arr,5); // ι11
}
```

(a) AutoSpec fails to validate RTE assertions

```
/*@ requires φ15: n > 0 && \valid_read(a + (0 .. n-1)); */
int amax(int *a, int n) {
  /*@ assert mem_access: φ12: \valid_read(a + 0); */
  int max = *(a + 0); // ι8
  /*@ loop invariant 1 <= i <= n;
      loop invariant \forall integer k; 0 <= k < i ==> max >= a[k];
      loop invariant φ16: i < n ==> \valid_read(a + (0 .. n-1)); */
  for (int i = 1; i < n; i++) // ι9
    /*@ assert mem_access: φ13:\valid_read(a + i); */
    if (max < *(a + i)) // ι10
      max = *(a + i);
  return max;
}
void main(void){
  int arr[5] = {1, 2, 3, 4, 5};
  int sum = amax(arr,5); // ι11
}
```

(b) Preguss succeeds in verifying RTE-freeness

Fig. 7. AutoSpec fails to generate specifications such as $\langle \phi_{15}, \iota_{\text{pre}}^{\text{amax}} \rangle$ and $\langle \phi_{16}, \iota_9 \rangle$ as Preguss does, thereby failing to verify the RTE-freeness of `max.c`.

as adopted in [Wen et al. 2024, Sect. 4.1], AutoSpec produces loop invariants that ensure functional correctness – the output `max` indeed represents the array maximum. However, this guarantee presupposes the validity of RTE assertions $\langle \phi_{12}, \iota_8 \rangle$ and $\langle \phi_{13}, \iota_{10} \rangle$, which are ultimately ensured by the precondition $\langle \phi_{15}, \iota_{\text{pre}}^{\text{amax}} \rangle$ and loop invariant $\langle \phi_{16}, \iota_9 \rangle$ generated by Preguss.

The performance gap roots in two fundamental methodological distinctions:

(1) *Guidance Principle:* AutoSpec is not RTE assertion-guided; it decomposes the program and synthesizes specifications based solely on program structures – specifically, an extended call graph where loops and functions are treated as nodes. In contrast, Preguss uses guard RTE assertions to decompose the holistic verification task into fine-grained V-Units, generating necessary specifications for each. The 29.4% increase in verified programs (22 by AutoSpec$_{\text{RTE}}$ vs. 17 by AutoSpec) underscores the critical role of RTE-assertion guidance.

(2) *Specification-Synthesis Strategy:* The sequence and dependencies of specification generation differ markedly. For instance, in Fig. 7 (b), Preguss first generates precondition $\phi_{15}$ to validate guard assertion $\phi_{12}$, then produces loop invariant $\phi_{16}$ at $\iota_9$ for guard assertion $\phi_{13}$ (due to the design principle of the order $\sqsubseteq$ over V-Units; see Definition 16). Note that $\phi_{16}$ depends on $\phi_{15}$. AutoSpec, however, prioritizes loop invariants (at $\iota_9$ as the loop is the leaf node in its call graph) without necessary preconditions, leading to invariants such as $\phi_{16}$ being discarded during verification due to their semantic unsatisfiability.

**Execution Time and LLM Inference Cost.** Under a 10-minute limit per program, Preguss completes all the specification synthesis, whereas AutoSpec fails to terminate within this limit for 28/51 programs (though it often produces partial output, as seen in Fig. 7(a) for `max.c`). This results in AutoSpec's average execution time being over 4× longer than that of Preguss. Furthermore, Preguss consumes approximately 6× more LLM inference resources than the baselines. This increased cost reflects Preguss's strategy of guiding LLMs through a fine-grained synthesis process – such as employing different strategies upon specification types and incorporating proof obligations – which yields higher-quality specifications. This aligns with the *test-time scaling paradigm* [Jurayj et al. 2025], whose viability was recently justified by reasoning models such as OpenAI-o1 [OpenAI 2024]. Moreover, these high-quality specifications substantially reduce manual verification effort, far outweighing the marginal cost of LLM inference, as demonstrated in Section 6.3.

In summary, Preguss demonstrates a substantial advantage over AutoSpec in verifying RTE-freeness on the Frama-C-Problems benchmark suite. Although AutoSpec$_{\text{RTE}}$ improves upon AutoSpec by incorporating RTE assertion generation, both baselines lack a fine-grained synthesis mechanism that is essential for consistently producing high-quality specifications.

Table 2. Effectiveness of PREGUSS on large-scale, real-world programs. #RTE: the number of RTE assertions generated by the static analyzer; Total #V-Unit: the total number of V-Units constructed; Verified #V-Unit: the number of verified V-Units (i.e., V-Units with validated guard assertions) across three independent trials; Std. #Spec: the number of specifications required to verify RTE-freeness, which are either sourced from public repositories (X509-parser) or manually crafted by verification experts within a five person-day effort (Contiki and SAMCODE); Generated #Spec: the number of specifications synthesized by PREGUSS in each experiment; Modified #Spec: the number of specifications adjusted by experts to achieve full RTE-freeness based on PREGUSS's output, including three types of interventions (from left to right): (i) adding a new property, (ii) correcting an over-constrained precondition, and (iii) removing a property that becomes *unknown* after modifying other specifications; Avg. SR and Avg. HER are explained in Section 6.3.

| Benchmark | #RTE | Total #V-Unit | Verified #V-Unit | Avg. SR | Std. #Spec | Generated #Spec | Modified #Spec | Avg. HER | Time (m) | Cost ($) |
|---|---|---|---|---|---|---|---|---|---|---|
| Contiki | 163 | 182 | (181, 174, 162) | **94.7%** | 111 | (110, 90, 96) | (8, 15, 14) | **88.9%** | 65.58 | 1.77 |
| X509-parser | 142 | 174 | (151, 153, 148) | **86.6%** | 251 | (108, 116, 142) | (39, 41, 40) | **84.1%** | 151.80 | 3.84 |
| SAMCODE | 44† | 133 | (121, 116, 119) | **87.9%** | 225 | (186, 202, 206) | (34, 59, 38) | **80.6%** | 80.48 | 2.36 |
| Atomthreads | 239 | 328 | (252, 258, 258) | **78.1%** | –‡ | (416, 398, 544) | –‡ | –‡ | 807.58 | 18.52 |

† We omit floating-point alarms in SAMCODE, which, according to the developers, are not of primary interest.
‡ Std. #Spec, Modified #Spec, and Avg. HER for Atomthreads are unavailable as the manual verification (handcrafting std. specifications and refining PREGUSS-generated specifications) was not completed within the 5 person-day limit.

### 6.3 RQ2: Evaluation on Large-Scale Real-World Programs

First, we observe that for each of the four large-scale projects, both AUTOSPEC and $\text{AUTOSPEC}_{\text{RTE}}$ typically fail to terminate within the allotted 15-hour time limit, producing fewer than 15 specifications or aborting unexpectedly. Therefore, we exclude them from the comparison in **RQ2**.

Table 2 reports the experimental results of PREGUSS applied to large-scale, real-world projects. In addition to metrics described in the table, we compute the *average V-Unit success rate* (Avg. SR) and the *average reduction rate of human verification effort* (Avg. HER) as follows:

$$\text{Avg. SR} = \frac{\sum \text{Verified \#V-Unit}}{n \times \text{Total \#V-Unit}}, \quad \text{Avg. HER} = 1 - \frac{\sum \text{Modified \#Spec}}{n \times \text{Std. \#Spec}}$$

where $n = 3$ is the number of repeated experiments accounting for the inherent uncertainty of LLMs. Given the impracticality of fully automating RTE-freeness verification for large-scale projects, Avg. SR and Avg. HER serve as the primary quantitative measures of PREGUSS's effectiveness. The former reflects the proportion of guard assertions successfully resolved, directing expert attention to the remaining unverified hypotheses. The latter quantifies the reduction in manual effort achieved by PREGUSS compared to verifying the project from scratch.

We observe that PREGUSS *achieves highly automated RTE-freeness verification for the three real-world programs* – Contiki (544 LoC, 10 functions), X509-parser (1,199 LoC, 20 functions), and SAMCODE (1,280 LoC, 48 functions) – as reflected by Avg. SR and Avg. HER both exceeding 80% with average execution times under 3 hours and LLM inference costs below $4. These results underscore PREGUSS's effectiveness in synthesizing high-quality specifications and scalability to large projects. In particular, compared to traditional approaches that often require months or even years of expert effort for verifying systems with thousands of LoC [Djoudi et al. 2021; Ebalard et al. 2019], the computational and financial costs incurred by PREGUSS are marginal.

An interesting observation is that, for the X509-parser benchmark, the total number of generated and modified specifications is substantially lower than the number of standard specifications in all experimental trials. This discrepancy indicates that the ground-truth specifications handcrafted by the X509-parser authors contain many redundant properties – such as assertions/postconditions describing program states that are not directly relevant to verification, as detailed in [Wang et al.

2025a, Appendix C]. This finding further highlights the capability of PREGUSS to produce concise, high-quality specifications.

For the spacecraft control system SAMCODE, besides the 87.9% automatically verified V-Units, we *identified 6 genuine RTEs* (uninitialized left-value accesses) within the hypothesis set returned by PREGUSS. These errors, which stem from two logical bugs in the system implementation, have been subsequently confirmed by the developers. One such case is detailed in Section 6.6 (Case 2).

An analysis of the manual modifications made to PREGUSS's generated specifications also reveals certain limitations (elaborated in Section 7). For Contiki, most adjustments involve adding preconditions for C standard library functions such as `memcpy` and `memset` from `string.h`. Since these external functions lack implementation details in the source project, PREGUSS currently relies on handcrafted contracts for them. In X509-parser and SAMCODE, a small number of over-constrained preconditions are generated – one of which is examined in Section 6.6 (Case 3). After correcting such an over-constrained precondition $p$, properties which are valid under hypotheses including $p$ may become *unknown*, and thus should be removed.

For the Atomthreads benchmark, PREGUSS achieves an average V-Unit success rate of 78.1%, albeit with higher resource consumption (average time of 807.58 mins and cost of $18.52). Resolving the remaining guard assertions is particularly challenging due to the presence of linked-list management modules, which require advanced specifications such as inductive definitions [Blanchard 2020] for full verification (see Section 7). Constructing these specifications demands prohibitive expert effort – exceeding our five person-day budget – and falls outside PREGUSS's current design scope. Nevertheless, PREGUSS successfully generates high-quality specifications for RTE assertions unrelated to the linked-list module, contributing significantly to the overall success rate.

**Specification-Quality Metrics.** We evaluate the quality of PREGUSS-generated specifications through soundness guarantees and the Avg. HER metric, instead of the completeness metrics used in [Endres et al. 2024; Ma et al. 2025]. Soundness encompasses (i) *syntactic correctness* of all generated specifications and (ii) *semantic satisfiability* under the hypothesis set $\mathcal{H}$, while Avg. HER measures how well PREGUSS-generated specifications can substitute for standard oracles. The completeness metrics, which assess how well specifications capture program behaviors for functional correctness verification, are not suitable for PREGUSS. This is because PREGUSS aims to generate minimal contracts [Gerlach 2019] – the necessary specifications on which guard assertions depend, rather than pursuing the strongest specifications that precisely describe all program behaviors.

### 6.4 RQ3: Ablation Study

We conduct an ablation study to assess the individual contributions of key components within the PREGUSS framework, using the large-scale, real-world projects from **RQ2** (excluding Atomthreads due to its computationally prohibitive verification cost; the same applies to **RQ4**). Several core modules are deemed essential and thus excluded from ablation: (i) the *modules for V-Unit construction and prioritization* (Section 5.1), which are fundamental to determining the verification targets establishing the pipeline; (ii) the *fine-grained specification synthesis strategies* for host and callee functions (Sections 5.2.1 and 5.2.2), as allowing precondition generation for callees would compromise termination guarantees (Section 5.3); and (iii) the *semantic validity check* (Section 5.2.3), which prevents false negatives and is critical for soundness. Consequently, we focus on two adjustable mechanisms: *feedback-driven refinement* (FDR) and *syntax correction* (SC). We evaluate three ablated configurations against the full PREGUSS configuration from **RQ2**: PREGUSS without FDR (PREGUSS - FDR), PREGUSS without SC (PREGUSS - SC), and PREGUSS without both (PREGUSS - FDR - SC).

As illustrated in Fig. 8, *disabling either FDR or SC reduces the average V-Unit success rate, execution time, and LLM inference cost across most benchmarks*, with the combined ablation PREGUSS - FDR -

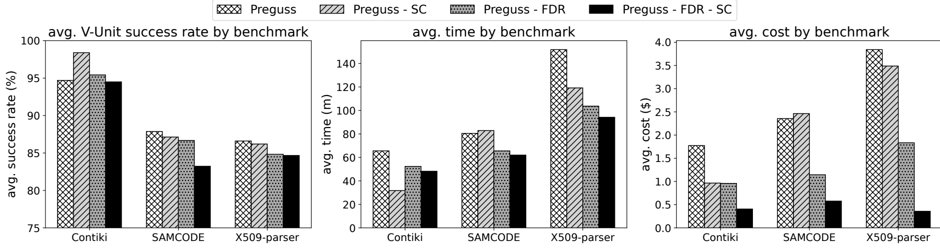



Fig. 8. Experimental results of the ablation study.

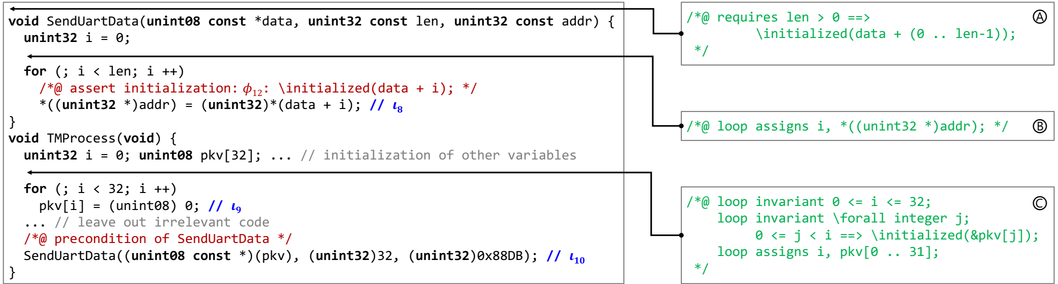



Fig. 9. Case 1: discharge false alarm $\langle\phi_{12}, \iota_8\rangle$ through specifications in Ⓐ–Ⓒ.

SC exhibiting the strongest negative effect. An exception occurs in the Contiki benchmark, where the V-Unit success rate unexpectedly increases when SC or FDR is disabled. This anomaly arises because Preguss - FDR - SC achieves a high baseline success rate (94.5%) on Contiki's relatively small codebase (544 LoC, 10 functions), where LLM uncertainty (temperature = 0.7) dominates over the contributions of FDR and SC. Overall, the high success rates (exceeding 83%) across all ablated configurations underscore the critical role of Preguss's essential modules – V-Unit management, fine-grained synthesis, and semantic checks – in achieving effective verification.

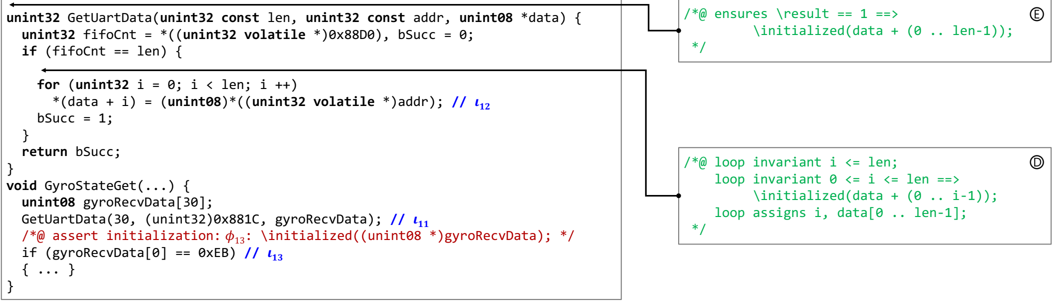
### 6.5 RQ4: Sensitivity to *iter* and LLMs

We observe that Preguss exhibits *no significant sensitivity of the V-Unit success rate to variations in iter, and achieves better performance with Claude 4 compared to GPT-5*; Details are in [Wang et al. 2025a, Appendix D].

### 6.6 Case Studies

We now discuss three representative cases found in SAMCODE, the spacecraft sun-seeking control system, to showcase how Preguss contributes to *discharging false alarms* (Fig. 9) and *identifying genuine RTEs* (Fig. 10). We also reveal a situation where Preguss may induce false alarms (Fig. 11). For clarity, we display minimal program contexts and retain only essential specifications.

**Case 1: Discharging False Alarms via Interprocedural Specifications.** The program depicted in Fig. 9 consists of two functions: (i) `SendUartData`, which iteratively transmits each element of array `data` (length `len`) to address `addr` at instruction $\iota_8$; (ii) `TMProcess`, which initializes each element of the `uint8` array `pkv` (length 32) at $\iota_9$ and invokes `SendUartData` at $\iota_{10}$ to transmit `pkv` to hardware address `0x88DB`. The static analyzer generates assertion $p_{12} = \langle\phi_{12}, \iota_8\rangle$, indicating a potential uninitialized left-value access UB at $\iota_8$. Although $p_{12}$ constitutes a false alarm and could be resolved through exhaustive analyzer configurations (e.g., full loop unrolling), such approaches often incur prohibitive computational costs. Leveraging $p_{12}$ and its verification feedback, Preguss

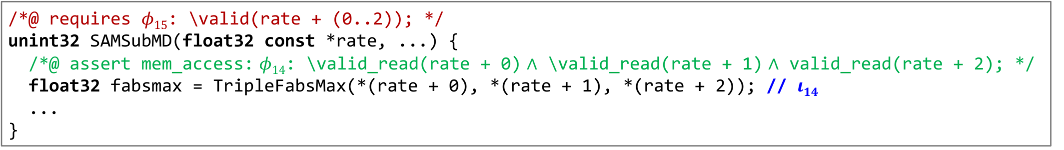


```
unint32 GetUartData(unint32 const len, unint32 const addr, unint08 *data) {
  unint32 fifoCnt = *((unint32 volatile *)0x88D0), bSucc = 0;
  if (fifoCnt == len) {
    for (unint32 i = 0; i < len; i ++)
      *(data + i) = (unint08)*((unint32 volatile *)addr); // ι12
    bSucc = 1;
  }
  return bSucc;
}
void GyroStateGet(...) {
  unint08 gyroRecvData[30];
  GetUartData(30, (unint32)0x881C, gyroRecvData); // ι11
  /*@ assert initialization: φ13: \initialized((unint08 *)gyroRecvData); */
  if (gyroRecvData[0] == 0xEB) // ι13
  { ... }
}
```

```
/*@ ensures \result == 1 ==>
        \initialized(data + (0 .. len-1));
 */
```
Ⓔ

```
/*@ loop invariant i <= len;
    loop invariant 0 <= i <= len ==>
        \initialized(data + (0 .. i-1));
    loop assigns i, data[0 .. len-1];
 */
```
Ⓓ

Fig. 10. Case 2: specifications in Ⓓ and Ⓔ aid in diagnosing the cause of invalid RTE assertion $\langle \phi_{13}, \iota_{13} \rangle$.



```
/*@ requires φ15: \valid(rate + (0..2)); */
uint32 SAMSubMD(float32 const *rate, ...) {
  /*@ assert mem_access: φ14: \valid_read(rate + 0) ∧ \valid_read(rate + 1) ∧ valid_read(rate + 2); */
  float32 fabsmax = TripleFabsMax(*(rate + 0), *(rate + 1), *(rate + 2)); // ι14
  ...
}
```

Fig. 11. Case 3: the over-constrained precondition $\langle \phi_{15}, \iota_{\text{pre}}^{\text{SAMSubMD}} \rangle$ induces a false alarm.

synthesizes precondition Ⓐ and loop invariant Ⓑ that collectively validate $p_{12}$. Recognizing that the precondition of `SendUartData` serves as a guard assertion at call site $\iota_{10}$, PREGUSS further generates specifications in `TMProcess` to discharge this precondition. This yields loop invariants Ⓒ which guarantee initialization of all pkv elements, and all relevant properties are verified as *true*.

**Case 2: Identifying Genuine RTEs via Interprocedural Specifications.** The program in Fig. 10 comprises two functions: `GetUartData` and `GyroStateGet`. The former first reads value from address `0x88D0` into variable `fifoCnt` and then branches as follows: If `fifoCnt == len` holds, it iteratively receives values from address `addr`, writes each into the array `data` (of length `len`) at instruction $\iota_{12}$, and returns 1; Otherwise, it returns 0. The function `GyroStateGet` invokes `GetUartData` at $\iota_{11}$ to initialize array `gyroRecvData` from hardware address `0x881C`. The static analyzer flags assertion $p_{13} = \langle \phi_{13}, \iota_{13} \rangle$, indicating a potential uninitialized left-value access UB.

To validate $p_{13}$, PREGUSS generates loop invariants Ⓓ and a postcondition Ⓔ for `GetUartData`, both verified as *true*. However, $p_{13}$ remains *unknown*, suggesting that `gyroRecvData[0]` may be uninitialized under certain conditions. Manual inspection of postcondition Ⓔ reveals that `gyroRecvData` is guaranteed initialized only if `GetUartData` returns 1. This indicates a latent issue: initialization may fail when the value at address `0x88D0` does not equal `len`, causing `GetUartData` to return 0. To address this, we introduce a conditional check at $\iota_{11}$ to handle the return value of `GetUartData`, ensuring subsequent instructions execute only when the return value is 1.

**Case 3: Inducing a False Alarm by an Over-Constrained Precondition.** The function `SAMSubMD` in Fig. 11 accepts a set of input parameters, including a constant array `rate`, and invokes function `TripleFabsMax` to compute the maximum absolute value of the first three elements of `rate`. Guided by the analyzer-generated assertion $\langle \phi_{14}, \iota_{14} \rangle$ – which requires the first three elements of `rate` to be readable to prevent an invalid memory access UB – PREGUSS synthesizes a precondition with predicate `\valid(rate + (0..2))`. This predicate incorrectly asserts that the first three elements of `rate` must be writable, contradicting the constant nature of `rate` and thus triggering a false alarm. The correct predicate should be `\valid_read(rate + (0..2))`. Although PREGUSS employs refined strategies to mitigate over-constrained preconditions, such issues may still arise occasionally, forming a primary component of the hypotheses $\mathcal{H}$ and necessitating expert intervention (e.g., manual weakening of preconditions) to complete the verification.

### 6.7 Threats to Validity

*Internal Validity.* The first primary threat to internal validity arises from the *inherent uncertainty* in LLM-based inference. To mitigate this, we repeat each experiment three times and report averaged results. The second primary threat is *data leakage*, i.e., are the evaluation code and the corresponding ground-truth ACSL specifications already part of Claude 4's training data? To isolate this potential influence, we build our dataset with three representative real-world projects: (i) Contiki, whose AES-CCM* module is verified but lacks publicly available ACSL specifications; (ii) Atomthreads, which to our knowledge has not been previously verified; and (iii) SAMCODE, which is both unverified and not publicly accessible. The consistent performance of Preguss on these benchmarks effectively rules out data-leakage concerns. The third threat concerns the potential misalignment between our evaluation metric – the V-Unit success rate in **RQ3** and **RQ4** – and the actual reduction in human verification effort (as assessed in **RQ2**), since the V-Units that remain unvalidated may require more sophisticated specifications than those successfully verified. Nonetheless, given that the primary objectives of **RQ3** and **RQ4** are to qualitatively assess the impact of individual components in Preguss on its overall performance, V-Unit success rate remains a robust, adequate metric.

*External Validity.* Major threats to external validity concern the soundness of the static analyzer and the completeness of the deductive verifier underpinning Preguss. Regarding the former, abstract interpretation-based static analyzers like Frama-C/Rte and Frama-C/Eva typically target specific categories of UBs and RTEs, which may limit the soundness of Preguss in guaranteeing RTE-freeness for those categories. For the latter, verifiers such as Frama-C/Wp rely critically on the underlying SMT solvers (e.g., Z3 [de Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022]). Limitations in these solvers – such as inadequate theorem modeling capabilities – can lead to false alarms where valid properties are incorrectly reported as *unknown* hypotheses.

## 7 Limitations and Future Directions

We pinpoint scenarios where Preguss is inadequate and provide potential solutions thereof.

Preguss currently faces challenges in handling advanced specifications, such as inductive definitions and axioms [Blanchard 2020], which are especially crucial for verifying programs involving complex data structures like linked lists and trees (recall Atomthreads in **RQ2**). One potential improvement is to incorporate few-shot examples of such specifications to prompt the LLM.

Despite its refined mechanisms, Preguss may occasionally produce over-constrained preconditions due to LLM hallucinations, as illustrated in Case 3 of Section 6.6. A promising direction is to explore how abstract interpretation-based static analyzers can be used to adjust LLM-generated preconditions by reasoning about over-approximated constraints at call sites.

As project scale increases, the program context of a single V-Unit – comprising the host function and its callees – may exceed the LLM's context window limit. Employing program slicing to extract only the callee functions (or the minimal set of statements) that are semantically relevant to the guard assertion could help reduce context size. Moreover, the current design of the program context omits global and external variables, which may result in false alarms related to these variables. Incorporating fine-grained data-flow analysis to retrieve such information could remedy this issue.

Finally, when a project uses external library APIs with unavailable implementations, handcrafted contracts are currently required. A promising extension would be to leverage LLMs to infer plausible contracts from function names and parameter types, thereby reducing the manual effort involved.

## 8 Related Work

Automated specification synthesis remains a critical challenge for scaling program verification [Ammons et al. 2002]. Traditional techniques are often tailored to specific types of specifications, such

as loop invariants [Colón et al. 2003; Jhala and McMillan 2006; Le et al. 2019; McMillan 2010], preconditions [Cousot et al. 2013, 2011], postconditions [Alshnakat et al. 2020; Molina et al. 2022; Wei et al. 2011], and assertions [Terragni et al. 2020]. Recently, the emergence of LLMs has led to a new paradigm (cf. [Zhang et al. 2025]), with several approaches demonstrating significant advantages over traditional methods by leveraging LLMs for specification synthesis [Cao et al. 2025; Ma et al. 2025; Mugnier et al. 2025; Pirzada et al. 2024; Wen et al. 2024; Wu et al. 2024b,a]. Below, we review closely related LLM-based approaches and clarify their distinctions from Preguss.

*LLM-Based Intraprocedural Specification Generation.* Several studies [Pirzada et al. 2024; Wu et al. 2024b,a] aim to generate loop invariants in support of bounded model checkers like ESBMC [Menezes et al. 2024], while Laurel [Mugnier et al. 2025] targets assertion synthesis for deductive verifiers such as Dafny. These methods primarily address the synthesis of *intraprocedural properties* within individual functions or lemmas, without considering call relations. In contrast, Preguss is designed to generate comprehensive interprocedural specifications – including contracts and invariants – for programs with complex call hierarchies. Given this fundamental difference in scope and objective, these intraprocedural approaches are orthogonal to Preguss and are thus not selected as baselines.

*LLM-Based Interprocedural Specification Generation.* Two notable works, AutoSpec [Wen et al. 2024] and SpecGen [Ma et al. 2025], address interprocedural specification synthesis. Our Preguss framework is inspired by AutoSpec, which employs static analysis to construct a comprehensive call graph (treating both functions and loops as nodes) and iteratively generates specifications for each node in a bottom-up fashion (cf. Section 5.1.2). A thorough discussion of the fundamental methodological distinctions between Preguss and AutoSpec is presented in Section 6.2. SpecGen adopts a mutation-based strategy to refine semantically invalid specifications by applying four mutation operators to predicate expressions and selecting the corrected variants. Moreover, SpecGen is implemented on OpenJML [Cok 2011] and targets JML specifications [Burdy et al. 2005] for Java programs, whereas Preguss is built on Frama-C/Wp to synthesize ACSL specifications for C programs. Given these fundamental differences in verification targets, implementation frameworks, and specification languages, we select only AutoSpec as a baseline for comparative evaluation.

In a nutshell, Preguss demonstrates superior scalability to large-scale programs – exceeding a thousand lines of code – due to two key distinctions from existing approaches: (i) it decomposes the monolithic RTE-freeness verification problem into a sequence of V-Units with sliced code contexts, thereby mitigating LLM context limitations; and (ii) it generates context-aware interprocedural specifications tailored to the host function and callee functions, rather than producing intraprocedural properties of specific types [Mugnier et al. 2025; Pirzada et al. 2024; Wu et al. 2024b,a] or synthesizing all types of specifications indiscriminately [Ma et al. 2025; Wen et al. 2024].

## 9 Conclusion

We have presented Preguss – a modular, fine-grained framework for inferring formal specifications synergizing between static analysis and deductive verification. We showcase that Preguss paves a compelling path towards the automated verification of large-scale programs: It enables nearly fully automated RTE-freeness verification of real-world programs with over a thousand LoC, while facilitating the identification of 6 confirmed RTEs in a practical spacecraft control system.

## 10 Data-Availability Statement

The artifact (including dataset) [Wang et al. 2026] is available at https://doi.org/10.5281/zenodo.18768810.

## Acknowledgments

This work has been partially funded by the NSFC under grant No. 62572427, by the ZJNSF Major Program (No. LD24F020013), by the Huawei Cooperation Project (No. TC20250422031), by the CCF-Huawei Populus Grove Fund (No. CCF-HuaweiSY202503), by the CASC Open Fund (No. LHCESET202502), by the NSFC under grant No. 62402038 and 62192730, and by the Fundamental Research Funds for the Central Universities of China (No. 226-2024-00140). The authors would like to thank Cheng Wen and Yutao Sun for their helpful discussions, Huangying Dong for specifying the SAMCODE spacecraft software, Yazhou Tang for testing the artifact, and the anonymous reviewers for their constructive feedback.

## References

2000. *The Explosion of the Ariane 5*. Retrieved October, 2025 from https://www-users.cse.umn.edu/~arnold/disasters/ariane.html

2020. *A repository dedicated for problems related to verification of programs using the tool Frama-C*. Retrieved October, 2025 from https://github.com/manavpatnaik/frama-c-problems

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

Anoud Alshnakat, Dilian Gurov, Christian Lidström, and Philipp Rümmer. 2020. Constraint-Based Contract Inference for Deductive Verification. In *20 Years of KeY*. Lecture Notes in Computer Science, Vol. 12345. Springer, 149–176. doi:10.1007/978-3-030-64354-6_6

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

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL*. ACM, 4–16. doi:10.1145/503272.503275

Anthropic. 2025. *System Card: Claude Opus 4 & Claude Sonnet 4*. Retrieved October, 2025 from https://www-cdn.anthropic.com/6be99a52cb68eb70eb9572b4cafad13df32ed995.pdf

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

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

Allan Blanchard. 2020. *Introduction to C program proof with Frama-C and its WP plugin*. https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*. https://dl.acm.org/doi/abs/10.5555/3495724.3495883

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

Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.* 7, 3 (2005), 212–232. doi:10.1007/s10009-004-0167-4

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

David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 6617)*. Springer, 472–479. doi:10.1007/978-3-642-20398-5_35

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV (Lecture Notes in Computer Science, Vol. 2725)*. Springer, 420–432. doi:10.1007/978-3-540-45069-6_39

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

Partick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179. doi:10.1007/3-540-45937-5_13

Partick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *(VMCAI)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). LNCS 7737, Springer, Heidelberg, Berlin, Heidelberg, 128–148. doi:10.1007/978-3-642-35873-9_10

Partick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *(VMCAI)*, D. Schmidt R. Jhala (Ed.). LNCS 6538, Springer, Heidelberg, Austin, Texas, 150–168. doi:10.1007/978-3-642-18275-4_12

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

Adel Djoudi, Martin Hána, and Nikolai Kosmatov. 2021. Formal Verification of a JavaCard Virtual Machine with Frama-C. In *FM (Lecture Notes in Computer Science, Vol. 13047)*. Springer, 427–444. doi:10.1007/978-3-030-90870-6_23

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)*. https://www.sstic.org/media/SSTIC2019/SSTIC-actes/journey-to-a-rte-free-x509-parser/SSTIC2019-Article-journey-to-a-rte-free-x509-parser-ebalard_mouy_benadjila.pdf

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE (2024), 1889–1912. doi:10.1145/3527325

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. https://www.usenix.org/system/files/usenixsecurity24-fang.pdf

Robert W. Floyd. 1993. Assigning Meanings to Programs. In *Program Verification*. Studies in Cognitive Systems, Vol. 14. Springer Netherlands, 65–81. doi:10.1007/978-94-011-1793-7_4

Yu-Fu Fu, Meng Xu, and Taesoo Kim. 2025. Agentic Specification Generator for Move Programs. In *ASE*. IEEE, 1286–1298. doi:10.1109/ASE63991.2025.00110

Jens Gerlach. 2019. *Minimal contract Hoare-style verification versus abstract interpretation.* Technical Report. https://vessedia.technikon.com/downloads/VESSEDIA-D5.7-Minimal-contract-Hoare-style-verification-vs-abstract-interpretation-PU-M36.pdf

Philippe Herrmann and Julien Signoles. 2025. *RTE - Runtime Error Annotation Generation.* https://www.frama-c.com/download/frama-c-rte-manual.pdf

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

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

ISO/IEC JTC 1/SC 22. 2024. *ISO/IEC 9899:2024 Information technology — Programming languages — C.* ISO. https://www.iso.org/standard/82075.html

Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *TACAS (Lecture Notes in Computer Science, Vol. 3920)*. Springer, 459–473. doi:10.1007/11691372_33

William Jurayj, Jeffrey Cheng, and Benjamin Van Durme. 2025. Is That Your Final Answer? Test-Time Scaling Improves Selective Question Answering. In *ACL (2)*. Association for Computational Linguistics, 636–644. doi:10.18653/v1/2025.acl-short.50

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

Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. 2024. *Guide to Software Verification with Frama-C.* Springer Cham. doi:10.1007/978-3-031-55608-1

Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Springer. doi:10.1007/978-3-662-50497-0

Kelvin Lawson. 2015. *Atomthreads: Open Source RTOS.* Retrieved October, 2025 from https://github.com/kelvinlawson/atomthreads

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *PLDI*. ACM, 788–801. doi:10.1145/3314221.3314634

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

Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *USENIX Security Symposium*. USENIX Association, 489–507. https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *ICSE*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

John McCarthy. 1961. A basis for a mathematical theory of computation, preliminary report. In *IRE-AIEE-ACM Computer Conference (Western)*. ACM, 225–238. doi:10.1145/1460690.1460715

Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *CAV (Lecture Notes in Computer Science, Vol. 6174)*. Springer, 104–118. doi:10.1007/978-3-642-14295-6_10

Rafael Sá Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro. 2024. ESBMC v7.4: Harnessing the Power of Intervals - (Competition Contribution). In *TACAS (3) (Lecture Notes in Computer Science, Vol. 14572)*. Springer, 376–380. doi:10.1007/978-3-031-57256-2_24

Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *ICSE*. ACM, 1008–1020. doi:10.1145/3510003.3510120

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1519–1545. doi:10.1145/3720499

OpenAI. 2024. *Learning to reason with LLMs.* Retrieved October, 2025 from https://openai.com/index/learning-to-reason-with-llms/

OpenAI. 2025. *GPT-5 System Card.* Retrieved October, 2025 from https://cdn.openai.com/gpt-5-system-card.pdf

Alain Ourghanlian. 2015. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology* 47, 2 (2015), 212–218. doi:10.1016/j.net.2014.12.009 Special Issue on ISOFIC/ISSNP2014.

Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, Inria Lille, and Shahid Raza. 2018. Towards Formal Verification of Contiki: Analysis of the AES-CCM* Modules with Frama-C. In *EWSN*. Junction Publishing, Canada/ ACM, 264–269. https://dl.acm.org/doi/10.5555/3234847.3234910

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

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

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

Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *ESEC/SIGSOFT FSE*. ACM, 1178–1189. doi:10.1145/3368089.3409758

Chengpeng Wang. 2025. *Advances in AI-Powered Code Security: Next-Level Bug Detection.* Technical Report. https://chengpeng-wang.github.io/slides/talk_GitHub.pdf

Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Haokun Li, Mingqi Yang, Xiao Yi, Shengchao Qin, Yixing Luo, Xiaofeng Li, Bin Gu, Liqiang Lu, and Jianwei Yin. 2025a. A Tale of 1001 LoC: Potential Runtime Error-Guided Specification Synthesis for Verifying Large-Scale Programs. *CoRR* abs/2512.24594 (2025). https://doi.org/10.48550/arXiv.2512.24594

Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Mingqi Yang, Haokun Li, Xiao Yi, Shengchao Qin, and Jianwei Yin. 2025b. Preguss: It Analyzes, It Specifies, It Verifies. In *LMPL*. ACM, 118–123. doi:10.1145/3759425.3763394

Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Mingqi Yang, Haokun Li, Xiao Yi, Shengchao Qin, and Jianwei Yin. 2026. *OOPSLA26 Artifact – A Tale of 1001 LoC: Potential Runtime Error-Guided Specification Synthesis for Verifying Large-Scale Programs.* doi:10.5281/zenodo.18768810

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:10.1145/3691620.3695487

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*. https://dl.acm.org/doi/10.5555/3600270.3602070

Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *ICSE*. ACM, 191–200. doi:10.1145/1985793.1985820

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

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

Haoze Wu, Clark W. Barrett, and Nina Narodytska. 2024a. Lemur: Integrating Large Language Models in Automated Program Verification. In *ICLR*. https://iclr.cc/virtual/2024/poster/18684

Yedi Zhang, Yufan Cai, Xinyue Zuo, Xiaokun Luan, Kailong Wang, Zhe Hou, Yifan Zhang, Zhiyuan Wei, Meng Sun, Jun Sun, Jing Sun, and Jin Song Dong. 2025. Position: Trustworthy AI Agents Require the Integration of Large Language Models and Formal Methods. In *ICML*. https://icml.cc/virtual/2025/poster/40101

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö. Arik. 2024. Chain of Agents: Large Language Models Collaborating on Long-Context Tasks. In *NeurIPS*. doi:10.52202/079017-4202

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 2025-10-10; accepted 2026-02-17