

Neuro-symbolic Requirements Elicitation: Utilizing Formal Verification Counterexamples as Contextual Prompts

Okba Tibermacine

National School of Artificial Intelligence (ENSIA)

Sidi Abdallah, Algeria

okba.tibermacine@ensia.edu.dz

Chouki Tibermacine

IRISA, University of Southern Brittany

Vannes, France

chouki.tibermacine@univ-ubs.fr

Abstract

Incomplete and ambiguous requirements remain a common source of defects in safety-critical software systems. Formal methods provide rigorous correctness guarantees but rely on precise specifications, while Large Language Models (LLMs) enable flexible natural-language processing but lack logical consistency guarantees. Bridging the gap between informal stakeholder intent and formally verifiable specifications remains a key challenge in automated software engineering. This paper presents C-GRiD (Counterexample-Guided Requirements Discovery), a neuro-symbolic framework that integrates model checking into the requirements elicitation process. C-GRiD uses an LLM to synthesize an initial system model and a set of candidate requirements properties, which are then analyzed by a symbolic verifier. Counterexamples generated by the verifier are converted into structured natural-language queries that elicit missing constraints from stakeholders. This enables an iterative refinement loop that progressively reduces specification incompleteness while maintaining logical grounding of generated artifacts. We evaluate C-GRiD on 10 benchmark specifications using a controlled constraint-removal protocol to measure requirement recovery. Under a perfect oracle assumption, the framework achieves a mean constraint recovery recall of 79.2%. To assess robustness under realistic interaction settings, we introduce a simulated noisy stakeholder with a 25% error rate. In this setting, C-GRiD maintains a recall of 70.8%. Results show consistent improvements over one-shot and template-based baselines, demonstrating that counterexample-guided refinement can support the construction of more complete and logically grounded software specifications.

CCS Concepts

• **Software and its engineering** → **Requirements analysis**; **Formal software verification**; • **Computing methodologies** → *Natural language processing*.

Keywords

Requirements Engineering, Formal Verification, Neuro-Symbolic AI, Large Language Models, Specification Discovery, Model Checking

ACM Reference Format:

Okba Tibermacine and Chouki Tibermacine. 2026. Neuro-symbolic Requirements Elicitation: Utilizing Formal Verification Counterexamples as Contextual Prompts. In *Proceedings of the 41st IEEE/ACM International Conference*



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2882-2/2026/10

<https://doi.org/10.1145/3832783.3834379>

on *Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3832783.3834379>

1 Introduction

An important source of failures in complex software systems rarely stems from coding bugs, but from incomplete, ambiguous, or contradictory requirements [17, 22, 26]. Domain experts often possess a latent mental model with unstated assumptions and unexplored edge cases. A key challenge of requirements engineering is to make this latent model explicit and logically sound.

Two paradigms address this challenge, each with a key limitation. Formal Methods offer mathematical rigor via languages like TLA+ and Alloy, but their steep learning curve creates a *Blank Page Problem*: domain experts are rarely equipped to write or read formal specifications. Large Language Models offer natural language fluency and can translate informal descriptions into specifications, but as statistical models they lack logical grounding and operate without formal correctness guarantees—a *Logical Grounding Problem* [5]. LLMs are prone to hallucinating incorrect constraints and missing subtle corner cases (e.g., race conditions), posing severe reliability risks in safety-critical systems.

In this paper, we use formal verification failures not to judge a final specification, but to drive interactive discovery of a complete one [16, 22]. Our core insight is that a counterexample—a finite execution trace witnessing property violation in a candidate specification—provides a deterministic, semantic prompt signaling a missing or incorrect requirement.

We embody this insight in *Counterexample-Guided Requirements Discovery (C-GRiD)*, a neuro-symbolic feedback loop targeting early development phases lacking validated models or complete requirements. C-GRiD uses an LLM to generate an initial formal model and candidate properties from incomplete descriptions. A symbolic model checker explores this model for property violations. Each counterexample is translated by the LLM into a targeted natural-language question for the stakeholder (e.g., “Should concurrent transfers be allowed if they might cause an overdraft?”). The stakeholder’s answer provides a new constraint, repeating the loop to systematically close requirement gaps.

This paper makes the following contributions:

- (1) We introduce a software engineering approach that integrates formal verification into the requirements discovery process, making counterexample-driven refinement accessible to non-experts through natural language interaction.
- (2) We define a practical neuro-symbolic architecture that provides a safety net for LLM-based specification generation,

using a symbolic verifier to ground the LLM’s output in logical truth and mitigate hallucinations.

- (3) We present an empirical validation using a controlled constraint-removal methodology, demonstrating that our approach recovers substantially more missing constraints than automated baselines, with demonstrated robustness to imperfect stakeholder feedback.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 introduces the C-GRiD framework. Section 4 presents the experimental evaluation. Section 5 discusses findings and threats to validity, and Section 6 concludes.

2 Related Work

C-GRiD lies at the intersection of AI-assisted requirements engineering, requirements formalisation, formal verification, and the neuro-symbolic use of LLMs.

2.1 AI- and ML-enabled Requirements Elicitation

Artificial intelligence has been applied to requirements elicitation primarily to automate processing of natural language artifacts rather than integrating formal verification. Systematic reviews identify tasks such as extracting candidate requirements, classifying and prioritizing them, detecting conflicts, and generating analyst questions, predominantly using classical ML and transformers [6, 7, 21, 31]. Mapping studies document limitations including language bias, lack of standard benchmarks, and deployment friction [18, 32, 33], while advanced conceptual models remain focused on informal artifacts [8]. Recent works evaluate LLMs as elicitation assistants for user stories, interviews, and multi-agent RE workflows [1, 30], or fine-tune models on standards like ISO 29148 [15].

Closest in spirit are interactive NLP/LLM requirements assistants (e.g., Arora et al. [2]), which resolve ambiguity via dialogue and text analysis. However, these operate entirely at the natural language level without verification feedback. C-GRiD complements these by coupling LLM elicitation with model checking, interpreting counterexamples as formal indicators of missing constraints.

2.2 NLP and LLMs for Requirements Formalisation

A parallel line targets *requirement formalisation*: transforming natural language requirements into formal specifications. Most classical approaches apply heuristic NLP to structured inputs [19, 32]. Subsequent work translates structured language into temporal logics, including ARSENAL [11], Giannakopoulou et al. [12], and targeted packages like NL2LTL [9], TR2MTL [23], NL2STL [24], and NL2CTL [35]. Other frameworks integrate NLP, ontologies, and LLMs to generate verifiable specifications [3], while surveys document LLM assistance across languages like Dafny, ACSL, Coq, Lean, and TLA⁺ [4].

These works predominantly *translate* existing requirements text. In contrast, C-GRiD focuses on *discovering* unarticulated requirements using counterexamples as contextualized evidence of missing assumptions.

Table 1: Positioning of C-GRiD within Specification Engineering Paradigms

Paradigm	Primary Target	Inputs / Outputs	Representative Works
Translation	Properties (Φ)	Natural language → LTL/CTL formulas	ARSENAL [11], NL2LTL [9], NL2CTL [35]
Validation	Formulas (Φ)	Interactive traces → user validation	LTLTalk [10], SynthTL [27], ARTEMIS [28]
Joint elicitation	Model (δ) and properties (Φ)	Incomplete specs → verified transition systems	C-GRiD (this work)

2.3 Formal Verification, Specification Mining, and Interactive Assistants

Formal verification depends critically on precise specifications, motivating research on mining them from behaviour. Surveys of LTL specification mining review constraint-solving, search-based and neural techniques for learning properties from traces [29]. Neuro-symbolic specification synthesis has also been explored: “seeking specifications” uses an LLM to generate ACSL contracts for C code and studies how symbolic analyses can augment prompts [14], while broader studies position LLMs as assistants for code analysis and bug detection with formal methods as a safety net [25]. Closest to our setting among these, SpecVerify pairs an LLM with the ES-BMC verifier to generate assertions from textual requirements of cyber-physical systems and check them against Simulink/Stateflow models [34]; it uses counterexamples to improve conformance between an implementation and a predefined specification, whereas C-GRiD exploits them to derive the specification itself.

A distinct line of work uses model checkers to generate traces that elicit human-in-the-loop judgement about specification intent, and is the closest antecedent of our approach. LTLTalk [10] synthesises temporal specifications from natural language together with user feedback on system traces. SynthTL [27] and ARTEMIS [28] combine LLMs and model checkers to generate and validate temporal formulas by presenting low-complexity distinguishing traces to stakeholders. Three paradigms are thus separable by what they refine. *Translation* maps natural language to LTL/CTL formulas (ARSENAL [11], NL2LTL [9], NL2CTL [35]). *Validation* presents interactive traces so that a user can confirm or reject a candidate formula, under a given behavioural model (LTLTalk [10], SynthTL [27], ARTEMIS [28]). C-GRiD performs *joint elicitation*: the loop refines both the requirements properties and the underlying transition model, starting from an incomplete specification in which either may be wrong, and it frames the counterexample as an elicitation prompt rather than as a discriminating query over a fixed hypothesis space.

Table 1 summarises the three paradigms.

2.4 Summary, Gap, and Distinction from CEGIS

AI-based approaches have advanced automation in elicitation and analysis mainly at the level of informal text [6, 7, 15, 31, 32], and

Table 2: Conceptual Distinction Between CEGIS and C-GRiD

Aspect	CEGIS	C-GRiD (Our Work)
Goal	Synthesize a correct program/model that meets a fixed specification.	Discover missing or incorrect requirements to complete the specification itself.
Oracle	A formal teacher or solver that provides a counterexample.	A (potentially non-expert) human stakeholder or domain-aware LLM.
Feedback	A concrete counterexample trace that falsifies the candidate.	Natural language clarification of intent in response to a scenario.
Refinement target	The candidate program or model within a formal hypothesis space.	The set of requirements Φ and the system model δ .

formalisation work maps given requirements into formal representations [3, 4, 19]. Neuro-symbolic frameworks integrating LLMs with formal analysis largely assume a fixed requirement baseline and use counterexamples to repair artefacts rather than to expand the requirement set [14, 34]. Requirements engineering is also increasingly recognised as essential to assuring ML-based systems, with neuro-symbolic methods proposed as the enforcement mechanism [8, 13]; C-GRiD is aligned with that perspective. C-GRiD addresses the identified gap by reframing counterexamples as first-class, semantically rich prompts in a human-in-the-loop elicitation loop, turning formal verification from a terminal correctness check into an interactive discovery engine.

C-GRiD’s iterative loop is structurally reminiscent of Counterexample Guided Inductive Synthesis (CEGIS) [20] but differs conceptually. CEGIS refines a formal hypothesis within a predefined synthesis space so as to satisfy a *fixed* specification; its oracle is a formal teacher or solver, its feedback a trace falsifying the candidate, and its refinement target the candidate program. C-GRiD instead refines the *problem statement itself*: its oracle is a (potentially non-expert) human stakeholder or domain-aware LLM, its feedback a natural language clarification of intent, and its refinement target the requirement set together with the system model. The counterexample is thereby elevated from hypothesis falsifier to requirements elicitation artifact. Table 2 summarises it.

3 The C-GRiD Framework

This section presents the theoretical foundations, architectural components and algorithmic implementation of C-GRiD. We formulate requirements elicitation as a cooperative, iterative search through the space of possible system specifications, guided by both symbolic verification and human-in-the-loop feedback.

3.1 Problem Formulation and Formal Foundations

To ensure domain independence and enable rigorous analysis, we model the target system as a **Labeled Transition System (LTS)**.

DEFINITION 1 (SYSTEM SPECIFICATION MODEL). *A formal system specification S is defined as a tuple $\mathcal{M} = \langle \Sigma, \Sigma_0, \mathcal{A}, \delta, \Phi \rangle$, where:*

- Σ is a finite set of system states. A state $s \in \Sigma$ is a valuation of all system variables $V = \{v_1, \dots, v_n\}$.
- $\Sigma_0 \subseteq \Sigma$ is the set of initial states.
- $\mathcal{A}$ is a finite set of actions that trigger state transitions.
- $\delta : \Sigma \times \mathcal{A} \rightarrow \Sigma$ is the transition relation, defining the behavioural model of the system.
- Φ is a set of formal properties, typically expressed in Linear Temporal Logic (LTL), that the system must satisfy. These represent the requirements properties, such as safety invariants ($\Box P$) or liveness properties ($\Diamond Q$).

We use a *system model* for $\langle \delta, \Sigma_0 \rangle$, *requirements properties* for Φ , and a *specification* for the tuple $\mathcal{M}$. Initially, both are imperfect: $\Phi_{\text{initial}} \subset \Phi_{\text{ideal}}$ and $\delta_{\text{initial}} \neq \delta_{\text{ideal}}$, where ideal elements represent latent stakeholder intent. C-GRiD restricts its scope to behavioral temporal properties over discrete LTSs; non-behavioral qualities like performance and usability are out of scope. C-GRiD discovers missing properties $\Phi_{\text{ideal}} \setminus \Phi_{\text{initial}}$ by treating property violations as probes into the stakeholder’s mental model.

3.2 Architectural Overview

The framework is a closed-loop system of four interconnected modules, illustrated in Figure 1. The core components are the Neural Synthesizer, the Symbolic Verifier, and the feedback loop connecting them through algorithmic analysis.

3.3 The Neural Synthesizer ($\mathcal{N}$)

This component maps informal documentation $\mathcal{D}$ and dialogue history $\mathcal{H}$ to specification $\mathcal{M}$: $\mathcal{N} : (\mathcal{D} \times \mathcal{H}) \rightarrow \mathcal{M}$. We use an LLM with *constrained decoding* to output a strict Intermediate Representation (IR) schema compiled into TLA+, mitigating syntax errors and hallucinations.

Constrained decoding applies to initial synthesis V_0 only; subsequent refinements operate directly on TLA+ text (Section 3.8) to preserve code structure. This asymmetry allows refinements to introduce errors avoided during initial synthesis, motivating the regression guard (Section 3.7).

3.4 The Symbolic Verifier ($\mathcal{V}$)

This component is the logical core of the framework. It accepts a candidate specification $\mathcal{M}_i$ and performs bounded model checking up to a predefined path depth k :

$$\mathcal{V}(\mathcal{M}_i, \Phi) = \begin{cases} \top & \text{if } \forall \pi \in \text{Paths}_k(\mathcal{M}_i), \pi \models \Phi \\ \tau & \text{if } \exists \pi \in \text{Paths}_k(\mathcal{M}_i), \pi \not\models \Phi \end{cases} \quad (1)$$

where $\tau = \langle s_0, a_1, s_1, \dots, a_k, s_k \rangle$ ($s_i \in \Sigma, a_i \in \mathcal{A}$) is a concrete counterexample trace that interleaves states with the actions relating them. The verifier’s soundness is critical: it acts as the logical baseline that grounds the suggestions of the neural component.

Why bounding the search? The elicitation loop needs rapid feedback on shallow violations in order to construct clarification prompts, so we bound exploration depth rather than verify to exhaustion. We use “bounded model checking” in the sense of *depth-bounded state exploration*: the backend is the explicit-state checker TLC with its search depth capped at k (Section 4.2.3), not SAT-based unrolling. The consequence is a completeness limit we make explicit:

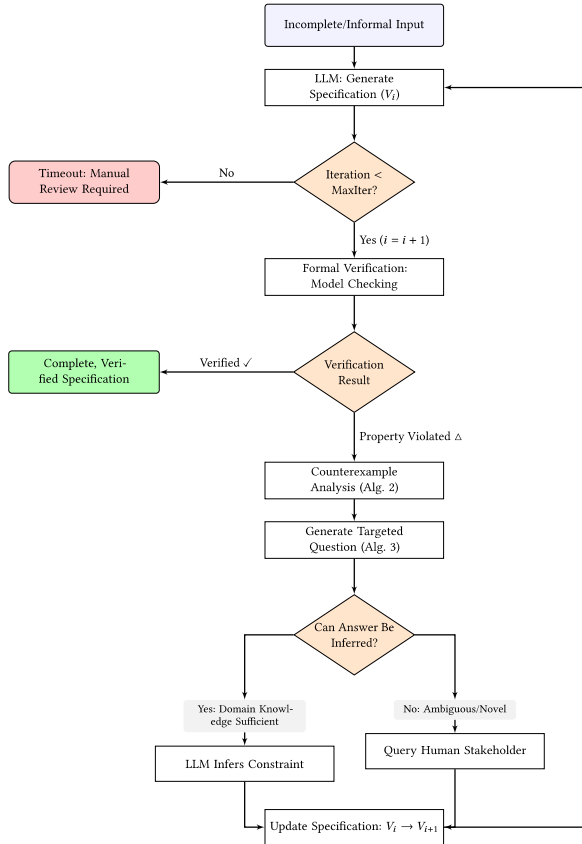


Figure 1: The detailed execution flow of the C-GRiD framework. The system iterates through verification loops, using counterexamples to trigger either automated domain inference (Mode 1) or human clarification (Mode 2), updating the specification (V_i) until convergence.

violations requiring traces longer than k are never surfaced, so a constraint whose only witness is a deep trace cannot be elicited. Unbounded verification would remove this limit at the cost of per-iteration latency that an interactive loop cannot absorb; evaluating that trade-off is a future work.

3.5 Algorithmic Implementation

3.5.1 The Main Control Loop (Algorithm 1). Algorithm 1 is the central coordinator of the iterative refinement process.

Initialization (Lines 1–3). The Neural Synthesizer generates an initial candidate specification V_0 from the informal requirements R ; V_0 is a joint hypothesis over the transition relation and the properties, either of which may be wrong. A history set is maintained to detect cycles.

The Verification Loop (Lines 4–27). The loop continues until a verified specification is found or a termination condition is met. In each iteration (Line 5) the symbolic verifier is invoked on V_i ; successful verification (Lines 7–8) is the primary termination condition.

Counterexample-Driven Refinement (Lines 10–12). If verification fails, a counterexample τ is produced and passed to the purely

Algorithm 1 Counterexample-Guided Requirements Discovery (C-GRiD)

Require: Informal requirements R , Property set Φ , Max iterations $N_{\max}$
Ensure: Verified specification V^* or status $\in \{\text{TIMEOUT}, \text{OSCILLATION}\}$

```

1:  $V_0 \leftarrow \text{LLMGENERATE}(R)$ 
2:  $i \leftarrow 0$ 
3:  $\text{History} \leftarrow \emptyset$ 
4: while  $i < N_{\max}$  do
5:    $(\text{status}, \tau) \leftarrow \text{MODELCHECK}(V_i, \Phi)$ 
6:   if  $\text{status} = \text{VERIFIED}$  then
7:     return  $V_i$  {Convergence to a correct specification}
8:   end if
9:   {Counterexample analysis phase}
10:   $\langle \phi_{\text{viol}}, s_{\text{viol}}, \sigma \rangle, \text{pattern} \leftarrow \text{ANALYZETRACE}(\tau, \Phi)$ 
11:  {Question generation phase}
12:   $q \leftarrow \text{GENERATEQUESTION}(\text{pattern}, \tau, R)$ 
13:  {Answer acquisition phase}
14:  if  $\text{CANINFERFROMDOMAIN}(q, \mathcal{K}_{\text{domain}})$  then
15:     $\text{ans} \leftarrow \text{LLMINFER}(q, \mathcal{K}_{\text{domain}})$  {Mode 1: Automated}
16:  else
17:     $\text{ans} \leftarrow \text{QUERYHUMAN}(q)$  {Mode 2: Interactive}
18:  end if
19:  {Specification refinement phase}
20:   $V_{i+1} \leftarrow \text{REFINESPECIFICATION}(V_i, \text{ans}, \tau, \phi_{\text{viol}})$ 
21:  {Convergence check}
22:  if  $V_{i+1} \in \text{History}$  then
23:    return  $\text{OSCILLATION}$  {Cycle detected}
24:  end if
25:   $\text{History} \leftarrow \text{History} \cup \{V_{i+1}\}$ 
26:   $i \leftarrow i + 1$ 
27: end while
28: return  $\text{TIMEOUT}$ 

```

symbolic ANALYZETRACE function (Algorithm 2), which dissects the failure into a structured format including a high-level error pattern (Line 10). GENERATEQUESTION (Algorithm 3) then formulates a targeted natural-language query (Line 12).

Feedback Acquisition and Update (Lines 14–20). The question is posed either to an automated domain reasoner or to a human stakeholder. The framework supports a *fully automated mode* (Mode 1), in which an LLM infers the constraint from a knowledge base $\mathcal{K}_{\text{domain}}$ (Line 15), and a *semi-automated mode* (Mode 2), which queries a human on novel or ambiguous issues (Line 17). $\text{REFINESPECIFICATION}$ (Line 20) then incorporates the answer. The answer may add a property to Φ , relax an existing property in Φ , strengthen a guard in the transition relation δ , or combine these; which of them applies is determined by the polarity of the stakeholder’s answer, as detailed in Section 3.6.

Convergence and Termination (Lines 22–26). An oscillation check terminates the process if V_{i+1} has been seen before; otherwise the loop continues until $N_{\max}$ iterations are exhausted.

3.5.2 Domain Knowledge Inference. Line 14 chooses between the two modes by testing whether q can be answered from a knowledge base $\mathcal{K}_{\text{domain}}$ of ontological facts (domain rules encoded as natural-language statements, e.g. “banking: transfers require positive balance”) and syntactic patterns with canonical answers (e.g. “Should [variable] be non-negative?”). It returns true above a match confidence of 0.85, otherwise deferring to the human. Approximately 40% of questions were answerable automatically.

3.5.3 Counterexample Trace Analysis (Algorithm 2). Algorithm 2 is a deterministic, symbolic procedure that converts a raw counterexample trace into structured, actionable information, abstracting away the complexity of the model checker’s output. It takes the trace τ and property set Φ and returns the violated property ϕ_{viol} , the point at which the violation is witnessed, the causal action sequence σ , and an error-pattern classification.

Phase 1 (Lines 2–8) locates the violation. Because the properties are temporal, satisfaction is defined over traces and prefixes rather than over individual states; there is in general no single “state where the violation occurred”. Our analysis targets safety properties, for which every violation has a finite bad prefix, and pins the violation to the shortest prefix $\tau[0..i]$ at which the invariant is first falsified. Liveness violations, which are witnessed by lasso-shaped traces, are reported to the stakeholder as whole traces and are not localised further. *Phase 2 (Line 10)* backtraces from the violation point, pruning irrelevant actions and returning the minimal sequence σ necessary to reach it. *Phase 3 (Lines 13–21)* classifies the failure: numeric bound violations indicate a MISSINGGUARD, two or more concurrent accesses to a shared resource in σ a RACECONDITION, circular resource waits a DEADLOCK, and remaining cases fall back to LOGICERROR.

3.5.4 Trace-to-Scenario Question Generation (Algorithm 3). Algorithm 3 is the neuro-symbolic component that bridges the cognitive gap between formal methods and stakeholder intent. It takes the error pattern, the trace, and a domain ontology mapping formal variable names to business concepts, and outputs a formatted natural language question with answer options. It first extracts domain entities (e.g., ‘Account-A’) and concrete values (e.g., ‘balance = -50’) to construct a short narrative of what went wrong, as illustrated in Table 3. To keep questions precise and resistant to hallucination, each error pattern is bound to a question template—a RACECONDITION always yields a question about atomicity or serialization—so that the interaction is predictable and focuses the stakeholder on a single decision. For unknown or complex logic errors the framework falls back to an open-ended LLM prompt.

3.6 Symmetric Resolution of Translation Errors and Missing Requirements

A counterexample on V_0 can reflect either a genuinely missing constraint or an over-constrained property introduced during translation. C-GRiD does not attempt to distinguish the two cases automatically; it resolves both through the same stakeholder-facing interface, and the polarity of the answer determines the repair. An answer of the form “this behaviour is allowed” identifies a translation artifact and triggers relaxation of Φ ; an answer of the form “this behaviour is disallowed” identifies a missing constraint and

Algorithm 2 Counterexample Trace Analysis

Require: Counterexample trace $\tau = \langle s_0, a_1, \dots, a_n, s_n \rangle$, Property set Φ

Ensure: Violation $\langle \phi_{\text{viol}}, s_{\text{viol}}, \sigma \rangle$, Error pattern *pattern*

```

1: {Phase 1: Identify violation point}
2: for  $i = 0$  to  $n$  do
3:   for each  $\phi \in \Phi$  do
4:     if  $\tau[0..i] \not\models \phi$  then
5:        $\phi_{\text{viol}} \leftarrow \phi$ ;  $s_{\text{viol}} \leftarrow s_i$ ; goto ViolationFound
6:     end if
7:   end for
8: end for
9: {Phase 2: Extract causal action sequence}
10:  $\sigma \leftarrow \text{BACKTRACEACTIONS}(\tau, s_{\text{viol}})$ 
11: {Phase 3: Classify error pattern}
12: pattern  $\leftarrow$  UNKNOWN
13: if  $\exists v \in s_{\text{viol}} : v < 0$  or  $v > \text{MAX\_VALUE}$  then
14:   pattern  $\leftarrow$  MISSINGGUARD
15: else if  $|\{a \in \sigma : \text{CONCURRENT}(a)\}| \geq 2$  then
16:   pattern  $\leftarrow$  RACECONDITION
17: else if  $\exists$  circular resource wait in  $\sigma$  then
18:   pattern  $\leftarrow$  DEADLOCK
19: else
20:   pattern  $\leftarrow$  LOGICERROR
21: end if
22: return  $\langle \phi_{\text{viol}}, s_{\text{viol}}, \sigma \rangle, \text{pattern}$ 

```

Algorithm 3 Generate Clarification Question from Counterexample

Require: Error pattern *pattern*, Trace τ , Domain ontology O

Ensure: Natural language question q with answer options

```

1: {Extract domain entities, actions, and values from trace}
2: entities, actions  $\leftarrow \text{EXTRACTENTITIESANDACTIONS}(\tau, O)$ 
3: values, condition  $\leftarrow \text{EXTRACTSTATECONTEXT}(\tau)$ 
4: scenario  $\leftarrow \text{BUILDSCENARIO}(\tau, \text{entities}, \text{values})$ 
5: if pattern = MISSINGGUARD then
6:   action  $\leftarrow \text{LASTACTION}(\tau)$ 
7:    $q \leftarrow$  "Should +action+ be allowed when +condition+?"
8: else if pattern = RACECONDITION then
9:    $q \leftarrow$  "Should concurrent +actions+ be serialized?"
10: else
11:    $q \leftarrow \text{LLMGENERATEQUESTION}(\text{scenario}, R)$  {Fallback}
12: end if
13: return  $\text{FORMATQUESTIONWITHOPTIONS}(\text{scenario}, q)$ 

```

triggers either the addition of a property to Φ or the strengthening of a guard in δ . The stakeholder therefore never has to reason about which case applies, or about formal logic at all. Two episodes illustrate each direction.

Example 1: translation artifact, resolved by relaxing Φ . During initial generation of *BasicTransfer* the LLM over-constrained the property set, producing $\Phi_{\text{err}} = \{\Box(\text{balance} \geq 100)\}$ from a misread business rule. A legitimate transaction reducing a balance to 50 triggered a TLC violation, which C-GRiD rendered as: “A

Table 3: Examples of Trace-to-Scenario Abstraction

Domain	Raw Counterexample (Snippet)	Generated Question
Banking	State 3: A.bal=100; T1: read(A); T2: read(A); T1: write(A-10); T2: write(A-10)	“Two transfers happened at the exact same time. Both saw the original balance, so the second transfer overwrote the first. Should concurrent transfers be locked?”
Logistics	State 5: Ord.st=Ship; Action: Cancel; Result: Ord.st=Can	“The system allowed an order to be cancelled after it was already marked as ‘Shipped’. Is this transition allowed?”

transaction reduced the balance to 50, violating the rule that balance must remain above 100. Is this allowed?” The stakeholder answered “Yes, any non-negative balance is allowed”, and the framework relaxed the property to $\Phi_{\text{fixed}} = \{\Box(\text{balance} \geq 0)\}$, correcting an error in the initial translation rather than adding a requirement.

Example 2: missing constraint, resolved by strengthening δ . In *MutualExclusion*, V_0 contained the invariant $\Phi = \{\Box \neg(\text{active}_1 \wedge \text{active}_2)\}$ but lacked serialization guards in δ . TLC found a trace in which concurrent execution violated mutual exclusion, and C-GRiD asked: “Both processes accessed the shared resource simultaneously. Should concurrent requests be serialized?” The stakeholder replied “Yes, serialize process execution”, and the system strengthened the action guards in δ to require a mutex lock.

3.7 Regression-Guarded Refinement

Because refinement edits TLA+ directly, an edit that correctly incorporates a new constraint may still disturb a part of the specification that was previously correct. We therefore specify a regression-guarded variant of the loop. After `REFINESPECIFICATION` produces V_{i+1} , the model checker re-checks every property in Φ_{verified} that held on V_i . If any is violated, the refinement is rejected as a regression and the synthesizer is re-prompted with the regression-inducing trace and an instruction to revise the change while preserving prior correctness, bounded to three retries before the run is reported as failed. The guard makes refinement monotone with respect to previously established properties, at the cost of one additional verification sweep per iteration. **This is a design proposal: it is specified here but is not part of the implementation used in Section 4, and none of the results reported below include it.** Implementing it, and reporting the resulting regression rate, is our immediate next step.

3.8 Core Prompt Templates

Two structured system prompts drive the loop. They are keyed to the error-pattern taxonomy of Algorithm 2 and were fixed before the reported runs. *Question generation* receives the trace details and the error-pattern name and is asked to synthesize a clear natural-language question, in domain terminology, about whether the observed behaviour is allowed. *Specification refinement* receives the current TLA+ code, the stakeholder answer and the causal trace,

and is asked to refine the transition guards or properties to incorporate the feedback while avoiding any other modification. The complete prompt set, including the Mode 1 inference prompt and the noisy-oracle prompts, is in the replication package.

3.9 Scope Limitation: Variables Absent from the State Space

Because discovery is driven entirely by the model checker, a missing requirement that refers to a domain variable not present in the current state space cannot produce a counterexample and will never be surfaced by the automated loop. The recall figures reported below are therefore measured over constraints expressible in the variables of the generated model, and should not be read as completeness with respect to the stakeholder’s full intent. Introducing such variables remains a manual analysis step, or must be inferred during initial synthesis.

3.10 Convergence Analysis

Sufficient conditions. Let Σ be finite (as enforced by bounded model checking). If each stakeholder answer is *consistent* (does not contradict previous answers) and *strictly restrictive* (generates a property ϕ_{new} that eliminates at least one previously allowed execution trace), then C-GRiD converges to a verified specification S^* or proves unsatisfiability in finitely many steps. *Argument:* at iteration i a counterexample $\tau_i \in \mathcal{T}(\mathcal{M}_i)$ is found and feedback yields ϕ_{i+1} with $\tau_i \not\models \phi_{i+1}$. Since $\Phi_{i+1} = \Phi_i \cup \{\phi_{i+1}\}$, we have $\mathcal{T}(\mathcal{M}_{i+1}) \subset \mathcal{T}(\mathcal{M}_i)$; the trace set of a finite-state system is finite and strictly reduced at each step, so the loop terminates.

Interaction with property relaxation. The monotonicity assumption is violated by exactly one class of step: relaxation of an over-constrained property (Example 1 above), which weakens Φ and therefore enlarges $\mathcal{T}(\mathcal{M})$. The argument is consequently stated for a two-phase loop. Each answer is classified as relaxing or restricting; a relaxation may be applied to a given property at most once, so the number of relaxation steps is bounded by $|\Phi_0|$, and termination follows from strict trace-set reduction over the restriction steps, which are the only ones that recur.

Behaviour under realistic conditions. This argument assumes correct oracle feedback. Under a noisy oracle, termination is not guaranteed by state-space reduction alone, since incorrect answers may induce cycles or expand the trace set; convergence is then ensured in practice only by the explicit timeout and oscillation detection mechanisms, which trigger after N_{max} iterations or on a repeated specification state.

4 Empirical Evaluation

Evaluating any requirements discovery technique faces a methodological obstacle: how can one quantitatively measure the recovery of requirements that are, by definition, unknown? We adopt a controlled constraint-removal protocol that begins from a complete, formally verified specification and systematically removes a known set of critical constraints, creating a testbed with objectively measurable incompleteness and computable recall and precision.

We state the resulting scope restriction explicitly, because it bounds every claim that follows. What we measure is the recovery of *formal constraints removed from an existing verified specification*.

Table 4: Benchmark Suite Characteristics

Specification	Domain	# Vars	# Constr	Complexity
BasicTransfer	Banking	3	4	Small
ShoppingCart	E-commerce	4	5	Small
OverdraftProtection	Banking	4	6	Medium
PatientRecords	Healthcare	5	5	Medium
MutualExclusion	Concurrency	3	4	Medium
SpecificationName	Generic	2	2	Small
TwoPhaseCommit	Distributed	8	10	Large
AuctionProtocol	Distributed	6	9	Medium
ElevatorController	Embedded	10	15	Large
TrafficLightController	Embedded	7	11	Medium

We do not measure the recovery of unstated *informal* stakeholder requirements, and we do not measure the correctness of the initial natural-language-to-TLA+ translation. These are related but distinct problems, and the second is arguably the harder one. The protocol isolates the neuro-symbolic loop so that recall and precision are computable at all; validating the full elicitation pipeline, from informal documents through to a verified specification, requires the human studies outlined in Section 6.

4.1 Research Questions

- **RQ1 (Effectiveness):** How effectively does C-GRiD recover missing requirements compared to automated baselines under ideal conditions (perfect oracle)?
- **RQ2 (Robustness to Noise):** How does C-GRiD’s performance degrade when interacting with an imperfect, “noisy” stakeholder?
- **RQ3 (Efficiency & Scalability):** Is the C-GRiD process efficient, and how does its performance scale with specification complexity?

4.2 Evaluation Methodology

4.2.1 Benchmark Suite. Our benchmark suite comprises 10 formally verified specifications from public literature, detailed in Table 4. The suite spans multiple domains and complexity levels, from small state machines to larger distributed protocols. Complexity is classified by the number of state variables (#Vars) and constraints (#Constr). The suite is deliberately weighted towards small and medium specifications, since the protocol requires a complete, independently verified ground-truth specification for every benchmark, which is expensive to obtain; the largest, *ElevatorController*, has 10 variables and 15 constraints. We discuss the consequences for external validity in Section 5.2.

4.2.2 Protocol and Baselines. For each specification we created incomplete variants by stratified random removal of constraints, at two levels $k \in \{3, 5\}$, simulating moderately and significantly incomplete requirements. The stratification is a quota over constraint types and k is the size of that quota rather than an achieved count: where a specification holds fewer constraints of a requested type, the sampler takes all available and continues. Between one and three constraints are therefore removed per configuration, and every recall figure uses that number, not k , as its denominator—which

is why values such as 66.7% arise at $k = 5$. We compare against two automated baselines:

- **Gen-Once (One-Shot LLM):** a sophisticated but non-interactive use of LLMs, employing a carefully engineered prompt with few-shot examples of similar specifications. It tests the value of the iterative, verification-driven loop against a purely generative approach.
- **Auto-Repair (CEGIS-style Synthesis):** a purely symbolic refinement approach using the same counterexample analysis as C-GRiD but applying fixed syntactic repair templates (e.g., “add guard: variable > 0 ” or “enforce mutual exclusion by locking the resource”) instead of generating a semantic question. Its complete failure (0% recall) is instructive: generic syntactic templates, without domain semantics or the ability to query a stakeholder, cannot cover the diversity of missing constraints in the suite.

On translation systems as baselines. We deliberately do not compare against ARSENAL [11], NL2LTL [9], NL2CTL [35] or related tools, because they solve a different problem. Those systems map a *given* natural-language requirement to a formula under a fixed behavioural model; they take the requirement set as input and cannot, by construction, propose a requirement that was never stated. Running them on our protocol would measure translation fidelity on the constraints that remain, not recovery of the constraints that were removed, and the resulting recall would be zero by definition rather than by finding. The two lines are complementary rather than competing: a translation front-end would be a natural source of the initial V_0 that C-GRiD then refines, and evaluating that combination is future work.

4.2.3 Implementation Details and Experimental Setup. Experiments were run on an 8-core Intel Xeon CPU with 64 GB RAM. We used the Groq API with the llama-3.3-70b-versatile model, at temperature 0.3 for refinement tasks—chosen in pilot experiments to balance reproducibility against the semantic diversity needed for constraint synthesis. The symbolic backend was the TLC model checker for TLA+, with a bounded depth of $k = 15$, $N_{\max} = 10$ iterations, and a timeout of 300 s per verification run.

To address the “oracle assumption” threat we designed a two-phase evaluation:

- (1) **Perfect Oracle:** A simulated stakeholder that consults S_{GT} (the ground-truth specification) to provide ideal, correct answers, establishing an upper bound.
- (2) **Noisy Oracle:** a simulated stakeholder with a 75% probability of a correct answer, 15% of being “ambiguous” (providing no constraint), and 10% of providing an incorrect, mutated constraint.

We note that this noise model is deliberately simple. It captures per-answer error and abstention, but not the phenomena that make real elicitation hard: partial domain knowledge, intent that evolves as the dialogue proceeds, and answers that contradict earlier ones. It is best read as a stress test of the loop’s error-recovery mechanism rather than as a model of human behaviour. Our full replication package, including the noisy oracle implementation, all prompts and benchmark data, is available online.

4.2.4 Metrics and Analysis. We use three primary metrics: **Recall** (the fraction of removed constraints correctly recovered), **Precision** (the fraction of newly added constraints that were correct), and the **F1 Score**, macro-averaged over runs rather than computed from the mean recall and precision. To evaluate statistical significance we use a paired t-test after confirming normality with a Shapiro-Wilk test, and report Cohen’s d for effect size. We evaluate constraint equivalence using strict semantic matching that preserves structural differences: array-indexed constraints (e.g., `balance[from] >= amount`) are distinguished from non-indexed variants (e.g., `balance >= 0`), as these are semantically different predicates.

Termination categories. A run can stop for reasons that differ sharply in meaning, and a single “converged” flag conflates them. We therefore classify each run into one of four categories. **V** (*verified-convergent*): the checker verified V_i against Φ before $N_{\max}$. **RS** (*recall-saturated*): every removed constraint was recovered, but the loop continued to $N_{\max}$ without a final clean verification. **VT** (*vacuously terminated*): the reduced specification already satisfied Φ_{init} , so no counterexample was produced and no elicitation took place. **F** (*failed*): terminated by oscillation detection or timeout. The convergence rates we report aggregate V, RS and VT; the per-run breakdown is given in Table 6 so that the aggregate is not over-read. In particular, a VT run carries no information about elicitation quality, since the loop was never given a counterexample to work from.

4.3 Results and Analysis

We executed 40 C-GRiD runs, 20 under each oracle, across the ten-specification benchmark at two incompleteness levels, together with 24 baseline runs (12 per baseline) on the six-specification subset described below.

4.3.1 RQ1: Effectiveness. As shown in Table 5, C-GRiD under ideal conditions (perfect oracle) achieved a mean recall of 79.2% and a high precision of 90.0%, indicating that the framework identifies correct missing constraints while largely avoiding the hallucination problem of purely generative models. The strengthened Gen-Once baseline recovered 2.8% of the removed constraints: even with sophisticated prompting and few-shot examples, a non-interactive approach rarely uncovers a missing requirement. The purely symbolic Auto-Repair baseline recovered nothing (0.0% recall), confirming that syntactic template-matching is insufficient for domain-specific semantic constraints.

The baselines were evaluated on a six-specification subset, giving 12 paired configurations, whereas C-GRiD’s figures cover the full suite. We therefore restrict the significance test to the 12 configurations on which all three approaches were run rather than pair measurements taken over different suites. On that subset C-GRiD recovers 65.3% of removed constraints against 2.8% for Gen-Once, a mean difference of 62.5 percentage points (95% CI [34.6, 90.4]), significant by a paired t -test ($t(11) = 4.93$, $p < .001$) with a large effect size ($d_z = 1.42$). Extending both baselines to the full suite is the first item of future work.

Decomposing the aggregate convergence rate by termination category (Table 6) gives a more precise picture than the aggregate alone. Under the perfect oracle, 2 of the 20 configurations were verified-convergent, 13 were recall-saturated, 2 were vacuously

Table 5: Overall Performance Comparison Across All Approaches.

Approach	Recall	Precision	F1	Convergence
Gen-Once (One-Shot) [§]	2.8%	8.3%	0.04	100%*
Auto-Repair (CEGIS) [§]	0.0%	0.0%	0.00	16.7%
C-GRiD (Noisy)	70.8%	80.0%	0.74	75.0% [‡]
C-GRiD (Perfect)	79.2%	90.0%	0.83	85.0% [‡]

*Terminates in 1 iter.; does not imply verification success.

[‡]Aggregate of V, RS and VT; see Table 6.

[§]Evaluated on the six-specification subset; see text.

terminated and 3 failed; under the noisy oracle the split is 2 / 11 / 2 / 5. The dominant category is recall-saturation: in most successful runs every removed constraint had been recovered, but the loop exhausted $N_{\max}$ without reaching a clean verification, because subsequent refinements kept introducing new violations elsewhere (Section 5.1). The appropriate reading of these results is therefore that C-GRiD reliably *finds* missing constraints, while reliably *integrating* them into a re-verified specification remains an open problem.

4.3.2 RQ2: Robustness to Noise. When the simulated stakeholder provided incorrect or ambiguous feedback 25% of the time, C-GRiD’s mean recall dropped by only 8.4 percentage points to a still-strong **70.8%**, and the convergence rate degraded gracefully from 85% to 75%. This suggests the symbolic verifier acts as a corrective mechanism: even when the LLM is led astray by an incorrect answer, a subsequent verification step is likely to produce a new counterexample, giving the framework another opportunity to ask a clarifying question and return to the correct path.

Iteration count. The noisy oracle terminates in fewer iterations on average than the perfect oracle (6.3 vs. 7.1), which is at first sight counter-intuitive. It is an artifact of early termination rather than faster convergence. Incorrect or contradictory constraints drive the loop into oscillation or timeout, and those runs stop after very few iterations, as the failed $k = 5$ rows of Table 6 show. Because the noisy oracle has more failed runs than the perfect oracle, and failures are short, the global mean is pulled down even though the successful runs are no faster.

4.3.3 RQ3: Efficiency and Scalability. C-GRiD converged in an average of 7.1 iterations with the perfect oracle and 6.3 with the noisy oracle. For moderately incomplete specifications ($k = 3$) the perfect oracle achieved 85.0% recall with 90.0% precision and a 100% convergence rate; as incompleteness increased ($k = 5$) recall remained strong at 73.3% with unchanged 90.0% precision, while the convergence rate dropped to 70.0%, a graceful degradation. The framework also handled our largest benchmarks, including ElevatorController with 15 constraints, converging in the majority of runs. One property of these results should be stated plainly. The four largest specifications account for eight of the twenty configurations and all eight reach 100% recall, but they are also the configurations on which the sampler removed the fewest constraints, so they are individually the easiest instances despite the largest surrounding models. They show that the loop is not destabilised by a larger specification; they are not evidence that recall holds as

the number of missing constraints grows. Per-benchmark results appear in Table 6.

In a few cases, such as *OverdraftProtection* at $k = 3$, the framework converged with 0% recall. This occurs when the reduced specification is vacuously correct for the given properties; without a counterexample from the model checker the loop terminates immediately, as there is no feedback to drive the discovery process. Such runs are labelled VT in Table 6 and, as noted above, say nothing about elicitation quality.

5 Discussion

Our results support the hypothesis that formal verification, used as an interactive feedback mechanism for LLMs, is an effective method for discovering missing requirements: the symbolic verifier supplies the rigorous, targeted feedback needed to steer the LLM’s generative capability toward a more complete specification. The 100% convergence rate for moderately incomplete specifications ($k = 3$) under a perfect oracle indicates reliability at the most common scale of requirement gap. As the termination breakdown in RQ1 shows, however, most of those runs are recall-saturated rather than verified-convergent, so what is demonstrated is reliable discovery of the missing constraints rather than reliable production of a re-verified specification.

5.1 Failure Analysis

The drop in convergence for more complex cases ($k = 5$) warrants closer analysis. The three non-convergent perfect-oracle runs are named in the caption of Table 6. In all of them the framework identified at least one correct missing constraint, so elicitation was not the limiting factor.

The failures stemmed from the automated refinement step: while integrating a correct new constraint, the LLM sometimes introduced a subtle error elsewhere in the specification, producing a new and different counterexample. This pattern of *cascading refinement failures* indicates that the automated, correct editing of formal specifications remains an open challenge and our primary direction for future work.

The prevalence of recall-saturated terminations is evidence of the same effect at larger scale: in 13 of the 20 perfect-oracle configurations the loop had already recovered every removed constraint and still could not reach a clean verification within $N_{\max}$, which is the signature of refinements that repair one property while disturbing another. We deliberately do not attach a single corruption rate to this phenomenon, because our present instrumentation does not record, for each refinement, which previously-satisfied properties held before and after the edit. Obtaining that figure requires exactly the re-verification sweep described in Section 3.7, which is why we report the regression guard as the next step rather than as a measured result.

5.2 Threats to Validity

We address four standard categories of validity threats.

5.2.1 Construct Validity. The main threat is the *oracle assumption*: our simulated perfect and noisy stakeholders are idealized, whereas real stakeholders are ambiguous, inconsistent and change

their minds. *Mitigation*: we frame the study as validating a necessary precondition—the efficacy of the neuro-symbolic loop under controlled, repeatable conditions—rather than as a study of the socio-technical elicitation process. Strong performance with a perfect oracle and graceful degradation under noise establish an upper bound and evidence of resilience, motivating a full human study. A second threat is the *realism of constraint removal*; we use stratified sampling across constraint types (guards, invariants) to simulate diverse omissions, and note that the protocol, while synthetic, is what makes objective metrics computable.

A third threat is the *initial formalization step*. In practice, translating informal requirements into V_0 can introduce errors that mimic missing requirements. C-GRiD is designed to surface these symmetrically rather than to suppress them (Section 3.6), and we model-check V_0 before entering the loop. However, our protocol begins from existing formal specifications rather than from informal documents, so it cannot measure how often each case arises in practice. Quantifying that distribution, and adding automated semantic validation of the initial translation, requires a study anchored in natural-language requirements documents, which we have not conducted and which we regard as the most important gap in this evaluation.

5.2.2 Internal Validity. The main threat is *LLM non-determinism*. *Mitigation*: all experiments used a fixed random seed, and the reported figures are re-derivable from the archived run records.

5.2.3 External Validity. Threats include *benchmark representativeness*, *model dependence* and *scalability*. Our 10 specifications span diverse domains and are drawn from established formal methods literature, but do not capture industrial complexity. Results are specific to llama-3.3-70b-versatile; the framework is architecturally model-agnostic and we expect the *relative* improvement over a one-shot baseline to hold, since the contribution lies in the iterative process rather than in the model.

Scalability is the most serious of the three, and we state it plainly. Our largest benchmark has 10 variables and 15 constraints, whereas industrial specifications may have hundreds of variables and compositional structure. C-GRiD also operates at the level of the whole specification rather than the individual requirement, so each verification call scales with the entire model rather than with the requirement under discussion, and the state-space explosion problem applies in full. Behaviour at 50+ variables is empirically unvalidated. We therefore regard compositional verification not as an optional extension but as a prerequisite for industrial deployment, and we do not claim applicability at that scale on the basis of the present results.

5.2.4 Conclusion Validity. The primary threat is the *small sample size*: 20 configurations per oracle for C-GRiD, and 12 paired configurations for the baseline comparison. *Mitigation*: the effect on the paired subset ($d_z = 1.42$, $t(11) = 4.93$) is large enough to detect at this size, and test assumptions were verified (normality via Shapiro-Wilk, $p > 0.05$). We regard the baseline comparison as indicative rather than definitive until both baselines are run over the full suite.

Table 6: Detailed Experimental Results per Benchmark for C-GRiD with Perfect vs. Noisy Oracles. Termination codes: V verified-convergent, RS recall-saturated (all removed constraints recovered, loop ran to $N_{\max}$), VT vacuously terminated (no counterexample generated), F failed. Under the perfect oracle, failures were oscillation (MutualExclusion, PatientRecords) and timeout (ShoppingCart).

Specification	k	C-GRiD (Perfect Oracle)				C-GRiD (Noisy Oracle)			
		Recall	Iter.	Conv.?	Term.	Recall	Iter.	Conv.?	Term.
BasicTransfer	3	50.0%	2	✓	V	50.0%	2	✓	V
BasicTransfer	5	0.0%	1	✓	VT	0.0%	1	✓	VT
OverdraftProtection	3	0.0%	1	✓	VT	0.0%	1	✓	VT
OverdraftProtection	5	33.3%	2	✓	V	33.3%	2	✓	V
MutualExclusion	3	100.0%	10	✓	RS	100.0%	10	✓	RS
MutualExclusion	5	66.7%	2	✗	F	66.7%	2	✗	F
TwoPhaseCommit	3	100.0%	10	✓	RS	100.0%	10	✓	RS
TwoPhaseCommit	5	100.0%	10	✓	RS	100.0%	10	✓	RS
AuctionProtocol	3	100.0%	10	✓	RS	100.0%	10	✓	RS
AuctionProtocol	5	100.0%	10	✓	RS	50.0%	2	✗	F
SpecificationName	3	100.0%	10	✓	RS	100.0%	10	✓	RS
SpecificationName	5	100.0%	10	✓	RS	50.0%	2	✗	F
ShoppingCart	3	100.0%	10	✓	RS	100.0%	10	✓	RS
ShoppingCart	5	66.7%	2	✗	F	33.3%	2	✗	F
ElevatorController	3	100.0%	10	✓	RS	100.0%	10	✓	RS
ElevatorController	5	100.0%	10	✓	RS	100.0%	10	✓	RS
TrafficLightController	3	100.0%	10	✓	RS	100.0%	10	✓	RS
TrafficLightController	5	100.0%	10	✓	RS	100.0%	10	✓	RS
PatientRecords	3	100.0%	10	✓	RS	100.0%	10	✓	RS
PatientRecords	5	66.7%	2	✗	F	33.3%	2	✗	F
Average	–	79.2%	7.1	85.0%	–	70.8%	6.3	75.0%	–

6 Conclusion

Eliciting formally verifiable specifications from informal stakeholder requirements remains a recognized challenge in software engineering. This paper introduced C-GRiD, a neuro-symbolic framework that integrates formal verification directly into iterative requirements discovery. Rather than using model checking solely for post-development validation, C-GRiD translates counterexample traces into natural-language queries, establishing an LLM-mediated feedback loop that systematically elicits missing constraints from stakeholders.

Using a controlled constraint-removal methodology, C-GRiD achieved a constraint recovery recall of 79.2% with a perfect oracle and 70.8% with a simulated noisy oracle, an improvement over both one-shot generative and template-based synthesis baselines. These findings support the hypothesis that combining LLM-based generation with symbolic verification supplies the logical grounding needed to reduce semantic hallucination and facilitate the discovery of unstated requirements.

Three limitations define our future work. First, integrating elicited constraints can introduce syntax or logic errors into the existing formal model; the prevalence of recall-saturated terminations suggests this is the dominant obstacle to full convergence, and we will implement and evaluate the regression-guarded loop of Section 3.7 alongside verifiable model transformations for monotonic refinement. Second, having established baseline performance with a simulated noisy oracle, we plan controlled user studies with software engineers and domain experts to evaluate the framework under natural human ambiguity, contradiction and cognitive load.

Such a study is also the only way to close the gap identified in Section 4, since our present evaluation measures recovery of formal constraints rather than of informal stakeholder intent. Third, to mitigate state-space explosion we will explore compositional verification, testing whether LLMs can decompose complex requirements into modular, independently verifiable sub-specifications, and extend input processing to semi-formal artifacts such as sequence diagrams and execution logs. Integrating model checking directly into the elicitation loop offers a structured, automated mechanism for requirements discovery and a practical route toward more complete and verified formal specifications.

Data Availability Statement

All data and artifacts underlying the reported results—the formal specification benchmarks, the simulated stakeholder prompts, the complete experimental results and the Python implementation of C-GRiD—are publicly available. The replication package is archived at <https://doi.org/10.5281/zenodo.19222714>.

Declaration on the Use of Generative AI

The authors used large language models (Google Gemini and Claude) to assist with code development and language editing. All code was reviewed, tested and validated by the authors, who take full responsibility for the correctness of the implementation and the integrity of all reported results.

References

- [1] Camila Almeida, Isaque Copque, Alvaro Oliveira, Murilo Arouca, Adriano Barbosa, Sávio Freire, Manoel Mendonça, and Julio Cesar S. P. Leite. 2025. From Elicitation Interviews to Software Requirements: Evaluating LLM Performance in Requirement Generation. [doi:10.29327/1588952.28-12](https://doi.org/10.29327/1588952.28-12)
- [2] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2024. *Advancing Requirements Engineering Through Generative AI: Assessing the Role of LLMs*. [doi:10.1007/978-3-031-55642-5_6](https://doi.org/10.1007/978-3-031-55642-5_6)
- [3] Arshad Beg, Diarmuid O'Donoghue, and Rosemary Monahan. 2025. Leveraging LLMs for Formal Software Requirements – Challenges and Prospects. [arXiv:2507.14330](https://arxiv.org/abs/2507.14330) [cs.SE] <https://arxiv.org/abs/2507.14330>
- [4] Arshad Beg, Diarmuid O'Donoghue, and Rosemary Monahan. 2025. A Short Survey on Formalising Software Requirements using Large Language Models. [arXiv:2506.11874](https://arxiv.org/abs/2506.11874) [cs.SE] <https://arxiv.org/abs/2506.11874>
- [5] Yufan Cai, Zhe Hou, David Sanan, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proc. ACM Program. Lang.* 9, POPL, Article 69 (Jan. 2025), 33 pages. [doi:10.1145/3704905](https://doi.org/10.1145/3704905)
- [6] Cheliger Cheliger, Jingwei Huang, Guosong Wu, Nadia Bhuiyan, Yuan Xu, and Yong Zeng. 2022. Machine learning in requirements elicitation: a literature review. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 36 (2022), e32. [doi:10.1017/S0890060422000166](https://doi.org/10.1017/S0890060422000166)
- [7] António Miguel Rosado da Cruz and Estrela Ferreira Cruz. 2025. Machine Learning Techniques for Requirements Engineering: A Comprehensive Literature Review. *Software* 4 (2025), Issue 3. [doi:10.3390/software4030014](https://doi.org/10.3390/software4030014)
- [8] Timothy Elvira, Tyler Thomas Procko, and Omar Ochoa. 2024. Requirements Elicitation for Machine Learning Applications: A Research Preview. In *Proceedings - 2024 Conference on AI, Science, Engineering, and Technology, AlxSET 2024*. [doi:10.1109/AlxSET62544.2024.00042](https://doi.org/10.1109/AlxSET62544.2024.00042)
- [9] Francesco Fuggitti and Tathagata Chakraborti. 2023. NL2LTL - A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*, Vol. 37. [doi:10.1609/aaai.v37i13.27068](https://doi.org/10.1609/aaai.v37i13.27068)
- [10] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive synthesis of temporal specifications from examples and natural language. *Proceedings of the ACM on Programming Languages* 4 (2020), Issue OOPSLA. [doi:10.1145/3428269](https://doi.org/10.1145/3428269)
- [11] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2016. ARSENAL: Automatic requirements specification extraction from natural language. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 9690. [doi:10.1007/978-3-319-40648-0_4](https://doi.org/10.1007/978-3-319-40648-0_4)
- [12] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2021. Automated formalization of structured natural language requirements. *Information and Software Technology* 137 (2021). [doi:10.1016/j.infsof.2021.106590](https://doi.org/10.1016/j.infsof.2021.106590)
- [13] Eleonora Giunchiglia, Fergus Imrie, Mihaela van der Schaar, and Thomas Lukasiewicz. 2025. Machine learning with requirements: A manifesto. *Neurosymbolic Artificial Intelligence* 1 (2025). [doi:10.3233/naai-240767](https://doi.org/10.3233/naai-240767)
- [14] George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2025. Seeking Specifications: The Case for Neuro-Symbolic Specification Synthesis. [arXiv:2504.21061](https://arxiv.org/abs/2504.21061) [cs.SE] <https://arxiv.org/abs/2504.21061>
- [15] Mohammad Kasra Habib, Daniel Grazier, and Stefan Wagner. 2025. ReqBrain: Task-Specific Instruction Tuning of LLMs for AI-Assisted Requirements Generation. [arXiv:2505.17632](https://arxiv.org/abs/2505.17632) [cs.SE] <https://arxiv.org/abs/2505.17632>
- [16] William T. Hallahan, Ranjit Jhala, and Ruzica Piskac. 2025. Counterexample-Guided Inference of Modular Specifications. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 140 (April 2025), 28 pages. [doi:10.1145/3720505](https://doi.org/10.1145/3720505)
- [17] F P Brooks Jr. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 20 (1987).
- [18] Kamaljit Kaur and Parminder Kaur. 2024. The application of AI techniques in requirements classification: a systematic mapping. *Artificial Intelligence Review* 57 (2024), Issue 3. [doi:10.1007/s10462-023-10667-1](https://doi.org/10.1007/s10462-023-10667-1)
- [19] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, and Chenghua Lin. 2023. Requirement Formalisation Using Natural Language Processing and Machine Learning: A Systematic Review. In *International Conference on Model-Driven Engineering and Software Development*, Vol. 1. [doi:10.5220/0011789700003402](https://doi.org/10.5220/0011789700003402)
- [20] A Solar Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley. <https://people.csail.mit.edu/asolar/papers/thesis.pdf>
- [21] Tong Li, Xinran Zhang, Yunduo Wang, Qixiang Zhou, Yiting Wang, and Fangqi Dong. 2024. Machine learning for requirements engineering (MLARE): A systematic literature review complemented by practitioners' voices from Stack Overflow. [doi:10.1016/j.infsof.2024.107477](https://doi.org/10.1016/j.infsof.2024.107477)
- [22] Robert Lorch, Baoluo Meng, Kit Siu, Abha Moitra, Michael Durling, Saswata Paul, Sarat Chandra Varanasi, and Craig Mcmillan. 2024. Formal Methods in Requirements Engineering: Survey and Future Directions. In *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormalISE) (Lisbon, Portugal) (FormalISE '24)*. Association for Computing Machinery, New York, NY, USA, 88–99. [doi:10.1145/3644033.3644373](https://doi.org/10.1145/3644033.3644373)
- [23] Kumar Manas, Stefan Zwicklbauer, and Adrian Paschke. 2024. TR2MTL: LLM based framework for Metric Temporal Logic Formalization of Traffic Rules. In *IEEE Intelligent Vehicles Symposium, Proceedings*. [doi:10.1109/IV55156.2024.10588650](https://doi.org/10.1109/IV55156.2024.10588650)
- [24] Yuchen Mao, Tianci Zhang, Xu Cao, Zhongyao Chen, Xinkai Liang, Bochen Xu, and Hao Fang. 2024. NL2STL: Transformation from Logic Natural Language to Signal Temporal Logics using Llama2. In *Proceedings of the IEEE International Conference on Cybernetics and Intelligent Systems, CIS*. Issue 2024. [doi:10.1109/CIS-RAM61939.2024.10672997](https://doi.org/10.1109/CIS-RAM61939.2024.10672997)
- [25] M. Martignano, A. Damiani, L. Nucciarelli, D. Gui, and S. Magalini. 2025. Software Verification and Generative AI Some Practical Examples and Considerations. *Ada Lett.* 44, 2 (June 2025), 51–55. [doi:10.1145/3742939.3742946](https://doi.org/10.1145/3742939.3742946)
- [26] Chedhli Masmoudi, Pascale Marange, Eric Bonjour, Eric Levrat, and Alain Kerbrat. 2022. Adopting formal methods on requirements verification and validation for cyber-physical systems: A systematic literature review. *IFAC-PapersOnLine* 55, 10 (2022), 3274–3279. [doi:10.1016/j.ifacol.2022.10.131](https://doi.org/10.1016/j.ifacol.2022.10.131) 10th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2022.
- [27] Daniel Mendoza, Christopher Hahn, and Caroline Trippel. 2024. Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers. In *Formal Methods in Computer-Aided Design, FMCAD*. Issue 2024. [doi:10.34727/2024/isbn.978-3-85448-065-5_17](https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_17)
- [28] Daniel Mendoza, Anastasia Mavridou, Andreas Katis, and Caroline Trippel. 2026. Automating Requirements Formalization: Using LLMs and Low-Complexity Distinguishing Traces for Semantic Validation. In *Proceedings of the 48th International Conference on Software Engineering (ICSE)*. https://cs.stanford.edu/~trippel/pubs/mendoza_ICSE26.pdf
- [29] Daniel Neider and Rajarshi Roy. 2024. What is formal verification without specifications? A survey on mining LTL specifications. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Springer, 109–125.
- [30] Malik Abdul Sami, Muhammad Waseem, Zheyang Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. 2024. AI based Multiagent Approach for Requirements Elicitation and Analysis. [arXiv:2409.00038](https://arxiv.org/abs/2409.00038) [cs.SE] <https://arxiv.org/abs/2409.00038>
- [31] Vaishali Siddeshwar, Sanaa Alwidian, and Masoud Makrehchi. 2024. A Systematic Review of AI-Enabled Frameworks in Requirements Elicitation. *IEEE Access* 12 (2024). [doi:10.1109/ACCESS.2024.3475293](https://doi.org/10.1109/ACCESS.2024.3475293)
- [32] Riad Sonbol, Ghaida Rebdawi, and Nada Ghneim. 2022. The Use of NLP-Based Text Representation Techniques to Support Requirement Engineering Tasks: A Systematic Mapping Review. [doi:10.1109/ACCESS.2022.3182372](https://doi.org/10.1109/ACCESS.2022.3182372)
- [33] Maria Jose Torres-Igartua, Angel J. Sanchez-Garcia, Jorge Octavio Ocharan-Hernandez, and Sodel Vazquez-Reyest. 2024. Analysis of Natural Language Processing Techniques and Tools for Requirements Elicitation: A Systematic Literature Review. In *Proceedings - 2024 12th International Conference in Software Engineering Research and Innovation, CONISOFT 2024*. [doi:10.1109/CONISOFT63288.2024.00017](https://doi.org/10.1109/CONISOFT63288.2024.00017)
- [34] Weiqi Wang, Marie Farrell, Lucas C. Cordeiro, and Liping Zhao. 2025. Supporting Software Formal Verification with Large Language Models: An Experimental Study. [arXiv:2507.04857](https://arxiv.org/abs/2507.04857) [cs.SE] <https://arxiv.org/abs/2507.04857>
- [35] Mengyan Zhao, Ran Tao, Yanhong Huang, Jianqi Shi, Shengchao Qin, and Yang Yang. 2024. NL2CTL: Automatic Generation of Formal Requirements Specifications via Large Language Models. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 15394 LNCS. [doi:10.1007/978-981-96-0617-7_1](https://doi.org/10.1007/978-981-96-0617-7_1)

Received 2026-03-26; accepted 2026-06-18