SPECAT: ASSESSING AND CATEGORIZING FORMAL SPECIFICATIONS FROM NATURAL LANGUAGE

Anonymous authors
Paper under double-blind review

ABSTRACT

As AI systems generate code at increasing scale, providing strong assurance such as formal verification becomes increasingly important. However, formal verification is only as good as the specification it verifies. In this work, we study formal specification synthesis and evaluation, a standalone but equally important problem. We first formulate *spec-evaluation* in the real-world setting: assessing a synthesized formal specification against a brief natural-language(NL) description and a few behavioral examples indicating user intent. We introduce the *Spec-Evaluation Methodology*, which assesses the relations among an NL description, behavioral examples, a candidate formal specification, and formal test artifacts. It interprets verifier or execution outcomes with critic assessments. We further propose SPECAT, an agentic workflow that synthesizes formal specifications and refines them through two complementary loops. *Behavioral-oracle refinement* uses user-supplied examples and verifier feedback, while *counterexample-guided refinement* uses an LLM-based Spec Critic to seek description–specification mismatches beyond those examples. Proposed challenges are formally evaluated, while ultimately under-specified or unresolved cases are left for human clarification.

We evaluate SPECAT on 704 benchmark-backend unit spanning Dafny, Verus, Why3, and Frama-C. Rather than returning a specification alone, SPECAT categorizes each candidate with an evidence-backed disposition: 391 are ENDORSED, 77 are REPAIRED following a machine-supported challenge, 70 retain an unrepaired challenge (DEFECT), and 86 are deferred because the available evidence is UNADJUDICATED or INCONCLUSIVE. The remaining 80 fail behavioral-oracle refinement. Together, these results show that SPECAT turns specification generation into a testable assurance step: it produces not only a candidate contract, but also machine-grounded evidence and an explicit disposition for that contract.

1 INTRODUCTION

Agentic coding tools now write functions, modules, and entire systems from a few sentences of natural language (Anthropic, 2026; OpenAI, 2026). Their outputs often compile and pass a set of tests, but neither property establishes that they implement the user's request. As more of the development process is automated, there are fewer opportunities for the human review that traditionally catches deviations from intent (Lahiri, 2026). Verification-aware programming languages or frameworks, such as Dafny (Leino, 2010), Why3 (Filliâtre & Paskevich, 2013), Frama-C (Baudin et al., 2021), and Verus (Lattuada et al., 2023) offer a scalable, machine-checkable way to establish that code meets a given specification.

Recent systems increasingly generate implementations, formal specifications, and machine-checkable proofs together, raising the prospect of *end-to-end verified code generation* (Deng et al., 2025) (Yang et al., 2025). In such a pipeline, formal verification establishes that generated code satisfies the generated specification. It does not establish that the specification captures the original natural-language request. A weak postcondition, an overly restrictive precondition, or an incorrect edge-case interpretation may all be verified perfectly. Moreover, generated code and specification may agree with each other while sharing the same misunderstanding of the task (Lahiri, 2026).

A formal specification makes the system's interpretation of a natural-language request explicit, in a machine-checkable form that can be challenged directly. Specification generation and testing are

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

thus a *standalone* problem, not merely an intermediate by-product of code generation. When the specification later guides implementation and verification, testing it can additionally expose a flawed interpretation before effort is invested in code that is correct only relative to that flawed contract.

We call this task *spec-evaluation*: assessing a candidate contract against a provided natural-language description and a small set of behavioral examples supplied by the user or programmer. The target is alignment with the stated description, not unexpressed private intent; ambiguities are retained for clarification or human review.

This setting differs from ordinary software testing in three ways (Goodenough & Gerhart, 1975). First, specifications are inputs to a verifier, and a failed check may reflect a verifier's own limitation rather than a semantic error of code implementations. Second, behavioral examples must be translated into backend-specific formal artifacts, and this translation can be unfaithful or masking. Third, a finite behavioral oracle cannot rule out a specification that is vacuous, too weak, or too restrictive on unobserved examples. Thus, verifier acceptance and agreement with a finite set of examples are useful but insufficient evidence of specification-description alignment.

Existing approaches to judging a specification each sidestep one horn of this problem and inherit another. Matching a generated specification against a hand-written reference specification (Ye et al., 2025), presumes exactly the trusted artifact we lack, and such references are costly to write and themselves error-prone. Checking spec–code equivalence (Zeng et al., 2026) removes the reference but reintroduces the implementation, and falls back on an unanchored language-model judge to connect the result to intent. One can also ask an LLM to judge the specification directly against the natural-language intent, but an ungrounded judgment provides no machine-checkable evidence for its conclusion.

To address this gap, we introduce a systematic methodology for *Spec-Evaluation* illustrated in Figure 1, which assesses four essential relations among a natural-language description D, candidate formal specification S, informal behavioral examples $\mathcal{O}$, and formal test artifacts T. A *Verifier* evaluates S–T, while a *Semantic Critic*, *Translation Critic*, and *Spec Critic* respectively assess the O–$\mathcal{S}$, O–$\mathcal{T}$, and S–D relations. These agentic assessments are diagnostic evidence rather than semantic verdicts: the system cross-checks them against backend evidence and retains unresolved disagreement for human review.

Building upon this methodology, we introduce SPECAT, an agentic workflow that generates and refines formal specifications from a description and behavioral examples. SPECAT coordinates a *Spec-Writer* and an *Advisor* with the *Test-Writer*, and the actors defined in the Spec-Evaluation methodology. It first performs *Behavioral-oracle refinement*: the Advisor uses the evaluation report to revise the specification, repair formal test artifacts, or reformulate obligations that the backend cannot establish based on the available behavioral examples. Candidates that clear this phase enter *Counterexample-guided refinement*, where a Spec Critic seeks potential mismatches with the description beyond the current behavioral oracle. A machine-supported, Advisor-accepted counterexample is added to the oracle and returns the candidate to refinement; unresolved cases are retained for human review. SPECAT returns each candidate specification with an evidence-backed outcome label.

We evaluate SPECAT on 704 suite–backend units spanning Dafny, Verus, Why3, and Frama-C. Rather than returning a specification alone, the workflow assigns each of the 624 candidates that pass initial diagnostic checks on available examples an evidence-backed disposition: 391 are ENDORSED, 77 are REPAIRED after a machine-supported challenge, 70 retain an unrepaired machine-supported challenge, and 86 are deferred because the available evidence is UNADJUDICATED or INCONCLUSIVE. A manual audit finds 58 of 60 sampled ENDORSED specifications (97%) and 45 of 51 REPAIRED specifications (88%) adequate relative to their written descriptions and target-language interfaces. On Codeforces-100, where an independent held-out execution oracle is available, counterexample-guided refinement increases the number of clean specifications among the same 88 diagnostic entrants from 46 (52%) to 54 (61%) and reduces the hidden-row violation rate from 5.5% to 1.1%.

In summary, we make the following contributions:

- We formulate *spec-evaluation*, the problem of assessing a generated formal specification against a natural-language description and intent-grounded behavioral examples, with no reference specification or implementation. We address the alignment gap between specification and description left open by end-to-end code verification.

- We propose the *Spec-Evaluation Methodology*, a role-based method for formalizing behavioral examples in a target backend and interpreting the resulting evidence. It combines verifier outcomes, blind diagnostic critic assessments, and an independent Spec Critic that evaluates the candidate specification against the provided natural-language description beyond the finite behavioral oracle.
- We introduce SPECAT, a counterexample-guided workflow that generates, tests, and refines formal specifications from natural-language description and limited behavioral examples. SPECAT returns each candidate with an evidence-backed outcome label and is evaluated on 704 units across four verification backends, including an execution-oracle setting that evaluates counterexample-guided refinement against held-out execution behavior.

2 RELATED WORK

2.1 GENERATING FORMAL SPECIFICATION FROM NATURAL LANGUAGE

Translating natural language into formal specification has a long history that predates LLMs. Early systems such as C2S (Zhai et al., 2020) mined specifications by aligning NL comment tokens with JML specification tokens and synthesized candidates under a syntax-guided template, whereas JDoctor (Blasi et al., 2018) maps Javadoc comments onto procedure pre- and postconditions. Both are bounded by handcrafted grammars and matchers, and target what the surrounding text already states rather than the user's underlying intent. LLM-based generation lifted this restriction, producing specifications from unrestricted natural language across different formalisms: temporal logic (Cosler et al., 2023), post-conditions as executable assertions (Endres et al., 2024), ACSL (Su et al., 2025), and rich logics of verification-aware languages such as Dafny (Lahiri, 2024).

A second body of work treats specification generation as one stage of an end-to-end verified-code pipeline. CLEVER (Thakur et al., 2026) and VERINA (Ye et al., 2025) elicit a specification alongside an implementation and proof in Lean, VerifyThisBench (Deng et al., 2025) requires all three at once from a natural-language problem, and Clover (Sun et al., 2024) accepts a candidate only when code, docstring, and annotations agree.

2.2 SPECIFICATION INFERENCE FROM CODE

A separate line infers specifications from an existing implementation—the opposite direction from intent formalization—using LLMs coupled with program analysis and verifier feedback. SpecGen (Ma et al., 2025) generates JML specifications for Java through conversational prompting and a mutation-based phase that selects verifiable candidates; AutoSpec (Wen et al., 2024) hierarchically decomposes C programs and lets static analysis and a verifier drive the LLM, validating each round to curb error accumulation. A related task, invariant generation, infers the loop invariants and intermediate annotations a proof requires—classically by dynamic mining (Daikon (Ernst et al., 2007)) or static guess-and-check (Flanagan & Leino, 2001), and with learning-based methods such as Code2Inv (Si et al., 2018). Both take the program (or its executions) as their source of truth: they summarize or constrain code that already exists, and so cannot formalize intent when no trusted implementation is available, nor reveal that an implementation is itself wrong.

2.3 EVALUATING SPECIFICATION QUALITY

Evaluating a candidate specification is the harder problem. Prior work scores a specification by its consistency with input/output oracles and its ability to reject output mutations (Endres et al., 2024; Lahiri, 2024); a related line instead matches the specification to a ground-truth reference, through testing-based subset relations (Ye et al., 2025), or cross-artifact consistency (Sun et al., 2024). VeriEquivBench (Zeng et al., 2026) drops the ground-truth specification, scoring spec–code mutual implication formally and judging intent by back-translating the specification to natural language for an LLM judge—yet it still requires the implementation and rests its intent check on an unanchored judgment. All of these draw their signal from an implementation, a reference specification, or reference behavior, and target a single verifier.

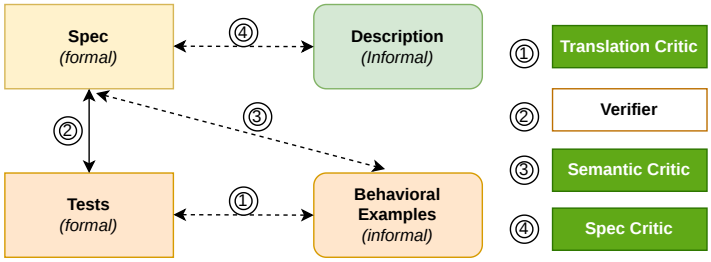

Figure 1: Spec-Evaluation relates four artifacts: an informal description D, candidate formal specification S, informal behavioral examples $\mathcal{O}$, and formal test artifacts T. It assumes that D and $\mathcal{O}$ are aligned, and evaluates the remaining four relations needed to connect the available evidence to S.

3 Spec-Evaluation: Assessing a Synthesized Specification

Formal verification establishes code–specification consistency, however, it does not indicate whether a spec adequately captures a provided natural-language description. We formulate the problem of description-spec alignment in Section 3.1, and describe the Spec-Evaluation methodology in Section 3.2.

3.1 The Problem: Evaluating a Specification Against a Natural Language Description

A problem instance consists of (a) a natural-language (NL) description D of the requested behavior, and (b) a set of language-agnostic *intent-grounded behavioral examples* $O = \{(x_i, e_i)\}^n$, where each x_i describes a concrete behavior, including the relevant inputs and, when applicable, an output or post-state, and $e_i \in \{\mathsf{true}, \mathsf{false}\}$. The label $e_i \in \{\text{true}, \text{false}\}$ specifies whether the provided description classifies that behavior as admissible or inadmissible.

Given a formal specification S, expressed as a precondition–postcondition contract in a verifier's logic, the objective is to assess whether S aligns with D. In particular, the objective concerns the stated description, not private intent that the description does not express.

Unlike conventional software testing, in which a program is executed on concrete inputs and its outputs are directly compared against a test oracle, specification testing operates on a formal artifact, e.g. formal tests T translated from O, which is interpreted by a verifier. The resulting verification outcome depends on the behavior of the prover, the fidelity of the formalized examples, and the coverage of the available behavioral oracle. Three properties characterize this setting, and together they motivate the system in Section 3.2.

1. **Verification outcomes are indirect evidence.** A specification is primarily analyzed by a *verifier* rather than executed as an implementation. Some backends can execute a subset of specification constructs on concrete values, but this capability depends on both the backend and the particular contract. Our methodology therefore does not assume that direct execution is available.

   In conventional testing, a failing test typically indicates a discrepancy between the implementation and the oracle. However, a verification failure or timeout may arise from limitations of the verification engine rather than from a genuine disagreement between the specification and the intended semantics. Verifier outcomes must therefore be interpreted as diagnostic evidence rather than treated as direct evidence of specification errors.

2. **The behavioral examples must be formalized, and the formalization is fallible.** The behavioral oracle is language-agnostic and each example must be translated into formal test artifact in the target verification language. One approach is to leverage a large language model for this translation; however, the resulting formalization may introduce errors and cannot be assumed to preserve the original intent. More generally, the formalization process is error-prone: it may be unfaithful, encoding incorrect expected values, reversing the

intended assertion, or omitting test cases. A passing formal test artifact is consequently not valid evidence unless the artifact itself faithfully represents its corresponding behavioral example.

3. **The behavioral oracle is incomplete.** A finite set of examples cannot characterize every behavior of a function, especially when contracts quantify over unbounded input space. Agreement with all available examples establishes agreement with the behavioral oracle, not complete alignment with the NL description. This limitation is shared with conventional software testing. A specification that agrees with every example may still be wrong on inputs not covered by the test set.

These difficulties mean that no single verification outcome can establish description–spec (D–S) alignment. Therefore we construct and cross-check several forms of evidence. In addition to the description D and candidate specification S, the proposed evaluation system uses informal behavioral examples O supplied by the user or programmer and backend-specific formal test artifacts T. Section 3.2 makes these four artifacts and their assessed relations explicit.

3.2 The Spec-Evaluation Methodology

A spec-evaluation instance contains four artifacts shown in Figure 1. The user or programmer supplies an informal natural-language description D and a set of informal behavioral examples O. The system evaluates a candidate formal specification S and a set of backend-specific formal test artifacts T. When formal tests are not supplied, a Test Agent translates the behavioral examples into T.

These four artifacts induce six possible pairwise relations. The system assumes that the user-supplied description and behavioral examples are aligned. When they are ambiguous or inconsistent, the appropriate outcome is clarification rather than an automated claim of correctness. It also does not directly assess the relation between D and T, because formal tests encode selected examples rather than the full description. Instead, it assesses the four essential relations that connect the informal evidence to the candidate specification: ① example–test translation fidelity (O–T) , ② spec–test satisfaction (S–T), ③ example–spec agreement (O–S), and ④ description–spec alignment (D–S) .

Informal Example–Formal Test. A *Translation Critic* assesses whether each formal test artifact in T faithfully represents its corresponding behavioral example in O. It identifies errors such as an incorrect input, reversed polarity, omitted condition, or overly weak assertion. This role is agentic because the relation connects a formal artifact to informal evidence.

Formal Spec–Formal Test. A verifier or an executor evaluates the formal test artifacts T against the candidate specification S. This relation is machine-grounded, but its outcome is indirect evidence: a non-pass may arise from a semantic disagreement, an unfaithful test artifact, or a limitation of the selected backend.

Informal Example–Formal Spec. The *Semantic Critic* assesses whether the candidate specification S should admit or reject each behavioral example in $\mathcal{O}$. Its prediction is made before observing the verifier or executor outcome. Comparing this prediction with the machine-grounded result helps distinguish a likely specification mismatch from a test-translation defect or backend limitation (see Table 13).

Informal Description–Formal Spec. The *Spec Critic* assesses whether the candidate specification S captures the provided description D beyond the available behavioral examples. It returns either an endorsement or a potential mismatch. The system does not treat this assessment as a semantic verdict: it cross-checks the Spec Critic's decision against the evidence obtained from formal tests. Agreement supports the assessment, whereas disagreement is retained for human review.

4 SPECAT: Diagnostic and Counterexample-Guided Specification Refinement

In this section, we present SPECAT, an agentic workflow that generates and refines formal specifications building on the Spec-Evaluation methodology described in the previous section. SPECAT

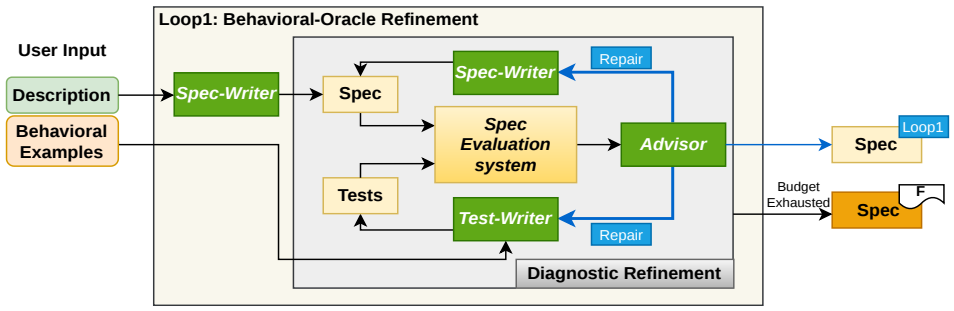

Figure 2: Overview of SPECAT's behavioral-oracle refinement loop. The loop formalizes user-supplied behavioral examples and uses verifier or execution feedback to diagnose and refine the candidate specification. It either advances a provisionally supported specification to counterexample-guided refinement or returns FAILURE when the diagnostic budget is exhausted

includes a *Spec-Writer* agent, which drafts and revises the candidate specification, and a *Test-Writer* agent, which synthesizes formal assertions. The core controller *Advisor* interacts with the verifier and critics in the evaluation system and coordinates with the Spec-Writer and Test-Writer to refine specs from the evidence produced by the evaluation system.

SPECAT includes two loops. In the first loop, it refines a candidate using the evidence from user-supplied behavioral examples, as shown in Figure 2. Candidates ($Spec^{Loop1}$) with no outstanding diagnostic conflict enter the second *Counterexample-guided refinement* loop shown in Figure 3. This loop challenges provisionally supported specifications against the description. A validated counterexample re-enters diagnostic refinement, whereas an endorsement from the Spec Critic finalizes the specification. The system outputs a final spec and a label associated with the specification.

SPECAT generates and validates a specification rather than an implementation. It therefore provides the specification stage that can precede code synthesis and formal verification in an end-to-end verified-code pipeline. Next we will first describe the diagnostic refinement procedure used by the two loops and then discuss the overall workflow.

4.1 DIAGNOSTIC REFINEMENT

Diagnostic refinement is the evidence-to-action procedure used by SPECAT. Given a description D, behavioral examples O, and a candidate specification S, the Test-Writer constructs formal test artifacts T. The evaluation system (as described in Section 3.2) then evaluates the relations among O, T, S and produces a diagnostic report. The report combines the backend outcome with the independent assessments of the Semantic Critic and Translation Critic. Note that Diagnostic refinement does not invoke the Spec Critic, because it does not assess description–spec alignment.

The Advisor reads the feedback from the spec evaluation system, the current candidate, and the refinement history. It either advances the spec to the next refinement stage (loop ends and outputs $Spec^{Loop1}$) or emits an instruction targeted at one of the formal artifacts. A specification-targeted instruction invokes the Spec-Writer to revise S. A test-targeted instruction invokes the Test-Writer to repair or regenerate T, reformulate an obligation, or construct an additional backend probe. Figure 2 illustrates this procedure in the grey box.

The Advisor enforces the rule: a backend non-pass alone does not justify weakening a specification. The workflow revises S only when the report supports a semantic mismatch. An unfaithful test artifact instead triggers artifact repair, while a faithful but unresolved obligation triggers reformulation or further machine-grounded evaluation.

4.2 SPECAT

Behavioral-oracle refinement loop SPECAT initializes a draft spec from description and then invokes diagnostic refinement with the supplied behavioral examples only, illustrated in Figure 2. The resulting loop alternates between evaluating the current candidate and applying the Advisor's

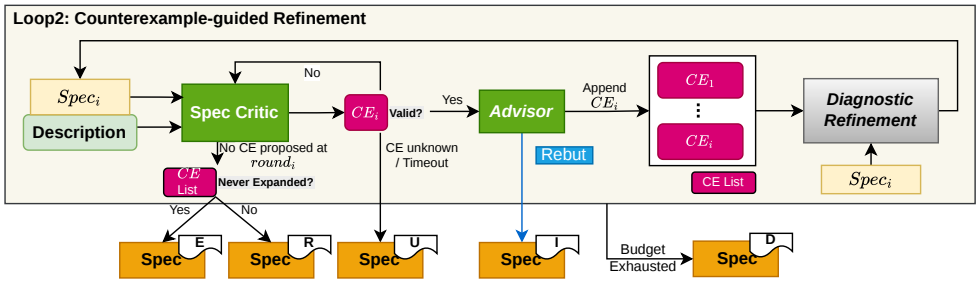

Figure 3: Overview of SPECAT's counterexample-guided refinement loop. The loop formalizes and backend-evaluates proposed counterexamples, using supported challenges to refine the candidate specification. It returns the resulting specification with an evidence-backed outcome label: ENDORSED, REPAIRED, DEFECT, UNADJUDICATED, or INCONCLUSIVE.

action until the Advisor advances the candidate or the budget is exhausted. In the latter case, SPECAT returns F indicating failure and the candidate never enters counterexample-guided refinement.

Counterexample-guided refinement loop After the provisional acceptance by the *Advisor* in the first loop, the workflow does not terminate refinement. SPECAT instantiates the abstract Spec Critic role from Section 3.2 as an independent counterexample-proposing agent, accessing the description-spec alignment. The Spec Critic agent may endorse the candidate specification, identify an ambiguity for human review, or propose a concrete counterexample: a behavior that the natural-language description and candidate specification classify differently. A proposed counterexample is not accepted on the Spec Critic's assertion alone. The Test-Writer translates it into a formal test artifact, and the evaluation backend evaluates that artifact against the current specification. The counterexample is confirmed when the backend establishes that the candidate specification classifies the behavior differently from the description-grounded expected polarity. A confirmed counterexample is returned to the Advisor as new refinement evidence. The Advisor can act on it in one of two ways: It can start another round of diagnostic refinement to repair the specification. It can rebut the counterexample, arguing that the Spec Critic misread the description; such disagreements are escalated to human review. The overview of the second loop is shown in Figure 3.

4.3 LABELS

SPECAT assigns the candidate one of the six outcomes. **Endorsed (E)**: the Spec Critic proposed no standing challenge under the allocated search budget and the candidate cleared behavioral-oracle refinement. However, it is not a proof for description–spec alignment. **Repaired (R)**: a confirmed counterexample leads to a revised specification, which the Spec Critic then endorses. **Unrepaired machine-supported challenge (D)**: a confirmed counterexample remains unrepaired when the iteration budget runs out. The specification is reported as defective, not endorsed. **Unadjudicated (U)**: the backend gives no decisive evidence for or against the counterexample, for example because of a timeout or an undischarged proof obligation. The challenge can be neither confirmed nor dismissed. **Inconclusive (I)**: the Spec Critic and the Advisor disagree after the rebuttal, or the challenge turns on an ambiguity in the description. The case is escalated for human review. **Failure (F)**: failure to clear behavioral-oracle refinement within budget.

5 EXPERIMENT RESULTS

5.1 SETUP

We evaluate SPECAT on eight benchmark–tool pairs spanning 704 units. *VerifyThisBench* (VT) (Deng et al., 2025) is evaluated in four tools including Dafny, Verus, Why3, and Frama-C/ACSL. *LeetCode(LC)-100* (Zeng et al., 2026), *MBPP* (Misu et al., 2024) and *Clover* (Sun et al., 2024) are program-suite benchmarks in Dafny. *Codeforces(CF)-100* (Agarwal et al., 2026) uses Verus and it provides a hidden suite for external validation only (Section 5.3). Each unit provides a natural-

Table 1: SPECAT Outcome of every attempted unit across all benchmarks and tools. Each unit is in one of six buckets; every share is % of Units. E: the Spec Critic raised no counterexample, or every one it raised was machine-disproved. R: a confirmed counterexample led to a corrected, re-validated specification. D: a confirmed counterexample stands unrepaired. U: a counterexample validation is unknown. I: Advisor and Spec Critic disagree on a confirmed counterexample, referred to human review. F: fails to clear behavioral-oracle refinement.

Suite	Tool	Units	E	R	D	U	I	F
CF-100	Verus	100	52 (52%)	26 (26%)	4 (4%)	3 (3%)	3 (3%)	12 (12%)
VT	Dafny	98	49 (50%)	14 (14%)	9 (9%)	6 (6%)	5 (5%)	15 (15%)
VT	Verus	98	52 (53%)	6 (6%)	14 (14%)	13 (13%)	5 (5%)	8 (8%)
VT	Why3	98	49 (50%)	5 (5%)	12 (12%)	11 (11%)	0 (0%)	21 (21%)
VT	Frama-C	98	21 (21%)	13 (13%)	24 (24%)	23 (23%)	7 (7%)	10 (10%)
LC-100	Dafny	100	71 (71%)	9 (9%)	2 (2%)	4 (4%)	0 (0%)	14 (14%)
MBPP	Dafny	50	43 (86%)	2 (4%)	1 (2%)	0 (0%)	4 (8%)	0 (0%)
Clover	Dafny	62	54 (87%)	2 (3%)	4 (6%)	1 (2%)	1 (2%)	0 (0%)
Total		**704**	**391 (56%)**	**77 (11%)**	**70 (10%)**	**61 (9%)**	**25 (4%)**	**80 (11%)**

language description and we use the examples from the benchmark as behavioral examples. Note that the hidden test cases from CF-100 are used for external validation only thus not observed by SPECAT. For each unit, we report its final category. The Spec-Writer, Test-Writer, critics, and Advisor use GPT-5.4 (OpenAI, 2026a). The Spec Critic uses GPT-5.6-Sol (OpenAI, 2026b) with an adversarial prompt. Each refinement loop has its own cost cap of $2.50 per unit, so a unit can cost at most $5.00 in total. When diagnostic refinement exceeds its cap, the unit does not enter counterexample-guided refinement. Appendix A.1 A.2 give further details on the models, tools, and benchmarks.

5.2 MAIN RESULTS

Table 1 reports the final evidence-backed disposition assigned to each attempted unit. The first loop advances 624 of 704 units (89%) to counterexample-guided refinement, declining the rest rather than emitting a specification it cannot justify (report as Failure(F) in the last column). Coverage ranges from 79–92% on the four VT formalizations of the same 98 contracts and saturates on the simpler suites (MBPP, Clover: 100%). The two competitive programming benchmarks are around 88%. Declines concentrate in concurrency, heap-pointer, and persistent-structure families.

Counterexample-guided refinement provides evidence beyond available examples. Among the 624 specs produced in the first loop, SPECAT endorses 391 specifications (63%) and repairs 77 (12%) after machine-supported counterexamples. It identifies 70 standing confirmed challenges (11%), while preserving 61 cases as unadjudicated (10%) when the available backend supplies no decisive machine evidence and referring 25 conflicting cases for human review (4%).

We manually audit whether ENDORSED and REPAIRED outcomes correspond to description-level adequacy: 60 sampled ENDORSED specifications and all 51 REPAIRED specifications outside CF-100. Fifty-eight endorsed specifications (97%) and 45 repaired specifications (88%) are adequate relative to their written descriptions and target-language interfaces; the remaining cases expose tool blind spots, repair errors, or interface limitations (Appendix I).

5.3 EXTERNAL VALIDATION ON CODEFORCES-100

Table 2 compares counterexample-guided refinement outcomes with the held-out execution oracle on CF-100; the hidden suite is evaluation-only and is never observed by SPECAT. The two sources of evidence disagree on 14 of 88 entrants. Relative to the held-out oracle, 12 are apparent false positives: the workflow endorsed or repaired a specification that still fails the hidden suite. The remaining two are apparent false negatives: counterexample-guided refinement challenged a specification that the hidden suite scores clean.

Seven of the 12 apparent false positives arise from checker conventions absent from the natural language contract, including output formatting, case sensitivity, and input serialization; one reflects a disagreement between benchmark text and its validator. Three are residual edge cases after an

Table 2: SPECAT labels versus executed hidden-suite result on 88 non-Failure units from CF-100. (columns as in Table 1). *Clean*: every sampled hidden case is met; *Failed*: at least one is violated. For 19 units the harness could not execute some case (crash, overflow, unbounded search) and no case was violated; each was settled by a mechanical re-implementation of the specification, cross-checked against every case the harness did decide. Brackets show the number of mechanically adjudicated cases.

Hidden suite	E	R	D	U	I	Total
Clean	48 (12)	18 (5)	2	2(1)	2	72 (18)
Failed	4	8	2	1(1)	1	16 (1)
Total	**52**	**26**	**4**	**3**	**3**	**88**

otherwise effective repair, and one is a genuine detection miss. The two apparent false negatives are contract-grounded challenges that the hidden oracle does not exercise. Thus, the remaining disagreements are dominated by contract–oracle convention gaps and sparse edge cases, rather than by a simple failure of intent-level reasoning. Detailed case studies are available in Appendix K.

5.4 ADDITIONAL COMPARISONS AND ABLATIONS

Appendix B compares the full SPECAT workflow with basic verifier-feedback refinement and SWE-Agent on VT-Dafny. Under the same post-hoc counterexample evaluation, SPECAT finalizes 63 of 98 units, compared with 14 for basic refinement and 25 for SWE-Agent.

Further appendix analyses characterize the workflow's components and limits. Spec Critic, Writer–Advisor, and Semantic Critic ablations show that model choice affects coverage and repair outcomes, while critic predictions remain diagnostic context rather than semantic verdicts (Appendix C, D, E). The paired counterexample-loop ablation shows that, on the same 88 CF units passing the first loop, refinement increases clean specifications from 46 to 54 and reduces hidden-row violations from 5.5% to 1.1% (Appendix F). Difficulty analyses show declining diagnostic-entry and endorsement rates on harder LeetCode and Codeforces tasks (Appendix G). Appendix H shows that diagnostic refinement is dominated by test-artifact and syntax repair, whereas counterexample-guided refinement primarily strengthens the specification in response to confirmed challenges. Finally, Appendix I reports manual-audit results for sampled specifications and cases in the remaining outcome buckets.

6 CONCLUSION

Formal verification establishes that code satisfies a specification, but it does not establish that a synthesized specification adequately formalizes a provided natural-language description. We formulate *spec-evaluation*, the problem of assessing this description–specification link from limited behavioral evidence, and propose a methodology that assesses relations among the available description, behavioral examples, candidate specification, and formal test artifacts. It combines verifier or execution outcomes with diagnostic critic assessments to distinguish possible specification defects from faulty formalizations and backend limitations. We propose SPECAT, an agentic workflow that generates, tests, and refines formal specifications from natural-language descriptions and limited behavioral examples. Across 704 benchmark–backend units, SPECAT produces not only a candidate specification but also an evidence-backed disposition: it may endorse the contract, revise it after a machine-supported challenge, retain an unrepaired challenge, or defer the case for further evidence or human review.

Limitations. SPECAT does not prove complete description-to-spec alignment; it turns a synthesized contract into an assurance artifact that can be challenged, refined, or deferred for clarification. Its evidence is bounded by the supplied description and examples, Spec Critic proposed counterexamples, and the selected verification backend. Ambiguous descriptions and requirements not expressible by the given interface or logic require human intervention, while backend or probe failures can leave challenges unadjudicated. Held-out execution evaluation is limited to Codeforces-100, and the 704 benchmark–backend units are not 704 independent tasks. In addition, in this work, we only evaluate specifications, not end-to-end verified code synthesis.

AI USE STATEMENT

In this work, we used generative AI tools for implementing the main framework, analyzing the experimental results, drafting and polishing the manuscript to improve readability, and preparing the supplementary artifact for release. We did not use AI tools to generate synthetic datasets, formulate or prove mathematical claims, perform translation, or conduct surveys or interviews; the remaining disclosure categories are not applicable to this work. We have reviewed all AI-assisted work. LLM-generated code was audited by an author. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work presents a specification-generation and evaluation methodology using public benchmarks and does not involve human subjects, personal data, or other foreseeable major ethical concerns.

REPRODUCIBILITY STATEMENT

The supplementary material includes the code artifact, full prompts and configurations, evaluated benchmarks, generated specifications, and instructions required to reproduce the SPECAT pipeline and reported results. We will publicly release the framework upon acceptance.

REFERENCES

Anmol Agarwal, Natalie Neamtu, Pranjal Aggarwal, Seungone Kim, Jannis Limperg, Cedric Flamant, Kanna Shimizu, Bryan Parno, and Sean Welleck. Verus-specgym: An agentic environment for evaluating specification autoformalization, 2026. URL https://arxiv.org/abs/2605.26457.

Anthropic. Claude fable 5 and claude mythos 5. Anthropic announcement, June 2026. URL https://www.anthropic.com/news/claude-fable-5-mythos-5. Accessed: 2026-09-20.

Anthropic. Introducing claude sonnet 5. Anthropic product release, June 2026. URL https://www.anthropic.com/news/claude-sonnet-5. Accessed: 2026-09-23.

Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: The Frama-C software analysis platform. *Communications of the ACM*, 64(8):56–68, 2021. doi: 10.1145/3470569.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp. 242–253, 2018.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pp. 383–396. Springer, 2023.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Xun Deng, Sicheng Zhong, Barış Bayazıt, Andreas Veneris, Fan Long, and Xujie Si. Verifythisbench: Generating code, specifications, and proofs all at once. *arXiv preprint arXiv:2505.19271*, 2025.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pp. 125–128. Springer, 2013. doi: 10.1007/978-3-642-37036-6_8.

Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. In *International symposium of formal methods Europe*, pp. 500–517. Springer, 2001.

John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, 1975. doi: 10.1109/TSE.1975.6312836.

Shuvendu K Lahiri. Evaluating llm-driven user-intent formalization for verification-aware languages. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 142–147. IEEE, 2024.

Shuvendu K. Lahiri. Intent formalization: A grand challenge for reliable coding in the age of ai agents, 2026. URL https://arxiv.org/abs/2603.17150.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023. doi: 10.1145/3586037.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 16–28. IEEE, 2025.

Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering*, 1(FSE):812–835, 2024. doi: 10.1145/3643763.

OpenAI. GPT-6 Astra: A new generation of intelligence. OpenAI announcement, September 2026. URL https://openai.com/index/gpt-6-astra/. Accessed: 2026-09-20.

OpenAI. Introducing GPT-5.4. OpenAI product release, March 2026a. URL https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-09-23.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. OpenAI product release, July 2026b. URL https://openai.com/index/gpt-5-6/. Accessed: 2026-09-23.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31, 2018.

Wenxian Su, Xi Wu, and Yongxin Zhao. Nl2acsl: Interactively translating natural language to ansi c specification language with large language models. *Knowledge-Based Systems*, pp. 115177, 2025.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally verified code generation. *Advances in Neural Information Processing Systems*, 38, 2026.

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

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. Autoverus: Automated proof generation for rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3454–3482, October 2025. ISSN 2475-1421. doi: 10.1145/3763174. URL http://dx.doi.org/10.1145/3763174.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. Veriequivbench: An equivalence score for ground-truth-free evaluation of formally verifiable code. In *International Conference on Learning Representations*, volume 2026, pp. 146015–146063, 2026.

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 25–37, 2020.

A Experiment Setup

A.1 Model Versions and tool details

The workflow uses GPT-5.4 (OpenAI, 2026a) for the Spec-Writer, Test-Writer, critics, and Advisor; GPT-5.6-Sol (OpenAI, 2026b) for the adversarial Spec Critic; and Claude Sonnet 5 (Anthropic, 2026) in the Spec Critic configuration comparison (Appendix E). In the ablation (Appendix D), we further use Deepseek-V4-pro and Deepseek-V4-flash (DeepSeek-AI, 2026). All models were accessed through OpenRouter.

The Verus verifier was run using version v0.2025.04.03.0f22710 (for VT Bench) and 0.2026.04.12.f1166c4 (for CF-100), while Why3 was evaluated with version v1.6.0. The Dafny toolchain ran on version v4.10.0. Frama-c tool chain ran on version 30.0.

A.2 Benchmarks

A.3 VerifyThisBench

We reorganize VerifyThisBench (Deng et al., 2025) into 98 function-level units, each with its own interface, applicable prose contract, and examples. The same 98 units are evaluated in Dafny, Verus, Why3, and Frama-C; thus, cross-tool differences reflect backend and ABI effects rather than differences in the attempted tasks.

A.4 LeetCode-100

We sample 100 units from VeriEquivBench (Zeng et al., 2026) LeetCode pool. Below shows the distributions of the sampled unit and that of the full pool.

	Easy	Medium	Hard	total
sample	33 (33%)	40 (40%)	27 (27%)	100
full pool	551 (25%)	1118 (51%)	505 (23%)	2174

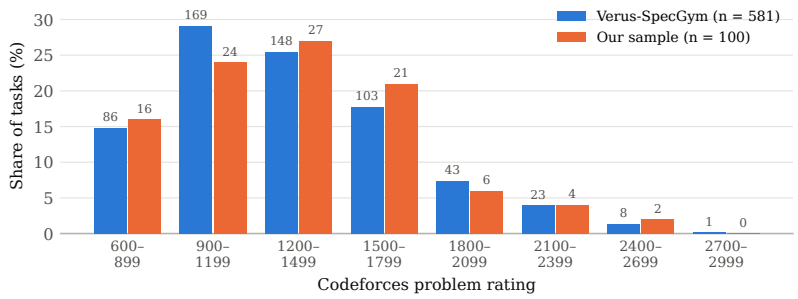

Figure 4: Codeforces problem-rating distribution: the Verus-SpecGym(Verus-SpecBench) corpus and the sample evaluated here, in 300-point bands. Percentages are of each row's total.

Arm	Filtered	E	R	D	U	I	Final / 98
Basic Refinement	33	14 (42%)	—	18 (55%)	—	1 (3%)	14 (14%)
SWE-Agent	46	25 (54%)	—	17 (37%)	—	4 (9%)	25 (26%)

Table 3: Baseline methods on VerifyThis/Dafny full suite. *Filtered* denotes specifications that satisfy filtering criterion. Percentages in columns E–I are computed over filtered samples. Labels follow definitions in Table 1. Both baselines are evaluated post-hoc using the same adversarial Spec Critic and machine-grounded counterexample-validation procedure, but do not receive counterexample-guided repair; hence R is not applicable.

A.5 CODEFORCES-100

We sample 100 tasks from Verus-SpecBench (Agarwal et al., 2026)'s 581 Codeforces tasks set. The sample is centred on the low-to-middle competitive band, with a long tail, shown in Figure 4.

Verus-SpecBench is the only suite carrying an *execution-based adequacy oracle* from upstream, which contains both positive and negative samples. A specification can therefore be measured against negatives that no component of the loop ever observed, which is what makes the holdout evaluation possible on this suite.

B BASELINE COMPARISONS

We compare SPECAT with two specification generation pipelines. Firstly, *verifier-feedback refinement* uses a single specification-writing agent that iteratively revises a candidate contract from compiler, verifier, and test feedback. In the second pipeline, we set up *SWE-Agent* (Yang et al., 2024) v1.1.0 running GPT-5.4, the pipeline's writer model. It has a shell, a file editor and the verifier, and chooses its own verification commands. A review step before submission asks it to confirm the interface, a passing dafny verify, and agreement with every seed row. The budget is $1 per unit comparable to average cost of SPECAT ($1.13) and no units exceeded the budget.

Basic Refinement loop submitted 80 and SWE-Agent submitted 98 compilable specs. We require linting, interface compliance, and agreement with all available behavioral examples under the shared formal test artifacts. We then apply the same adversarial Spec-Critic and machine-grounded counterexample-validation procedure to every baseline entrant. This evaluation is post-hoc: the baselines may be endorsed, found defective, or referred for review, but do not receive counterexample-guided repair.

On VerifyThis/Dafny, basic verifier-feedback refinement produces 33 specs passed the filter. Of these, 14 (42%) are endorsed and 18 (55%) retain a standing confirmed defect; one case is inconclusive. SWE-Agent produces 46 entrants, of which 25 (54%) are endorsed and 17 (37%) retain a confirmed defect; four cases are inconclusive. In contrast, SPECAT produces 63 finalized specifications out of 98 units (64%), comprising 49 endorsements and 14 machine-supported repairs.

	Machine outcome		
Semantic critic prediction	Proved	Executed true	Executed false
agrees with example	1,652	0	0
specification mismatch	0	22	0
prover limitation	0	373	13
Total	1,652	395	13

Table 4: Semantic Critic predictions against machine-settled behavioral-example tests on the four Dafny benchmarks. Tests with unfaithful formalizations are excluded. Execution is applied only after a prover refusal.

Table 5: Writer/Advisor model ablation on VerifyThis / Dafny (20 sampled units). Rows differ only in the writer/advisor model; the verifier and the adversarial GPT-5.6-Sol Spec-Critic are fixed. Buckets follow Table 1;

Writer/Advisor	E	R	D	U	I	F
GPT-5.4/GPT-5.4	12 (60%)	4 (20%)	1 (5%)	2 (10%)	0 (0%)	1 (5%)
DS V4-Pro/V4-Pro	11 (55%)	3 (15%)	1 (5%)	0 (0%)	1 (5%)	4 (20%)
DS V4-Flash/V4-Flash	9 (45%)	2 (10%)	1 (5%)	4 (20%)	3 (15%)	1 (5%)
DS V4-Flash/V4-Pro	12 (60%)	0 (0%)	2 (10%)	1 (5%)	1 (5%)	4 (20%)

C CAN THE SEMANTIC CRITIC DECIDE WHAT PROVER COULD NOT?

The Semantic Critic predicts whether the specification disagrees with the example or whether the non-pass reflects a prover limitation. We compare these predictions with execution-settled outcomes on the four Dafny benchmark cells. We exclude rows whose formal test artifacts were flagged as unfaithfully translated and retain only reports associated with the same specification as the executed tests, yielding 2,060 cases from 254 units.

The Semantic Critic agrees with the machine outcome on 2,025 of 2,060 rows (98.3%). This aggregate figure includes 1,652 prover-discharged rows, for which agreement is structurally expected. More relevantly, among the 408 prover-refused rows settled by execution, it correctly identifies 373 (91.4%) as prover limitations.

Its errors are nevertheless asymmetric. All 22 predictions of a specification mismatch execute to true, while all 13 execution-false rows are classified as prover limitations. Thus, the Semantic Critic is informative about the common case in which a refused proof reflects a prover limitation, but it is not a reliable detector of the rare semantic mismatches that require specification revision. We therefore use it only as non-binding diagnostic context: its predictions neither confirm counterexamples nor override proof or execution evidence.

D WRITER–ADVISOR MODEL ABLATION

We study how the models used by the Spec-Writer and Advisor affect SPECAT. We evaluate four configurations on 20-unit samples from VerifyThis/Dafny and Codeforces/Verus. The first three configurations use the same model for both the Spec-Writer and Advisor: GPT-5.4, DeepSeek V4-Pro, or DeepSeek V4-Flash. The fourth configuration uses V4-Flash as the Spec-Writer and V4-Pro as the Advisor. All other workflow components, including the evaluation backend and adversarial GPT-5.6-Sol Spec Critic, are held fixed. A run exceeding the one-hour budget is counted as not entering counterexample-guided refinement.

VerifyThis/Dafny. Table 5 shows that all configurations achieve substantial diagnostic-refinement coverage on this sample: 16–19 of 20 units enter counterexample-guided refinement. The GPT-5.4 configuration finalizes 16 units (12 endorsed and 4 repaired), compared with 14 for V4-pro, 11 for V4-flash, and 12 for the V4-flash writer with V4-pro Advisor. V4-pro has the largest endorsement fraction among its entrants (11/16), whereas V4-flash produces more unadjudicated and inconclusive

Table 6: Writer/Advisor ablation on Codeforces / Verus (20 sampled units). Rows differ only in the writer/advisor model; the verifier and the adversarial GPT-5.6-Sol Spec Critic are fixed. Buckets follow Table 1; every share is % of the sampled units. No sampled unit is UNADJUDICATED so the column is omitted. Hidden-suite columns are the executed holdout oracle: CLEAN is % of the non-F units.

Writer/Advisor	SPECAT					Hidden suite	
	E	R	D	I	F	CLEAN	Viol. rows
GPT-5.4/GPT-5.4	7 (35%)	7 (35%)	4 (20%)	0 (0%)	2 (10%)	14 (78%)	23
DS V4-Pro/V4-Pro	12 (60%)	1 (5%)	2 (10%)	0 (0%)	5 (25%)	12 (80%)	71
DS V4-Flash/V4-Flash	5 (25%)	4 (20%)	0 (0%)	0 (0%)	11 (55%)	7 (78%)	3
DS V4-Flash/V4-Pro	5 (25%)	3 (15%)	1 (5%)	1 (5%)	10 (50%)	7 (70%)	20

outcomes. These results suggest that both generation and refinement capability affect the ability to produce machine-supported specifications, although the 20-unit sample is too small to support a definitive ranking.

Codeforces/Verus. Table 6 exhibits a stronger coverage–quality trade-off. The GPT-5.4 configuration advances 18 of 20 units and finalizes 14, while V4-pro advances 15 and finalizes 13. V4-flash finalizes all 9 of its entrants, but reaches counterexample-guided refinement for only 9 of 20 units; substituting a V4-pro Advisor increases neither coverage nor the number of finalized units. Under the held-out execution oracle, the clean rates among entrants are similar across configurations (70–80%), but aggregate violation rows differ substantially. In particular, V4-pro attains 12 clean entrants but leaves 71 violation rows, illustrating that clean-unit counts and residual violation severity capture different aspects of performance.

Overall, the ablation does not identify a single model configuration that dominates every outcome. Stronger Writer–Advisor configurations generally improve coverage or finalization, whereas weaker configurations can yield high-quality specifications on the subset of units they complete.

E NEUTRAL VERUS ADVERSARIAL SPEC CRITIC

Table 7: Counter-example emission and quality per model+prompt combination over the same 88 Codeforces specs, *first judge round only* — every counterexample below targets the original spec, so the five configurations are strictly same-input and no repair loop intervenes. CONFIRMED: the probe verified the claim by execution and the advisor's grounded-concession rubric upheld it (bracket = per-counterexample precision); machine-REFUTED; UNADJ.: no machine verdict; Ruled: rebutted by the Advisor.

Model and Prompt	Challenged	CEs	CONFIRMED (prec.)	REFUTED	UNADJ.	Ruled
GPT-5.4 neutral	13	33	14 (42%)	11	5	3
GPT-5.4 adversarial	66	166	44 (27%)	86	6	30
Sonnet-5 neutral	17	27	25 (93%)	0	2	0
Sonnet-5 adversarial	32	42	40 (95%)	0	1	1
GPT-5.6-Sol adversarial	39	54	45 (83%)	1	3	5

We vary the Spec Critic's model and prompt while holding fixed the 88 Codeforces-100 specs passing the first loop, the machine-validation procedure, and the held-out execution oracle. We compare neutral and adversarial GPT-5.4, neutral and adversarial Sonnet-5, and adversarial GPT-5.6-Sol, the configuration used for all main results. Table 7 reports a first-round challenge behavior, and Table 8 reports end-to-end outcomes against the hidden suite.

Challenge rate. The prompt style mainly controls how often the Spec Critic challenges a candidate. With the model held fixed, the adversarial prompt raises the challenge rate from 15% to 75% for GPT-5.4 and from 19% to 36% for Sonnet-5. Precision, in contrast, is largely a property of the model: 27–42% for GPT-5.4, 93–95% for Sonnet-5, and 83% for adversarial GPT-5.6-Sol. More challenges

Table 8: SPECAT versus executed hidden-test result for the two adversarial judges under the identical two-phase (advisor-driven repair) loop, GPT-5.6-sol and Sonnet-5, over the same 88 Codeforces locks, as shipped and ungated (REPAIRED rows score each arm's own revised specifications). Each oracle column splits by judge model. Bucket semantics as Table 2; refusal brackets count mechanically adjudicated sets (clean / failed).

SPECAT label	Clean		Refused (adj.)		Failed		Total	
	5.6-sol	Sonnet-5	5.6-Sol	Sonnet-5	5.6-Sol	Sonnet-5	5.6-Sol	Sonnet-5
E	36	36	12 (12 / 0)	12 (12 / 0)	4	5	52	53
R	13	10	5 (5 / 0)	5 (4 / 1)	8	6	26	21
D	2	1	0	0	2	9	4	10
U	1	3	2 (1 / 1)	1 (0 / 1)	0	0	3	4
I	2	0	0	0	1	0	3	0
Total	**54**	**50**	**19 (18 / 1)**	**18 (16 / 2)**	**15**	**20**	**88**	**88**

Table 9: Codeforces: held-out execution results before and after counterexample-guided refinement. The hidden suite is Verus-Specbench's adjudication corpus of real hacks and wrong participant submissions, executed against the locked specification. Within each column *met*, *violated* and *undecided* are shares of that bucket's rows and sum to 100%; *undecided* is instrumentation loss (stack overflow, fixed-width overflow, unbounded existential search), not a specification verdict. *Clean* counts units with no violated or undecided row; *partial* counts units with no violated row but at least one undecided row; and *violating* counts units with at least one violated row.

	Precondition		Postcondition		
	valid accepted	invalid rejected	correct accepted	wrong rejected	All
Loop1 88 units: 46 clean (52%), 14 partial, 28 violating					
met	98.4	88.9	81.8	81.2	87.6
violated	1.5	9.0	8.1	5.8	5.5
undecided	0.1	2.2	10.2	12.9	7.0
pass rate of decided	*98.5*	*90.8*	*91.0*	*93.3*	*94.1*
+ counterexample-guided refinement 88 units: 54 clean (61%), 19 partial, 15 violating					
met	99.2	94.3	85.9	84.3	90.4
violated	0.8	3.5	1.6	0.1	1.1
undecided	0.1	2.2	12.5	15.6	8.5
pass rate of decided	*99.2*	*96.4*	*98.2*	*99.9*	*98.8*

alone therefore do not help: adversarial GPT-5.4 proposes 166 counterexamples, of which only 44 are confirmed.

End-to-end comparison. Of the 28 spec produced after the first loop that fail the hidden suite, adversarial GPT-5.6-Sol and adversarial Sonnet-5 each identify 23 as repaired or standing confirmed defects, and as detectors they are comparable. They differ in resolution: GPT-5.6-Sol produces more repairs (26 vs. 21) and leaves fewer shipped specifications failing the hidden suite (15 vs. 20), at the cost of three human-review referrals (vs. 0). Its endorsements remain well calibrated: after mechanical adjudication of execution refusals, 48 of 52 endorsed specifications are clean, compared with 48 of 53 for Sonnet-5.

F COUNTER-EXAMPLE GATE ABLATION

Table 9 compares the same 88 Codeforces Loop1 specifications before and after counterexample-guided refinement using the held-out Verus-SpecBench execution suite. Behavioral oracle refinement alone yields 46 clean units (52%) and 28 units with at least one violated hidden row. After counterexample-guided refinement, the number of clean units increases to 54 (61%), while the number of violating units falls to 15. Across all hidden rows, the violation rate decreases from 5.5%

to 1.1%, and the pass rate among rows with a decisive execution outcome increases from 94.1% to 98.8%.

The improvement occurs in every adequacy bucket. Violations decrease by 0.7 percentage points for valid inputs, 5.5 points for invalid inputs, 6.5 points for correct outputs, and 5.7 points for wrong outputs. The largest changes therefore occur in the postcondition buckets and in rejection of invalid inputs, consistent with the Spec Critic identifying missing constraints and edge cases not covered by diagnostic behavioral examples. The fraction of undecided rows rises modestly from 7.0% to 8.5%, reflecting limitations of the execution instrumentation rather than a specification verdict. Thus, on the one setting with an independent held-out oracle, counterexample-guided refinement improves both unit-level cleanliness and row-level agreement beyond diagnostic refinement alone.

G PASS RATE BY DIFFICULTY

Table 10: LeetCode outcomes by benchmark difficulty. *Loop1 Specs* denotes units entering counterexample-guided refinement after diagnostic refinement, reported as a percentage of *Tested*. *Endorsed* is reported as a percentage of entrants.

Difficulty	Tested	Loop1 specs	Endorsed
Easy	33	33 (100%)	29 (88%)
Medium	40	35 (88%)	29 (83%)
Hard	27	18 (67%)	13 (72%)

Table 11: Codeforces outcomes by benchmark rating. *Loop1 Specs* denotes units entering counterexample-guided refinement after diagnostic refinement, reported as a percentage of *Tested*. *Endorsed* and *Clean* are percentages of entrants. *Clean* counts entrants with no violated row under the Verus-SpecGym held-out suite.

Rating	Tested	Loop1 Specs	Endorsed	Clean
600–1199	40	39 (98%)	26 (67%)	27 (69%)
1200–1799	48	40 (83%)	22 (55%)	15 (38%)
1800+	12	9 (75%)	4 (44%)	4 (44%)

LeetCode. Table 10 groups the Dafny LeetCode-100 units by their benchmark difficulty labels. Diagnostic-refinement coverage declines monotonically with difficulty: all 33 Easy units enter counterexample-guided refinement, compared with 35 of 40 Medium units (88%) and 18 of 27 Hard units (67%). The endorsement rate among entrants also falls from 88% for Easy units to 83% for Medium units and 72% for Hard units. Thus, the workflow remains able to produce endorsed specifications for a substantial fraction of Hard tasks, but more difficult problems are more likely to retain diagnostic conflicts or unresolved obligations.

Codeforces. Table 11 groups the Verus-SpecGym sample into three rating bands. Coverage again declines with difficulty: 39 of 40 tasks in the 600–1199 band enter counterexample-guided refinement, compared with 40 of 48 tasks in the 1200–1799 band and 9 of 12 tasks rated 1800 or above. Endorsement rates similarly decrease from 67% to 55% and 44%. The held-out execution oracle shows a modest decrease in row-level pass rate, from 95.6% to 93.0% and 92.7%, respectively. Clean-unit rates are not monotone: they are 69%, 38%, and 44% across the three bands. Overall, the difficulty analysis indicates that harder tasks primarily reduce the workflow's ability to reach a diagnostically supported candidate and obtain a Spec Critic's endorsement.

H ADVISOR'S ACTIONS

Table 12 gives the action breakdown in the behavioral-oracle refinement loop (Loop 1) and counterexample-guided refinement loop (Loop 2). In Loop 1, across 2836 advisor turns over 624 locked units, the most common substantive decisions are `FIX_SYNTAX` (38%) and

Table 12: Advisor *repair* action usage by loop. Shares are within-loop: the first refinement loop covers all 624 locked units (2836 repair turns), the counterexample phase only the 201 units the gate challenged (360 repair turns), so the counts are not comparable across columns.

	Loop 1		Loop 2	
Action	Turns	%	Turns	%
HOLD_SPEC_FIX_TESTS	993	35.0	2	0.6
STRENGTHEN_SPEC	358	12.6	159	44.2
RELAX_SPEC	250	8.8	80	22.2
REFORMULATE	90	3.2	8	2.2
RESHAPE_ABI	67	2.4	0	0
FIX_SYNTAX	1077	38.0	111	30.8
DE_OVERFIT	1	0.0	0	0

HOLD_SPEC_FIX_TEST (35%). In the counterexample phase the picture inverts. Over 360 turns on the 201 units the gate challenged, that action all but disappears (0.6%) and STRENGTHEN_SPEC becomes the plurality (44%). This is the intended asymmetry: the first loop resists editing the specification to satisfy tests, while the second exists precisely to edit it in response to a confirmed counterexample.

I AUDITING THE RESULTS

We manually audited selected final specifications against their descriptions and probe artifacts, recording one verdict per case. Specifically, counterexample-guided refinement partitions the 624 candidate specifications into five outcome buckets. We exhaustively audited the two unresolved buckets: all 25 INCONCLUSIVE cases, where machine evidence did not settle the Advisor's ruling, and all 61 UNADJUDICATED cases, where no decisive machine evidence was available. We also sampled 60 of 391 ENDORSED cases to verify correctness. For REPAIRED, we audited all the non-CF cases (as CF units are scored with hidden tests), and for DEFECT we audited every outstanding counterexample.

I.1 ENDORSED: HIGH AUDIT ADEQUACY

We sampled 20 endorsements each from VerifyThis/Dafny, LeetCode/Dafny, and VerifyThis/Verus. Of the 60 cases, 58 are adequate, which still supports high endorsement precision. Two are blind spots of the instrument. In mex_sorted, a Dafny precedence error places the result constraint inside a forall, leaving the contract vacuous on an empty array; the Spec Critic reads the formula as its writer intended, and the compiler warning that flags it is discarded. In queue_pop_bottom, a by-value ABI cannot express the required post-state, so a contract that returns the correct element but leaves the queue unchanged still passes; a sibling operation that exposes a post-state parameter states both obligations.

I.2 REPAIRED

We audited all 51 REPAIRED cases that have no executed oracle; the bucket's other 26 are Codeforces and are graded by the hidden suite. 45 (88%) ship an adequate specification. The six inadequate cases fall into three categories. Three were introduced by the repair itself, each answering a correct counterexample by deleting an obligation rather than conditioning it. Two carry a defect that was already there before the repair. In both, the repair correctly fixes the counterexample it was given, but a second defect in the same specification goes untouched. One is an interface gap: the repair guesses the length of a field its C struct does not carry and thereby rejects one of the benchmark's own positive rows.

I.3 INCONCLUSIVE: MIXED AUDIT OUTCOMES

A case is INCONCLUSIVE when the machine settles the specification-side fact but the Advisor's contract-side ruling is rebutted. Among all 25 cases, 17 are adequate and 8 are inadequate. We next illustrate representative cases in which the Spec Critic and Advisor reach different rulings.

The ambiguities fall into four kinds: where an in-place effect lives under a value interface (10 cases); how wide the input domain is and what an empty input means (6); a term the description leaves open (4); and a description that contradicts its own examples or bounds (3). The remaining two are repair artifacts rather than ambiguities.

I.4 UNADJUDICATED: PROBE LIMITATIONS DOMINATE

A case is UNADJUDICATED when the Spec Critic raises a counterexample and no machine fact settles it. In 27 of the 61 cases the probe did not compile; in every one the candidate specification itself compiled and verified, and the failure lies in the probe (Verus: 13, Frama-C: 6, Why3: 6, Dafny: 2). We audited the remaining 34, whose probe compiled but was not discharged or timed out.

We identify 4 of the 34 as false alarms. Frama-C `colored_tiles` was attacked on the premise that ACSL's `\sum` is half-open, whereas it is inclusive; `merge` on a key-based ordering that its interface does not carry; Verus `abql_fairness` for requiring distinct requesters, which its description stipulates; and `search_tree_delete_min` through an inconsistency that needs a hand-assembled self-cyclic node no tree operation produces, and that the solvers never derive.

I.5 DEFECT

For the cases that fall in DEFECT, we audit the outstanding counterexamples and check whether each is genuine. We identify seven out of 70 cases as false alarms. The seven false alarms share one shape: the claim about the specification is true, and the claim about the contract is not. They fail in three ways. Three *import a convention the description never states*: that integer division truncates toward zero (Clover `avg`, whose Dafny division is Euclidean), that a non-null pointer excludes an empty array (Frama-C `mex_sorted`, where the seed rows admit the empty array and forbid the null pointer separately), or that a ghost ownership set may be set by hand to a value no constructor produces (Dafny `to_seq`). Three *contradict something the description or its property statement does say*: distinct requesters (Frama-C `abql_fairness`), well-formed inputs as the property's stated scope (Why3 `make_not`), and a queue indexed modulo its length, which presupposes a positive length (Dafny `program_p`). One *reads a garbled description literally* against the task it names (Clover `slope_search`). None concerns the specification's logic; all are errors in reading the contract, and they are the same three shapes as the Spec Critic's over-reads that the Advisor rebutted in INCONCLUSIVE. The Spec Critic's error mode does not change with the bucket; what changes is whether the Advisor caught it.

J INTERPRETING FORMAL-TEST AND CRITICS OUTCOMES

Table 13 enumerates how the system reads a formal-test outcome together with the two blind critic assessments of Section 3.2.

K CASE STUDIES

Table 2 records 14 disagreements with the executed oracle: 12 passed SPECAT (E+R) and the oracle fails (*false positives*) and two *false alarms*. We illustrate three representative cases.

Unstated checker convention (false positive; 7 of 12). **761-a**, ENDORSED, 3 of 122 hidden rows violated. The problem prints `YES` or `NO`. The decision mathematics is exact:

```
pub open spec fn yes_seq() -> Seq<char> { seq!['Y', 'E', 'S'] }
pub open spec fn no_seq()  -> Seq<char> { seq!['N', 'O'] }

pub open spec fn post_spec(in1: In1, out: Out) -> bool {
    ((out.answer == yes_seq()) || (out.answer == no_seq()))
```

```
    && ((out.answer == yes_seq()) == valid_interval_counts(in1.a, in1.b))
}
```

The three violated rows are correct answers written `"Yes"` and `"No"`: the Codeforces checker is case-insensitive, the statement shows uppercase examples only and never says so. The Spec Critic argues from the statement and had nothing to argue from. The other six cases of this kind are output whitespace and line shape, input chunking, and the benchmark's own double-escaped line endings.

Repair tail at an unsampled edge (false positive; 3 of 12). **1832-d1**, REPAIRED, 1 of 104 decided rows violated. The gate conceded a counterexample against the original lock and the accepted repair took the hidden violations from 79 to 1:

```
pub open spec fn post_spec(in1: In1, out: Out) -> bool {
    out.answer_lines.len() == 1 &&
    out.answer_lines[0].len() as i64 == in1.q &&
    answers_match(in1.a, in1.queries, out.answer_lines[0], 0usize)
}
```

The residual row is the $n{=}1$ instance, a single-element array, which the repaired `answers_match` still mis-scores. The repair fixed the general cause the Spec Critic had found; the edge it left is one no probe reached, and the re-probe of the repair, which asks the Spec Critic for a fresh round, did not reach it either.

Spec Critic right on the text, oracle data not complete (false alarm; 1 of 2). **1846-b**, DEFECT, 0 of 550 rows violated. Each tic-tac-toe row "consists of three characters"; the specification checks three valid cells and a length of at least three:

```
pub open spec fn row_has_three_cells(row: Seq<char>) -> bool {
    row.len() >= 3 &&
    valid_cell(row[0]) && valid_cell(row[1]) && valid_cell(row[2])
}
```

The Spec Critic proposed a four-character row, the probe confirmed the precondition accepts it, and the Advisor conceded: the text says exactly three. Every repair changing >= to == regressed the seed rows and was rejected, so the original ships, and against the oracle it is clean, while the "corrected" precondition would fail sixteen rows. The benchmark's serialisation double-escaped a line ending, so eight valid inputs carry a third row of five characters. The content-level check also rejects the oracle's genuinely invalid three-character rows, so it discriminates correctly in both directions. The other false alarm, 899-b, is a genuine input-format defect the hidden suite never exercises.

Formal Tests	Semantic Critic	Translation Critic	Evidence Interpretation
pass	agrees	faithful	machine evidence supports agreement with the example
pass	disagrees	faithful	conflicting evidence; inspect the critic's reasoning or assumptions
pass	—	unfaithful	masked pass; repair the artifact before using this result
non-pass	agrees	faithful	unresolved; attempt execution, reformulation, or a stronger proof probe
non-pass	disagrees	faithful	possible semantic mismatch; seek machine confirmation before revising S
non-pass	—	unfaithful	translation artifact; repair or regenerate the test

Table 13: Evidence patterns for interpreting formal-test outcomes. Both critic assessments are made without observing the engine outcome and are diagnostic only; they do not substitute for proof or execution.

L SPECAT TRACE DEMO

CF 447A asks for the index of the first collision while inserting n values into a hash table with p single-element buckets, or -1 if no collision occurs. SPECAT receives the problem statement, a fixed interface, and six behavioral examples, including $p = 10$ and $[0, 21, 53, 41, 53] \mapsto 4$. It terminates with label R: the counterexample loop repairs a specification that had passed behavioral-oracle refinement. This trace illustrates why agreement with the available examples is insufficient to establish a description-level requirement.

Behavioral-oracle refinement. After three syntax repairs the postcondition states both halves of "first": a conflict at the reported index, and no collision before it.

```
// k = out.answer - 1
(1 <= out.answer && out.answer <= in1.n)
&& (exists |j: i64| 0 <= j < k && bucket(in1, j) == bucket(in1, k))   // a conflict at k
&& (forall |a: i64, b: i64| 0 <= a < k && a < b < k                    // none before it
        ==> bucket(in1, a) != bucket(in1, b))
```

Two seed examples initially fail to prove because the verifier cannot derive the universal prefix condition from the ground facts in the test harness. The execution tier evaluates both examples successfully, so the Advisor retains the specification and instructs the Test-Writer to add explicit witnesses for the existential and universal claims. With these witnesses, all six examples are discharged by the verifier.

An erroneous relaxation. Despite this evidence, the Advisor subsequently treats the prefix-distinctness condition as overly restrictive for a permissive input/output interface and removes it.

```
(1 <= out.answer && out.answer <= in1.n)
&& (exists |j: i64| 0 <= j < k && bucket(in1, j) == bucket(in1, k))   // ANY conflict
```

This relaxation is incorrect: the problem requires the *first* collision, not any collision. The original and weakened specifications nevertheless agree on all six examples because each contains only one collision. Thus, the behavioral oracle provides no evidence against the relaxation.

Counterexample-guided repair. After behavioral-oracle refinement advances the candidate, the Spec Critic compares the statement with the weakened specification without observing the seed examples. It proposes $p = 10$, $n = 3$, and $x = [0, 10, 20]$, with output 3. The first collision occurs at insertion 2, but the weakened specification accepts output 3.

```
let exec_in1 = ExecIn1 { p: 10, n: 3, x: vec![0, 10, 20] };  // conflicts at 2 and 3
let exec_out = ExecOut { answer: 3 };                          // not the FIRST
// executed: pre = true, post = true  -- the spec accepts the forbidden output
```

The formal probe confirms that the input satisfies the precondition and that the weakened postcondition accepts the output. The Advisor therefore accepts the challenge and restores the omitted prefix-distinctness conjunct. The revised specification again discharges the six behavioral examples; the three proposed counterexamples are rejected when rechecked against it.

Observations. The behavioral examples are useful for checking concrete consequences of the description and for exposing faults in the test artifact or candidate specification. They may nevertheless fail to distinguish competing interpretations of the description. Here every supplied example has a unique collision, so the faithful contract and the weakened "any collision" contract agree on all of them. This also permits an Advisor error: it relaxes the minimality requirement despite verifier-backed agreement with every example. The counterexample loop corrects that error by supplying the missing discriminating case; its formal probe establishes that the weakened specification accepts a behavior the description forbids. The final specification has no violated row in the 74-case hidden suite, although that suite also contains no case that separates an arbitrary collision from the first collision.