SPECBRIDGE: Learning Natural-Language Formalization Plans for the Formal Specification Synthesis Task

Wenjie Zhang
National University of Singapore
wjzhang@nus.edu.sg

Yun Lin*
Shanghai Jiao Tong University
lin_yun@sjtu.edu.cn

Zining He
Shanghai Jiao Tong University
hezining@sjtu.edu.cn

Meiqi Wu
Shanghai Jiao Tong University
mickey1331@sjtu.edu.cn

Jin Song Dong
National University of Singapore
dcsdjs@nus.edu.sg

Abstract

Software requirements are usually written in natural language, which is essential for human communication but insufficient as a target for machine-checked correctness. While natural-language descriptions can state intended functionality, verification requires formal specifications with explicit types, relations, quantifiers, guards, witnesses, and edge cases. To bridge this divide, we study the *Formal Specification Synthesis* task: given a natural-language programming requirement and fixed formal signatures, synthesize a proof-assistant specification for the intended input-output relation. This task is difficult for two primary reasons. First, a large representation gap separates informal natural language from the rigorous structures needed by a proof assistant. Second, generated formal specifications are notoriously hard to evaluate, as proving full equivalence between specifications is often too complex to serve as routine feedback. To address these intertwined challenges, we propose SPECBRIDGE, a reconstruction-guided formalization framework. At its core, SPECBRIDGE introduces a Natural-Language Formalization Plan (NLFP), a semi-structured, readable intermediate representation that captures key formalization choices before decoding to a target specification language. Through reconstruction-guided pattern mining, SPECBRIDGE learns these NLFPs by reconstructing formal specifications to discover reusable patterns, ultimately transferring them to improve natural-language task generation. Furthermore, to tackle the evaluation bottleneck, we introduce a multi-layer evaluation protocol for formal-specification correctness. This protocol utilizes *LLM Specification Alignment* as an LLM-based judge, alongside *Validation Feedback* and *Verification Feedback*, which approximate "running" testcases on formal specifications by translating propositions into executable pre/post checks and proving testcase obligations in Lean. In our experiments, SPECBRIDGE outperforms the few-shot + CoT baseline by 6.1, 15.2, and 28.4 percentage points on *LLM Specification Alignment*, *Computable Validation Pass Rate*, and *Proof Validation Pass Rate*, respectively. These results show improved specification quality under our evaluation protocol.

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Natural language

```
Return only positive numbers in the list.
```

Formal specification (Lean)

```
def problem_spec
  (implementation :  List Int → List Int)
  (numbers :  List Int) :=
  let spec (result :  List Int) :=
    result.all (λ x => x > 0 ∧ x ∈ numbers) ∧
    numbers.all (λ x => x > 0 → x ∈ result) ∧
    result.all (λ x =>
      result.count x = numbers.count x);
  ∃ result, implementation numbers = result ∧
    spec result
```

Implementation (Lean)

```
def implementation
  (numbers :  List Int) :  List Int :=
  numbers.filter (λ x => x > 0)
```

Implementation (Python)

```
def implementation(l:  list):
    return [e for e in l if e > 0]
```

Figure 1: An example from the CLEVER benchmark illustrating the relation shape that the target formal specification encodes.

1 Introduction

Software engineering naturally begins with requirements, which are typically written in natural language. While these informal descriptions are indispensable for communicating intended functionality among humans, they are fundamentally insufficient for machine-checked program verification. Natural language is inherently partial, implicit, or ambiguous; it lacks the rigorous contract required by a verifier or proof assistant. To bridge this divide and verify that a program satisfies its intended behavior, we require a *formal specification*, a mathematically precise description of the allowed input-output relation constructed with explicit types, relations, quantifiers, guards, witnesses, and constraints.

In this paper, we study the *Formal Specification Synthesis* task: given a natural-language programming task and fixed formal signatures, synthesize a formal specification that precisely captures the intended input-output relation. This task is a critical bottleneck in verified-code generation. After all, a verification proof is only as meaningful as the specification it targets; the formula must faithfully capture the intended behavior before any proof can certify conformance. While this task can be instantiated in various machine-checkable languages, we ground our study in Lean, a powerful proof-assistant language based on dependent type theory. Lean's ability to express typed programs, first-order logical constraints, and inductive predicates makes it an ideal and practical target for formal-specification synthesis. As illustrated in Figure 1, the specification serves as the rigorous central contract, ensuring that the concrete implementation aligns with the original natural-language intent.

Solving the *Formal Specification Synthesis* task requires overcoming two challenges. The first is a *representation gap* during generation. Direct natural-language-to-formal-specification models must leap directly from human-centric prose to valid target-language code, simultaneously deducing the formal predicate structure, quantifier scopes, and edge cases in a single step. The second challenge is a *feedback bottleneck*. Evaluating generated specifications is notoriously difficult because checking full mathematical equivalence between two formal propositions is highly complex. Consequently, models suffer from sparse feedback: when an equivalence check fails, it is often unclear whether the error stems from a deep semantic misunderstanding, a trivial syntax flaw, or merely a divergence in benchmark-specific relational style.

To address these challenges, we propose SPECBRIDGE, a reconstruction-guided formalization framework. At the core of our approach is the introduction of a Natural-Language Formalization Plan (NLFP). An NLFP acts as a semi-structured, readable bridge that explicitly records key formalization choices, such as relation shape, guard placement, and collection behavior, before any decoding to the target language occurs. Rather than forcing a model to jump directly from natural language to Lean, SPECBRIDGE first constructs this formally grounded plan and then translates it into a precise specification. To learn these NLFPs, SPECBRIDGE refines prompts through two complementary paths in each training round: an auto-encoding path (formal specification $\rightarrow$ NLFP $\rightarrow$ formal specification) that exposes reusable relational patterns, and a task-generation path (natural language $\rightarrow$ NLFP $\rightarrow$ formal specification) that applies them to requirements. Feedback from both paths is aggregated before the shared prompt guidance is updated.

Furthermore, to overcome the feedback bottleneck, we introduce a multi-layer evaluation protocol that does not rely on strict equivalence proofs. We utilize *LLM Specification Alignment* for high-level semantic judgements, alongside two testcase-driven feedback channels: *Validation Feedback* and *Verification Feedback*. Because Lean propositions are not directly executable, *Validation Feedback* translates the generated specifications into executable Python pre/post-condition checks, while *Verification Feedback* leverages Lean itself to formally prove whether the generated specification accepts or rejects concrete testcase behaviors. Together, they provide fast, proof-backed behavioral feedback.

We evaluate SPECBRIDGE on the CLEVER benchmark [1]. While CLEVER proposes a broad verified-code pipeline, our focus remains squarely on the foundational step of synthesizing the formal specification from natural language. Compared with a direct few-shot + chain-of-thought (CoT) baseline, SPECBRIDGE improves *LLM Specification Alignment*, *Computable Validation Pass Rate*, and *Proof Validation Pass Rate* by 6.1, 15.2, and 28.4 percentage points, respectively. Ablation studies indicate that the reconstruction path, NLFP prompt refinement, and testcase-based feedback each contribute to these improvements.

In summary, this paper makes the following contributions:

- A reconstruction-guided formalization framework (SPECBRIDGE) that uses NLFPs to effectively bridge the representation gap between natural-language task descriptions and formal specifications.
- A multi-layer formal-specification evaluation protocol combining *LLM Specification Alignment*, *Validation Feedback*, and *Verification Feedback* to provide robust, test-driven feedback.
- An implementation of SPECBRIDGE and rigorous empirical evaluations demonstrating substantial, consistent gains over few-shot + CoT baselines.

Source code, demonstrations, datasets, results, and supplementary materials are available on the project site: `https://sites.google.com/view/specbridge`.

2 Motivating Example

To make our motivation concrete, consider the CLEVER task introduced in Figure 1: given a list of integers `numbers`, return the positive elements. Although the natural-language requirement is one short sentence, its formalization must choose a concrete input-output relation. Figure 2 brings the requirement, learned-prompt guidance, formalization plan, Lean relation, and behavioral probes together in one view.

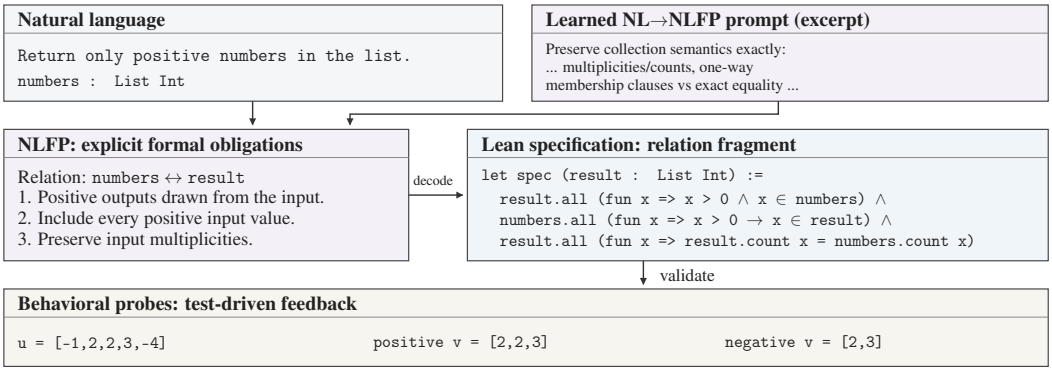


Figure 2: A prompt-guided motivating example. Natural-language input and an excerpt from a learned NL-to-NLFP prompt jointly guide the illustrated plan. The prompt calls for preserving collection semantics, including multiplicities/counts rather than relying on membership alone. The plan records positivity, coverage, and duplicate preservation before Lean decoding; behavioral probes then test the resulting relation. This schematic illustration is not a verbatim model trace or a proof of specification equivalence.

This example exposes two bottlenecks. The *representation gap* arises because direct natural-language-to-Lean generation must infer these obligations, choose guard and quantifier scopes, and emit valid

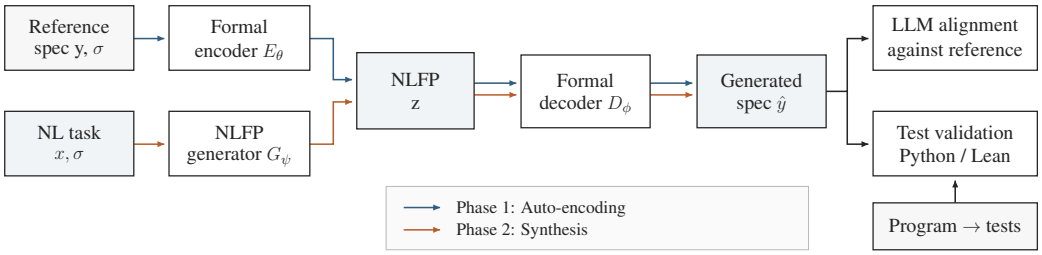


Figure 3: SPECBRIDGE training overview. Blue arrows show the auto-encoding path from a reference specification through an NLFP to a reconstructed specification. Orange arrows show the task-generation path from natural language through an NLFP to a generated specification. Both paths share the formal decoder. Reference specifications are used only during training and evaluation.

target-language syntax in one step. The *feedback bottleneck* arises because proving equivalence between complete specifications is difficult, even when concrete behaviors can already expose defects. For the displayed input, the correct output preserves both copies of `2`, whereas the corrupted output drops one; a positivity-only or membership-only predicate would miss this distinction.

SPECBRIDGE addresses the representation gap by making the relation choices explicit in an NLFP before Lean decoding. It addresses the feedback bottleneck through executable checks and Lean proof obligations on positive and negative behaviors. These signals refine reusable generator and decoder prompts for later rounds rather than directly editing the displayed task-specific NLFP. They remain sampled evidence rather than a proof that the generated relation captures every aspect of the natural-language intent.

3 Method

To address the representation gap and feedback bottleneck highlighted in Section 2, we propose SPECBRIDGE, a reconstruction-guided formalization framework. This section first defines the specification synthesis task, then explains how SPECBRIDGE learns Natural-Language Formalization Plans (NLFPs), and finally describes the test-driven feedback channels used for prompt refinement and evaluation.

3.1 Problem Formulation

We define *Formal Specification Synthesis* as the problem of generating a formal specification y from a natural-language task input x. In this paper, we instantiate the task in Lean, aligning with the first stage of the CLEVER benchmark: synthesizing a Lean `problem_spec` given a natural-language task description and fixed formal signatures.

Each training instance is $(x_i, y_i, \sigma_i, c_i, T_i)$, where x_i is the natural-language task, y_i is the benchmark formal specification, σ_i is the formal signature and local declaration context, c_i is the canonical Python program, and T_i is a set of input-output testcases. The objective is to learn a mapping $(x, \sigma) \rightarrow \hat{y}$ that produces a high-quality formal specification for unseen tasks. At inference time, the forward generator receives only x and σ; benchmark specifications, canonical programs, and testcases are used only to construct training feedback and held-out evaluation signals.

3.2 Reconstruction-Guided NLFP Pattern Mining

Directly mapping human-centric prose to rigorous Lean code exacerbates the representation gap. SPECBRIDGE bridges this gap by decoupling formalization planning from target-language decoding through a Natural-Language Formalization Plan (NLFP), denoted as z. An NLFP is a semi-structured, readable text artifact that stores the formalization choices direct generation often misses, such as relation shapes, guard placements, witness choices, and collection semantics. It does not introduce external information; instead, it preserves relational patterns learned from formal reconstruction before the model emits Lean syntax.

As illustrated in Figure 3, each round evaluates two complementary paths:

1. **Auto-Encoding Path (Reconstruction):** We start from the benchmark formal specifications to learn which structures must survive a round trip. An encoder prompt E_θ maps a Lean formal specification into an NLFP, while a decoder prompt D_ϕ reconstructs it back into Lean: $z_i = E_\theta(y_i, \sigma_i)$ and $\hat{y}_i^{\mathrm{AE}} = D_\phi(z_i, \sigma_i)$. If $\hat{y}_i^{\mathrm{AE}}$ closely aligns with y_i, it confirms the NLFP successfully captured the necessary formal skeleton.
2. **Task-Generation Path (Transfer):** A generator prompt G_ψ maps natural-language task inputs into an NLFP using learned formalization guidance, without receiving y_i as a forward input. The decoder D_ϕ renders the final specification: $\hat{z}_i = G_\psi(x_i, \sigma_i)$ and $\hat{y}_i^{\mathrm{task}} = D_\phi(\hat{z}_i, \sigma_i)$.

3.3 Iterative Prompt Refinement

Because intermediate NLFPs are text-based, SPECBRIDGE learns by updating prompts rather than by backpropagation. The update step relies on LLM reflection over structured failures.

We define the learnable prompt bundle as $P = (E_\theta, G_\psi, D_\phi)$. Algorithm 1 gives the procedure for an arbitrary number N of refinement rounds; our experiments use $N = 5$. Within a round, both paths process every training example using frozen skill pools; feedback is aggregated before the three pools are refined. The encoder's learned formalization choices reach the generator through shared prompt guidance across rounds, not a separately passed schema object at inference time. NLFPs are readable, semi-structured text rather than instances of a fixed grammar; their adequacy is assessed through the decoded Lean specification.

Algorithm 1 Reconstruction-guided refinement of the skill pools

Require: Training set $\mathcal{D} = \{(x_i, y_i, \sigma_i, c_i, T_i)\}$; initial pools $P_0 = (E_0, G_0, D_0)$; number of rounds N

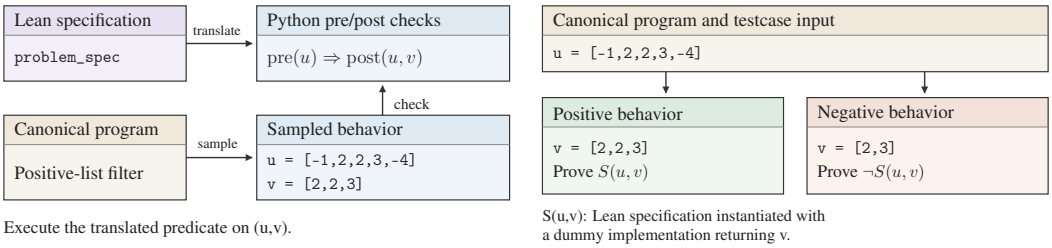
for round $= 1, \ldots, N$ **do**
  // Forward pass with frozen pools
  for $i = 1, \ldots, |\mathcal{D}|$ **do**
    $(z_i^{\mathrm{AE}}, \hat{y}_i^{\mathrm{AE}}) \leftarrow \mathrm{Reconstruct}(y_i, \sigma_i; P_{\mathrm{round}-1})$
    $(z_i^{\mathrm{task}}, \hat{y}_i^{\mathrm{task}}) \leftarrow \mathrm{Formalize}(x_i, \sigma_i; P_{\mathrm{round}-1})$
  end for
  // Aggregate feedback before any pool update
  $\mathcal{E}_{\mathrm{round}} \leftarrow \mathrm{Aggregate}\big(\{\mathrm{Evaluate}(i, \hat{y}_i^{\mathrm{AE}}, \hat{y}_i^{\mathrm{task}})\}_{i=1}^{|\mathcal{D}|}\big)$
  // Attribute errors and refine each skill pool
  for all $S \in \{E, G, D\}$ **do**
    $S_{\mathrm{round}} \leftarrow \mathrm{Refine}(S_{\mathrm{round}-1}, \mathrm{Attribute}(\mathcal{E}_{\mathrm{round}}, S))$
  end for
end for
return $P_N = (E_N, G_N, D_N)$

3.4 Multi-Layer Evaluation and Feedback

The same signals that guide prompt refinement also define our evaluation protocol. Because strict specification equivalence is difficult to prove routinely, SPECBRIDGE uses three complementary channels as training feedback and as final diagnostic metrics.

LLM Specification Alignment. An LLM judge compares a candidate specification against the task, benchmark specification, and canonical Python behavior. *LLM Specification Alignment* reports the fraction of held-out problems judged aligned with the ground-truth intent; it is an absolute alignment score, not a pairwise win rate against the baseline.

Validation Feedback (Python Executable Checks). *Validation Feedback* uses the canonical Python program c_i as a behavioral oracle. An LLM translates the generated Lean `problem_spec` into executable Python pre/post-condition checks over testcases $T_i = \{(u_{ij}, v_{ij})\}$. A testcase passes when the precondition is false or the postcondition holds. *Computable Validation Pass Rate* reports the fraction of passed checks, providing broad but non-certifying feedback on guards, edge cases, and collection behavior.



(a) Validation Feedback

(b) Verification Feedback

Figure 4: Two testcase feedback channels for the same CLEVER task. The executable channel translates the candidate specification into Python checks; the Lean channel proves acceptance of a positive behavior and rejection of a corrupted output.

Verification Feedback (Lean Proof Obligations). *Verification Feedback* keeps behavioral checks inside Lean. We construct positive testcases T_i^+, where $v = c_i(u)$, and negative testcases T_i^-, where $\tilde{v}$ is intentionally corrupted. For each concrete behavior, a dummy implementation returns the testcase output, and Lean attempts to prove that the generated specification accepts the positive behavior or rejects the negative one. *Proof Validation Pass Rate* reports the fraction proved successfully. Failure may reflect proof-search limits rather than an incorrect specification, while success gives kernel-checked evidence at that concrete behavioral boundary.

Figure 4 shows the two feedback channels on the same positive-list task: executable validation checks the Python translation against sampled behavior, while verification constructs Lean obligations for both valid and corrupted outputs.

3.5 Structured Feedback as the Training Objective

The refinement objective is the round-level structured report $\mathcal{E}_{\text{round}}$, rather than a differentiable loss. It aggregates five signals: Lean compilation errors; semantic disagreement with the task or reference; executable pre/post-check failures; disagreement with canonical-program behavior; and positive or negative Lean obligations that were not proved. The reflection LLM attributes recurring failures to the relevant skill pool and revises its prompt instructions, for example by preserving a helper operator or choosing an index rather than a value as a witness. A failed proof is diagnostic rather than definitive. The updated pools are reused in the next round.

4 Experiments

We organize the evaluation around three research questions:

- **RQ1 (Baseline Comparison):** How does SPECBRIDGE compare with direct generation, generic planning, reranking, and direct feedback refinement on CLEVER?
- **RQ2 (Ablation):** Which components of SPECBRIDGE contribute to specification quality?
- **RQ3 (Generalization):** Does performance carry across model backends, prompt sources, and datasets?

4.1 Experimental Setup

We evaluate the primary five-fold comparison on CLEVER and cross-dataset transfer on the full 189-task VERINA dataset [1, 2]. Each CLEVER instance provides the original natural-language task, formal signatures, a reference Lean `problem_spec`, supporting definitions, and a canonical Python implementation. VERINA is used for the separate frozen-prompt transfer experiment under RQ3.

For the CLEVER main experiment, we use five-fold cross-validation to prevent data leakage: in each isolated run, NLFP guidance and prompt bundles are learned only from that fold's training split and evaluated on its held-out split. The primary full-benchmark SPECBRIDGE results use the round-5 prompt bundle. Although intermediate diagnostics suggest that round 2 is often a strong cost-performance checkpoint, the final round-5 evaluation remains above the baseline across all

reported metrics. The three primary metrics, *LLM Specification Alignment*, *Computable Validation Pass Rate*, and *Proof Validation Pass Rate*, aggregate these held-out results. Supplementary controlled comparisons, transfer experiments, and reliability diagnostics are reported separately, with their reporting scope documented in Appendix F. Additional implementation details are provided in Appendix H.

All tabled rates are reported as proportions on $[0, 1]$, rounded to two decimal places; counts and monetary costs retain their original units.

4.2 Compared Systems

We use the CLEVER-style few-shot + CoT protocol as the primary directly applicable baseline under the requirement-and-signature input interface. RQ1 additionally compares generic planning, validation-based reranking, and direct Lean feedback refinement. RQ2 removes components from SPECBRIDGE, including a controlled variant without forward feedback. Systems that assume implementation access or an already specified verification target are discussed conceptually rather than treated as interchangeable baselines.

- **Few-shot + CoT (Baseline):** The model decodes the original natural-language task and formal signatures directly into a Lean formal specification using a few-shot prompt with chain-of-thought reasoning.
- **Free-form Plan-then-Spec (Baseline):** The model writes an unconstrained natural-language plan before producing the Lean specification.
- **Best-of-5 Validation Reranking (Baseline):** Five directly generated Lean specifications are ranked using the validation signals, and the highest-ranked candidate is selected.
- **Direct Lean Feedback Refinement (Baseline):** Validation and verification feedback refine the Lean specification directly, without an NLFP.
- **SPECBRIDGE (Ours):** Our deployable reconstruction-guided formalization framework, which first maps the natural-language task into an explicit NLFP, and then decodes that intermediate plan into a Lean formal specification.

4.3 Results and Analysis

RQ1: Comparison with baselines. Table 1 compares SPECBRIDGE with direct generation, free-form planning, validation-based reranking, and direct Lean feedback refinement on CLEVER. In the primary five-fold comparison, *Proof Validation Pass Rate* rises from 58.2% for few-shot + CoT to 86.6% for SPECBRIDGE. API-cost details for these two primary configurations appear in Appendix E.

Table 1: RQ1. Baseline comparison on CLEVER dataset.

Method	LLM Specification Alignment	Computable Validation Pass Rate	Proof Validation Pass Rate
Few-shot + CoT	0.61	0.76	0.58
Free-form Plan-then-Spec	0.65	0.85	0.76
Best-of-5 Validation Reranking	0.60	0.83	0.50
Direct Lean Feedback Refinement	0.54	0.85	0.80
SPECBRIDGE	**0.67**	**0.91**	**0.87**

Free-form planning improves on direct generation but trails SPECBRIDGE in computable and proof validation. Direct Lean feedback improves proof validation over direct generation, while Best-of-5 reranking does not explain the gain. Removing forward feedback from SPECBRIDGE is analyzed under RQ2 as an ablation, not an RQ1 baseline.

Finding for RQ1. SPECBRIDGE achieves the highest alignment, computable-validation, and proof-validation rates among the compared methods.

RQ2: Component ablation. To isolate the drivers of these gains, we ablate the reconstruction auto-encoding path, NLFP prompt refinement, Validation Feedback, and canonical-program semantic

feedback. Table 2 shows that each component contributes on at least one metric. Removing reconstruction most harms *LLM Specification Alignment*, while removing NLFP prompt refinement most harms *Proof Validation Pass Rate*. A complementary ablation removes forward feedback from the NLFP pipeline (Table 9). A fuller interpretation appears in Appendix E.

Table 2: RQ2. Component ablation on CLEVER.

Method	LLM Spec. Alignment	Computable Pass Rate	Proof Pass Rate
Full SPECBRIDGE	**0.67**	**0.91**	**0.87**
w/o reconstruction auto-encoding	0.47	0.89	0.85
w/o NLFP prompt refinement	0.61	0.90	0.63
w/o Validation Feedback	0.49	0.90	0.82
w/o canonical-program semantic feedback	0.52	0.90	0.82

Finding for RQ2. Reconstruction mainly supports semantic alignment, while iterative refinement is crucial for producing specifications that Lean can certify on concrete proof obligations.

RQ3: Generalization. Across model backends. To verify that the framework is not tied to one model architecture, we evaluate it across three LLM backends, using the same backend for prediction and prompt refinement. As shown in Table 3, SPECBRIDGE improves over few-shot + CoT for all evaluated models. The dominant gain varies by backend, suggesting that semantic alignment, executable behavior, and proof-backed certifiability capture complementary dimensions of specification quality.

Table 3: RQ3. Performance across LLM backends, with prompts refined on the same backend used for inference.

Model	Method	LLM Spec. Alignment	Computable Pass Rate	Proof Pass Rate
`gpt-5.4`	Few-shot + CoT	0.61	0.76	0.58
`gpt-5.4`	SPECBRIDGE	**0.67**	**0.91**	**0.87**
`deepseek-v3.2`	Few-shot + CoT	0.24	0.78	0.67
`deepseek-v3.2`	SPECBRIDGE	**0.42**	**0.91**	**0.69**
`gemini-3.1-pro-preview`	Few-shot + CoT	0.88	0.89	0.79
`gemini-3.1-pro-preview`	SPECBRIDGE	**0.90**	**0.92**	**0.82**

Across the three evaluated backends, SPECBRIDGE improves over few-shot + CoT.

Across prompt sources. Unlike the preceding backend comparison, this experiment holds the inference backend fixed at `deepseek-v3.2`. The controlled comparison changes the source of the refined prompt between two SPECBRIDGE variants; the few-shot + CoT row is a reference baseline. Table 4 shows that GPT-5.4-refined guidance raises alignment from 0.42 to 0.58 and proof validation from 0.69 to 0.78 relative to DeepSeek self-refinement, while computable validation remains 0.91. This transfers prompt instructions, not model parameters.

Table 4: RQ3. Prompt-source transfer on CLEVER; DeepSeek-v3.2 is the inference backend in every row.

Configuration	LLM Specification Alignment	Computable Validation Pass Rate	Proof Validation Pass Rate
Few-shot + CoT	0.24	0.78	0.67
SPECBRIDGE: DeepSeek-refined	0.42	**0.91**	0.69
SPECBRIDGE: GPT-5.4-refined	**0.58**	**0.91**	**0.78**

Across datasets. This experiment holds the CLEVER-trained prompt bundle fixed and changes the evaluation dataset to all 189 VERINA tasks [2]. No VERINA reference specification or validation

feedback is used for prompt refinement. With the same inference backend and tasks for both methods, Table 5 shows improvements on all three metrics. The result concerns transfer under this task interface; stateful and multi-function specifications remain outside the evaluated setting. Protocol details appear in Appendix F.

Table 5: RQ3. Frozen-prompt transfer from CLEVER to the full VERINA dataset (189 tasks).

Method	LLM Specification Alignment	Computable Validation Pass Rate	Proof Validation Pass Rate
Few-shot + CoT	0.56	0.61	0.83
CLEVER-trained SPECBRIDGE	**0.89**	**0.74**	**0.94**

Finding for RQ3. The model and prompt-source comparisons show gains beyond the primary backend, and the frozen-prompt experiment suggests transfer to VERINA under the reported protocol.

5 Discussion

Formalization Choices Captured by NLFPs. NLFPs make formalization choices explicit before Lean decoding. They record type and helper anchors, collection semantics, guard and witness scope, and Lean scaffolding such as local names and tuple destructuring. These choices explain why a short requirement can require several formal clauses and why a plausible direct translation may still be incomplete. Appendix A gives concrete examples.

Training Dynamics of Prompt Refinement. Prompt refinement improves most in the first two rounds and then stabilizes. The `NL -> NLFP` prompt grows most because it must infer choices from raw natural language. Figure 8 and the round-by-round analysis appear in Appendix E.

The Complementarity of Validation and Verification Feedback. Validation Feedback and Verification Feedback play different roles. *Validation Feedback* is broad and fast because it translates a generated Lean predicate into executable Python checks over many canonical-program testcases, but we do not treat this translated checker as a soundness certificate. The formal signal is *Verification Feedback*: each successful obligation is kernel-checked inside Lean, so a high proof-validation rate shows that the generated specifications are easier to certify at concrete behavioral boundaries even if executable translation has blind spots. Cross-channel consistency and manual auditing provide additional safeguards, but cannot exclude correlated errors or untested boundary defects (Appendix F.1). Their complementarity is illustrated by the string XOR case analyzed in Appendix D: executable checks pass, but proof validation exposes an invalid Lean string-index representation.

Evaluator reliability. Lean checks a constructed proposition, but cannot establish that an LLM faithfully translated the intended requirement. We therefore assess cross-channel consistency, artifact-level translation and obligation fidelity, and sensitivity to testcase coverage separately. Tables 6 and 7 summarize the CLEVER consistency and coverage experiments. The artifact audit distinguishes faithful evaluator construction from candidate-level specification correctness; protocol details appear in Appendix F.1.

Table 6: Cross-channel consistency on CLEVER (161 tasks; 20 testcases per task).

Method	Pass rate
Few-shot + CoT	0.45
Direct Lean feedback	0.70
SPECBRIDGE	**0.74**

Table 7: Proof-validation coverage sensitivity on 161 CLEVER tasks with fixed specifications.

Testcases per task	Proof pass rate
10 positive + 10 negative	0.89
100 positive + 100 negative	0.85

Failure modes. We analyze failed CLEVER specifications by their primary error mechanism (Table 11 in Appendix F.3). A misplaced guard can make an implication vacuously true; an underconstrained output relation can accept values unrelated to the input. Collection errors can lose multiplicity or order, while overconstraints reject otherwise valid outputs. These mechanisms suggest distinct prompt fixes: preserve guard scope, state the input–output relation, and specify collection semantics explicitly.

Limitations. Our evaluation covers CLEVER and the full VERINA dataset, but remains focused on small, single-function input-output specifications. Frozen-prompt transfer to VERINA does not establish performance on multi-function APIs, mutable state, heap ownership, concurrency, or temporal properties (Appendix F). An NLFP makes a chosen interpretation explicit; it cannot recover intent absent from the requirement. For example, "return positive numbers" alone does not decide whether output order and duplicates must be preserved. Training references, signatures, and benchmark conventions can guide that choice, but genuinely competing interpretations require clarification from the requester. Behavioral feedback also relies on a canonical executable program; abstract domains would need another oracle. Lean validates a successful proof of the constructed obligation, while LLM-translated checks and generated obligations can still miss intended semantics or boundary cases.

6 Related Work

Natural-language formalization. Lang2LTL grounds natural-language commands in temporal logic, while nl2spec links temporal subformulas to source phrases for interactive correction [3, 4]. Our target is a typed, function-level Lean `problem_spec`. Here, guard scope, witness choice, and collection semantics must be made explicit; NLFPs record these choices before Lean decoding.

Program specifications from language and code. The closest requirement-centered work includes nl2postcond, which generates method postconditions from natural language, and NL2Contract, which also infers preconditions [5, 6]. AutoSpec uses static analysis and verification feedback; SpecGen uses program comprehension, specification mutation, and verifiability-based selection [7, 8]. These systems differ in their input contracts and target languages. SPECBRIDGE generates from a requirement and fixed Lean signature. Reference specifications and canonical implementations support learning and evaluation, but are not task-specific inputs to the deployed generator. RQ1 therefore compares methods under the same generation interface.

Intermediate reasoning and feedback. Draft, Sketch, and Prove uses informal proof sketches to guide proof search; AlphaVerus and Baldur improve proof-oriented artifacts [9–11]. Our NLFP instead plans the specification's semantics. Reconstruction and task generation expose complementary errors, and executable and Lean feedback refine reusable prompts before final task-specific generation. Appendix G provides further comparisons.

7 Conclusion

We introduced SPECBRIDGE for *Formal Specification Synthesis* from natural-language requirements and fixed Lean signatures. Its reconstruction-guided Natural-Language Formalization Plans make guard, witness, and collection choices explicit before Lean decoding. Prompt refinement combines semantic judgement with executable checks and Lean proof obligations. Across our evaluated tasks, this approach improves specification quality, with the largest gains in kernel-checked proof validation. The remaining failures and ambiguity limits show where explicit plans still need better requirement clarification and boundary-case feedback.

Acknowledgments and Disclosure of Funding

This research is supported in part by the National Natural Science Foundation of China (62572300), the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office, and partially by HUAWEI's AI Hundred Schools Program using the Ascend AI technology stack. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore. We are grateful to Prof. Zhenjiang Hu for his insightful suggestions on this work.

References

[1] 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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2505.13938.

[2] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *2nd AI for Math Workshop@ ICML 2025*, 2025. URL https://arxiv.org/abs/2505.23135.

[3] Jason Xinyu Liu, Ziyi Yang, Benjamin Schornstein, Sam Liang, Ifrah Idrees, Stefanie Tellex, and Ankit Shah. Lang2ltl: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*, 2023. URL https://arxiv.org/abs/2302.11649v1.

[4] 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*, pages 383–396. Springer, 2023. URL https://arxiv.org/abs/2303.04864.

[5] 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. URL https://arxiv.org/abs/2310.01831.

[6] Cedric Richter and Heike Wehrheim. Beyond postconditions: Can large language models infer formal contracts for automatic software verification? *arXiv preprint arXiv:2510.12702*, 2025. URL https://arxiv.org/abs/2510.12702.

[7] 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*, pages 302–328. Springer, 2024. URL https://arxiv.org/abs/2404.00762.

[8] 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)*, pages 16–28. IEEE, 2025. URL https://arxiv.org/abs/2401.08807.

[9] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.12283.

[10] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement. In *International Conference on Machine Learning*, pages 587–615. PMLR, 2025. URL https://arxiv.org/abs/2412.06176.

[11] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023. URL https://arxiv.org/abs/2303.04910.

[12] Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 658–666, 2005. URL https://arxiv.org/abs/1207.1420.

[13] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 27(2):99–123, 2001. URL https://doi.org/10.1109/32.908957.

[14] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. In *International symposium of formal methods Europe*, pages 500–517. Springer, 2001. URL https://link.springer.com/chapter/10.1007/3-540-45251-6_29.

[15] Amir Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977. URL https://doi.org/10.1109/SFCS.1977.32.

[16] Edmund M. Clarke and E. Allen Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982. URL https://doi.org/10.1016/0167-6423(83)90017-5.

[17] Thanh Le-Cong, Bach Le, and Toby Murray. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 21991–22014, 2025. URL https://arxiv.org/abs/2503.04779.

[18] Md Rakib Hossain Misu, Iris Ma, and Cristina V Lopes. Veriact: Beyond verifiability–agentic synthesis of correct and complete formal specifications. *arXiv preprint arXiv:2604.00280*, 2026. URL https://arxiv.org/abs/2604.00280.

A Additional NLFP Learning Examples

Building upon the discussion in Section 5, this section provides qualitative examples showing how NLFPs turn underspecified natural language into explicit formalization decisions before decoding.

What NLFPs capture. The primary utility of an NLFP is to freeze formalization decisions that are often omitted from natural-language requirements and difficult to recover after a model has already committed to a Lean predicate. In particular, NLFPs bind high-level task meaning to strict target-language representations, record collection behavior that distinguishes plausible but underconstrained specifications from behaviorally precise ones, and preserve the structural shape of propositions through guard placement, witness choices, and local scaffold. Prompt-refinement feedback further teaches compile-relevant Lean surface forms, such as tuple destructuring and indexed-variable conventions, so the decoder learns more than a high-level paraphrase of the task.

Type-theoretic shape. A fundamental capability of the NLFP is binding high-level task intent to the precise Lean types and eliminators utilized by the benchmark. Figure 5 illustrates a string-processing task where the NLFP explicitly records that the input is a `String`, the output is a `List Nat`, and the result must align with a list of space-separated groups. This represents far more than a natural-language paraphrase: it structurally commits to list length equality, per-index access, and helper-level anchors (e.g., `count_max_paren_depth`). The example demonstrates that NLFPs expose exactly the representational choices that the decoder must preserve: tokenization, result shape, index alignment, and helper-level semantics.

Collection semantics. Many CLEVER tasks appear deceptively brief in English yet exhibit profound differences in their formal collection semantics. For instance, "Return only positive numbers" could theoretically be encoded as set-like membership, multiset constraints, an order-preserving filter, or weaker positivity-only checks. The NLFP in Figure 6 explicitly resolves this ambiguity by stating the same relation emphasized in the motivating example: output values must be positive and drawn from the input, every positive input value must appear in the output, and multiplicity must be preserved through counts. This example highlights a critical learned skill: once the NLFP locks in the intended list relation, the decoder can realize the benchmark's membership-and-count structure rather than a weaker membership-only condition.

Guard and witness scope. Tasks heavy in Boolean logic and witness generation particularly benefit from explicit structural planning. For a task identifying "close elements," the training NLFP documents that the Boolean result is governed by two distinct *indices* (not merely two values) and that the return relation must be guarded by `numbers.length > 1`. As a result, the decoded Lean accurately preserves both the index witnesses and the guarded Boolean branch:

NLFP excerpt.

```
Input: paren_string : String.
Result type: List Nat.
The result is a list of natural numbers. Each entry corresponds to one parsed group
from the input string, in the same left-to-right order as the groups appear after
the intended space-based tokenization.

Define groups as the exact list of groups obtained by splitting paren_string on
the literal space separator. Preserve the benchmark-safe tokenization anchor.
```

Reference Lean anchor.

```
let paren_space_split := paren_string.split (fun x => x = ' ');
result.length = paren_space_split.length /\
forall i, i < result.length ->
  let group := paren_space_split[i]!;
balanced_paren_non_computable group '(' ')' ->
  count_max_paren_depth group = result[i]!
```

Figure 5: A compact example of NLFPs carrying type and helper choices: `String` tokenization, `List Nat` output, index alignment, and Lean helper predicates are all explicitly planned.

NLFP excerpt.

```
The function takes a list of integers and returns the positive elements of the
input list. "Positive" means strictly greater than 0. The benchmark relation is
not merely positivity or set-like membership.

Collection semantics:
- Each output value must be positive and present in the input.
- Every positive input value must appear in the output.
- Multiplicity is preserved with count equality; do not weaken to set membership.
```

Decoded Lean.

```
let spec (result : List Int) :=
  result.all (fun x => x > 0 /\ x ∈ numbers) /\
  numbers.all (fun x => x > 0 -> x ∈ result) /\
  result.all (fun x => result.count x = numbers.count x);
exists result, implementation numbers = result /\ spec result
```

Figure 6: A list-output NLFP makes the critical positivity, membership, and multiplicity decisions explicit prior to target-language decoding.

```
let spec (res : Bool) :=
  numbers.length > 1 ->
    if res then numbers_within_threshold else not numbers_within_threshold
```

This explicitly addresses the underspecification inherent in raw English. A phrase like "Any two numbers closer than the threshold" fails to specify whether the witnesses are values or positions, whether duplicate values count, or exactly where the length guard should scope.

Lean syntax and local scaffold. Beyond semantic planning, prompt-refinement logs reveal that NLFPs acquire knowledge of compile-relevant Lean surface structures. For example, one feedback item specifically instructed the encoder to record tuple destructuring as `let (smaller, larger) := result` and to format a universal closest-pair clause as chained implications, countering the decoder's tendency to normalize them into projections and conjunction-packed premises. Another feedback loop caught an indexed-variable error in a recursive sum relation: the decoded tail condition incorrectly used `-99 <= arr[i]!` instead of the required `-99 <= arr[i']!`. These are not mere cosmetic adjustments; they form the vital structural scaffolding that ensures the binders, local names, and indexed subterms within the decoded Lean predicate perfectly align.

B Correct vs. Incorrect Testcases

Evaluating formal specifications fundamentally requires probing both their soundness and completeness. To this end, positive and negative testcases exert complementary selective pressures on the generated predicate. Positive cases verify that the specification accommodates canonical, expected behavior, while negative cases ensure it rigorously rejects deliberately corrupted outputs. Both are indispensable, as a specification may simultaneously be too strong in one region of the input space and too weak in another.

The contrast between *Validation Feedback* and *Verification Feedback* is easiest to see on concrete list relations. For the positive-list task, computable validation can rapidly accept sampled canonical cases such as `[-2, 3, 0, 3, -1, 1] -> [3, 3, 1]`, confirming strict positivity and duplicate preservation across many inputs. Proof validation checks fewer behaviors, and a failed obligation can reflect proof-search difficulty rather than an incorrect specification. However, a successful obligation gives kernel-checked evidence that the generated predicate accepts a particular input-output pair or rejects a corrupted one.

The nested-parentheses task illustrates why this testcase-level view is useful. The generated relation captures the regular nonempty behavior, accepting examples such as `"() ((())) ()" -> [1,3,1]`, where each token is mapped to its maximum nesting depth. A positive boundary case such as `"" -> []` then exposes exactly where the remaining ambiguity lies: empty-string tokenization and the benchmark's guarded helper semantics. This is more actionable than a generic failure signal because it identifies the specific formalization choice that needs refinement. Negative cases

provide the complementary pressure. For example, corrupted outputs such as replacing `[1,3,1]` with a depth-mismatched list should be rejected, and our proof-validation artifacts prove 9 out of 10 wrong-output obligations for this same family. Together, the positive and negative cases show whether a predicate is both permissive enough to accept intended behavior and restrictive enough to exclude invalid behavior.

C Testcase Proof vs. Full Specification Equivalence

Proving that two full specifications are equivalent is a substantially harder problem than validating concrete testcase behavior. Full equivalence requires Lean to show that two predicates agree across all possible inputs and outputs, often over infinite domains and user-defined helper functions. In practice, this demands induction, extensional reasoning, auxiliary lemmas, arithmetic facts, and careful unfolding of local definitions. These proof obligations are far beyond what current automated proof generators can reliably discharge across a heterogeneous benchmark such as CLEVER.

By contrast, *Verification Feedback* deliberately narrows the proof target to one concrete input-output behavior at a time. We inject a dummy implementation that statically returns a fixed output, so the resulting theorem asks whether the generated predicate accepts a positive behavior or rejects a corrupted one. Many such obligations reduce to decidable equalities, finite list membership/count facts, Boolean conditions, or ground arithmetic propositions. This makes proof construction more tractable: whenever a *Verification Feedback* obligation succeeds, Lean's kernel checks the proof of that constructed proposition. The result provides evidence at the selected behavioral boundary, conditional on the testcase and obligation faithfully representing the intended behavior; it does not establish universal specification equivalence.

D When Testcase Proof Obligations Succeed or Fail

Empirical analysis of our proof artifacts indicates that obligations are most likely to succeed when the fixed testcase effectively collapses the formal specification into decidable list operations, arithmetic checks, or Boolean evaluations. The `List Int` filter task serves as a prime example: once the input and output lists are concretely fixed by the dummy implementation, the proof obligation reduces to establishing straightforward equalities or membership/count facts over finite elements.

Conversely, obligations typically fail when proof construction would need to repair a representation-level mismatch, discover an omitted helper lemma, or elaborate a predicate whose underlying syntax is not type-correct. The string XOR case in Figure 7 illustrates this diagnostic role: proof validation localizes the issue to Lean's strict representation of string positions, rather than to a deep mathematical obstacle.

This string XOR case also demonstrates why computable and proof validation are not interchangeable. The Python executable checker can pass all sampled cases, giving a strong behavioral signal, while proof validation proves zero cases because the generated Lean tries to access strings using `String.get! i` with a `Nat` index. Lean-compatible string access instead requires constructing a `String.Pos`. Thus a specification can appear correct under executable translation while still violating the type-theoretic constraints of the proof-assistant target.

This separation helps examine potential false positives. The LLM-translated Python checker provides scalable feedback, but an overly charitable translation can inflate its pass rate. Lean checks each constructed proposition, yet an incorrect obligation or testcase polarity can still misrepresent the intended evaluation. Cross-channel consistency, manual auditing, and testcase-budget sensitivity therefore complement kernel checking (Appendix F.1). A higher *Proof Validation Pass Rate* indicates that more sampled obligations were discharged; it does not by itself establish that the natural-language requirement was formalized correctly.

E Additional Experimental Analysis

Main comparison. In the primary `gpt-5.4` setting, SPECBRIDGE improves *LLM Specification Alignment* from 60.6% to 66.7%, *Computable Validation Pass Rate* from 76.2% to 91.4%, and *Proof Validation Pass Rate* from 58.2% to 86.6%. The especially large proof-validation gain suggests that

NLFP excerpt.

```
Define the per-index XOR bit relation.
The result has the same length as a and b.
For every index i, result[i] is '1' exactly when a[i] and b[i] differ, and '0'
exactly when they are equal.
```

Lean issue exposed by proof validation.

```
generated: forall i, i < a.length ->
  ((result.get! i = '1') <-> (a.get! i != b.get! i))

Lean-compatible pattern: let i_pos := String.Pos.mk i;
  (a.get i_pos = b.get i_pos -> result.get i_pos = '0') /\
  (a.get i_pos != b.get i_pos -> result.get i_pos = '1')
```

Figure 7: Proof validation can expose Lean-specific type representation errors that executable testcase checks may inherently miss.

NLFPs and iterative feedback do more than make the generated Lean look plausible: they help the model produce relations whose concrete behavioral obligations Lean can certify more often.

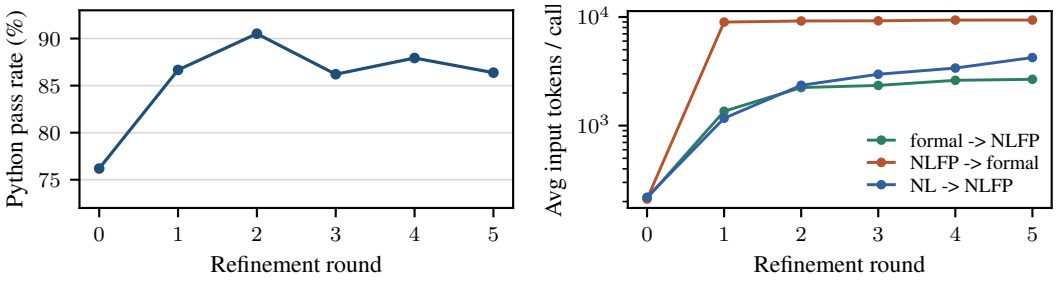
Ablation interpretation. The ablation study separates several sources of improvement. Removing reconstruction auto-encoding causes the largest drop in *LLM Specification Alignment*, indicating that learning NLFPs from formal-specification reconstruction is important for preserving task meaning. Removing NLFP prompt refinement causes the largest drop in *Proof Validation Pass Rate*, showing that iterative feedback is particularly important for producing proof-friendly relational structures. Removing Validation Feedback or canonical-program semantic feedback leaves *Computable Validation Pass Rate* relatively stable, but depresses both semantic alignment and proof validation, suggesting that executable behavioral signals guide relation choices even when they are not the final bottleneck.

Cross-model behavior. The framework improves over few-shot + CoT across all evaluated backends, but the dominant gain differs by model. `deepseek-v3.2` benefits most in *LLM Specification Alignment*, rising from 24.2% to 42.4%, whereas `gpt-5.4` benefits most in *Proof Validation Pass Rate*, rising from 58.2% to 86.6%. `gemini-3.1-pro-preview` starts from a stronger few-shot + CoT baseline and achieves the highest absolute *LLM Specification Alignment* and *Computable Validation Pass Rate*. These differences reinforce that semantic judgement, executable behavior, and proof-backed certifiability are complementary rather than interchangeable.

Statistical reporting. The main comparison, component ablations, and cross-backend rates are aggregate held-out rates over the 161-instance CLEVER benchmark under five disjoint folds, so each instance contributes once per compared system. We report aggregate rates without formal significance tests and make no p-value-based claims.

Prompt-refinement dynamics. For the `gpt-5.4` backend, direct few-shot + CoT generation yields a computable-validation pass rate of 76.19%. After refinement round 1, this rises to 86.67%, and reaches 90.52% by round 2. Later rounds fluctuate in the mid-to-high 80s rather than improving monotonically. This trajectory suggests that round 2 can be a useful cost-performance checkpoint, while our reported final evaluation uses the round-5 prompt bundle and remains above the baseline. Early feedback appears to teach broad formalization rules such as guard placement, collection semantics, type anchors, and local Lean scaffolding, while later feedback addresses narrower edge cases.

Prompt-length behavior. Prompt length mirrors this training dynamic. Round 0 denotes the initial, unrefined prompt bundle; rounds 1–5 denote the bundles after each update. The `NL -> NLFP` prompt grows the most because it must infer implicit formalization choices from raw, underspecified natural language. By contrast, the `formal -> NLFP` and `NLFP -> formal` prompts operate closer to an explicit Lean-grounded scaffold and require less contextual expansion.



(a) Python pass rate

(b) Prompt length

Figure 8: Prompt-refinement diagnostics for the `gpt-5.4` run, showing early validation gains and changes in prompt length.

Table 8: `gpt-5.4` API cost for a 161-task training run and forward evaluation, computed from usage logs at the default price tier.

Stage	Scope	API cost	Cost/problem
Training SpecBridge	one 161-problem run	$58.28	$0.362
Forward evaluation, few-shot + CoT	161 problems	$0.46	$0.003
Forward evaluation, SpecBridge	161 problems	$1.20	$0.007

F Supplementary Controls, Transfer, and Reliability Diagnostics

Reporting scope. The baseline comparison and forward-feedback ablation cover all 161 CLEVER tasks under the five-fold protocol. Conventional metrics, transfer, and evaluator-reliability experiments provide further diagnostics.

Supplementary protocol details. The forward-feedback ablation uses the CLEVER five-fold splits, the round-5 prompt bundle, and 10 positive plus 10 negative testcases per task. Alignment is calculated over 161 held-out task-level predictions; computable and proof validation use up to 3,220 testcase-level checks per method. The compared methods use the same task inputs and evaluation procedure.

Table 9: RQ2. Effect of removing forward feedback from the NLFP pipeline on all 161 CLEVER tasks.

Method	LLM Specification Alignment	Computable Validation Pass Rate	Proof Validation Pass Rate
Full SpecBridge	**0.67**	**0.91**	**0.87**
Without forward feedback	0.52	0.78	0.43

The forward-feedback ablation retains reconstruction-guided NLFP planning but removes downstream validation and verification refinement. It differs from the *w/o Validation Feedback* variant, which removes only one feedback channel.

For VERINA, we evaluate all 189 tasks in the released dataset using the `gpt-5.4` inference backend. Each task is normalized to its natural-language description plus fixed Lean signature. The prompts are frozen after CLEVER training; no VERINA reference or validation feedback is used for further refinement. The same backend, tasks, and evaluator are used for both methods. With 10 positive and 10 negative testcases per task, alignment covers 189 task-level decisions and computable or proof validation covers up to 3,780 checks per method.

F.1 Conventional Specification-Synthesis Metrics

Type checking measures formal validity, while positive acceptance and negative rejection characterize behavioral discrimination: an underconstrained predicate may accept both correct and corrupted

Table 10: Supplementary conventional metrics. Negative Rejection counts corrupted outputs for which Lean proves non-satisfaction; failure to prove rejection is not evidence of acceptance.

Metric	Few-shot + CoT	SPECBRIDGE
Initial Type-Check	0.61	0.61
Repaired Type-Check	0.94	0.97
Positive Acceptance	0.76	0.85
Proved Negative Rejection	0.16	0.53
Exact Match	0.00	0.00

outputs. The initial type-check rate is unchanged, but repaired validity and both behavioral diagnostics improve (Table 10). Exact match remains zero for both systems and is only a surface diagnostic because equivalent Lean relations may have different syntax. Across 161 tasks with 20 testcases per task, type checking and exact match use up to 161 candidate-level observations, positive acceptance up to 1,610 positive checks, and proved negative rejection up to 1,610 negative checks per method.

F.2 Cross-Channel Consistency, Audit, and Testcase Coverage

Full Consistency is evaluated per testcase behavior over the 161 CLEVER tasks, with 20 testcases per task (Table 6). Its conjunctive criterion requires the executable checker to return the expected result, the corresponding Lean obligation to compile, and Lean to prove the expected positive or negative proposition. This prevents either channel alone from determining success, although shared settings and cross-channel agreement cannot exclude correlated errors.

The artifact audit checks faithful Python translation, correct testcase polarity, faithful Lean-obligation construction, and agreement between a candidate specification and the task. These checks distinguish evaluator fidelity from task-level specification quality: an evaluator can faithfully check sampled behavior while missing a boundary defect in the candidate. The tenfold testcase-budget increase in Table 7 uses fixed specifications over the same 161 tasks: the 10-positive/10-negative setting contains 3,220 obligations, and the 100-positive/100-negative setting contains 32,200. The proof-validation pass rate decreases from 0.89 to 0.85 as coverage increases.

F.3 Task-Level Failure Analysis

Table 11: Primary failure mechanisms considered in the CLEVER analysis.

Primary failure type	Typical manifestation
Incorrect guard or precondition scope	A misplaced implication permits vacuous satisfaction.
Semantic underconstraint	The output is insufficiently tied to the input.
Collection, multiplicity, or ordering mismatch	The relation loses counts or ordering requirements.
Semantic overconstraint	Valid outputs are rejected.
Lean-specific type or helper mismatch	The proposed relation cannot elaborate as intended.
Other boundary cases	Edge behavior is formalized incorrectly.

Table 11 organizes the error mechanisms illustrated by failed specifications. Guard-scope errors include parsing conditions placed inside an existentially quantified implication, allowing vacuous satisfaction. Underconstraint includes replacing the input-to-parsed-value relation with `True`, disconnecting the result from the input. Other cases mishandle empty delimiter fragments, introduce fold-initialization values absent from the input, or strengthen approximate numerical requirements to exact equality. These observations motivate explicit NLFP instructions for scope, input-output dependencies, collection semantics, and boundary cases.

G Additional Related Work

From semantic parsing to specification inference. The endeavor of natural-language formalization traces its roots to semantic parsing, which maps natural language sentences directly to logical

forms [12]. Within software engineering, program-specification inference boasts a rich, pre-LLM history: tools like Daikon infer likely invariants empirically from execution traces [13], while Houdini filters candidate annotations using feedback from the ESC/Java verifier [14]. SPECBRIDGE differentiates itself by operating directly from natural-language task descriptions and fixed Lean signatures, leveraging reconstruction to explicitly learn intermediate formalization plans (NLFPs) rather than inferring specifications purely from traces or static code analysis.

Temporal-logic roots. Temporal logics, notably LTL and branching-time temporal logic, serve as the foundational specification languages for reactive and concurrent systems [15, 16]. While this lineage heavily motivates the broader natural-language-to-formal-specification agenda, temporal formulas are primarily designed to govern reactive traces or state-transition properties. In stark contrast, CLEVER-style Lean `problem_spec` synthesis demands typed, function-level input-output relations. This shift in target domain makes the handling of collection semantics, local helper definitions, witness binders, and rigid target-language representation choices the central challenge of our work.

Recent specification-generation systems. Recent work spans different input interfaces and targets. nl2postcond maps natural-language intent to program assertions, while NL2Contract extends this line to preconditions and postconditions [5, 6]; these are the closest requirement-centered tasks. AutoSpec and SpecGen instead incorporate program analysis or implementation context, while FormalBench and VeriAct study other specification and verification settings [7, 8, 17, 18]. SPECBRIDGE targets Lean `problem_spec` relations from a requirement and fixed signature. Its deployable generator sees neither canonical code nor reference specifications, and the different target languages and evaluation pipelines preclude a direct numerical comparison with those systems.

Baseline scope. We use a CLEVER-style few-shot + CoT protocol as the primary baseline because it is the directly reproducible baseline for the same task interface: natural-language task plus fixed Lean signature in, Lean `problem_spec` out. More elaborate search strategies such as Tree-of-Thought are prompting variants rather than established systems for this Lean specification-synthesis setting, and they still require task-specific validators and selection criteria. Proof-oriented systems such as COPRA, AlphaVerus, and Baldur solve a complementary problem: they repair or synthesize proofs/programs once a target specification or verification condition is already available. Comparing against them would therefore change the input contract rather than isolate the contribution of NLFP-based specification generation. We include free-form planning, Best-of-5 validation reranking, and direct Lean feedback refinement as baselines in Table 1. We classify NLFPs without forward feedback as a controlled ablation of SPECBRIDGE (Table 9).

H Experimental Environment

To ensure full reproducibility, all experiments were conducted on an isolated 48-core x86-64 cloud server provisioned with 96GB of memory. This centralized server orchestrates the entire evaluation pipeline, including API request dispatching, Lean environment checking, testcase construction, Python execution, and results aggregation. All large language model invocations were routed through official remote APIs; no local LLMs were served or fine-tuned during this study.

To maximize throughput, our evaluation runner utilizes up to 64 parallel workers, executing independent model calls and validation jobs concurrently. We evaluated the framework across three state-of-the-art model backends: `gpt-5.4-2026-03-05`, `deepseek-v3.2`, and `gemini-3.1-pro-preview`.

The canonical program utilized for all behavioral feedback corresponds to the ground-truth Python implementation provided with each CLEVER benchmark instance. During the evaluation phase, *Validation Feedback* translates the generated Lean preconditions and postconditions into executable Python assertions, subsequently running them against the canonical-program testcases. Parallelly, *Verification Feedback* rigorously constructs Lean proof obligations using these exact same positive and negative testcase behaviors to provide kernel-checked validation.

NeurIPS Paper Checklist

1. **Claims**
   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?
   Answer: [Yes].
   Justification: The abstract and introduction state the two main contributions: NLFP-based Formal Specification Synthesis generation and a multi-layer evaluation protocol. The introduction now separates the general task from the CLEVER benchmark, and the limitations section states the restricted scope.
2. **Limitations**
   Question: Does the paper discuss the limitations of the work performed by the authors?
   Answer: [Yes].
   Justification: The Discussion section includes a Limitations paragraph covering the single-function specification scope, the boundaries of cross-dataset transfer, dependence on canonical programs, and learned evaluation components.
3. **Theory assumptions and proofs**
   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?
   Answer: [N/A].
   Justification: The paper does not claim new theoretical theorems. It introduces a representation, an evaluation protocol, and an empirical prompt-refinement system.
4. **Experimental result reproducibility**
   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper?
   Answer: [Yes].
   Justification: The Method and Experiments sections specify the benchmark, held-out 5-fold protocol, compared systems, forward inputs, evaluation metrics, prompt-refinement procedure, model backends, API-cost accounting, and compute setup. The project site provides the implementation, prompts, generated artifacts, evaluation scripts, result summaries, and reproduction commands.
5. **Open access to data and code**
   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?
   Answer: [Yes].
   Justification: The introduction links to a project site that contains the source code, prompt bundles, generated NLFP/formal-specification artifacts, evaluation scripts, result summaries, and instructions for reproducing the reported experiments.
6. **Experimental setting/details**
   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?
   Answer: [Yes].
   Justification: The Experiments section specifies the 5-fold validation design, compared systems, model settings, remote-API setup, prompt-refinement rounds, few-shot baseline, and LLM-based prompt-refinement procedure used instead of gradient descent. The released scripts and artifacts record the low-level prompt, decoding, testcase, and proof-obligation settings used to obtain the reported numbers.
7. **Experiment statistical significance**
   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?
   Answer: [Yes].
   Justification: The paper reports full held-out aggregate rates over all 161 CLEVER instances under a leakage-free 5-fold protocol and includes an appendix note on statistical reporting. Because each instance contributes once per compared system, we report aggregate comparisons, ablations, and cross-model trends rather than per-run error bars or p-value-based significance claims.
8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes].

Justification: The Experimental Environment and Results sections state that experiments use official remote APIs, describe the 48-core x86-64 orchestration machine with 96GB memory and up to 64 parallel workers, and report `gpt-5.4` API costs from persisted usage logs. The released artifact records the exact commands and software environment used by the runners.

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes].

Justification: The work studies automated formal-specification generation and evaluation on programming benchmark artifacts. It does not involve human subjects, sensitive personal data, surveillance, deception, or high-risk deployment.

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes].

Justification: The paper frames the work as an assistive tool for formal-specification drafting and proof-backed validation, while the Discussion limitations describe risks from learned executable checks, generated proof scripts, incomplete coverage, and evaluation blind spots.

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A].

Justification: The paper does not release a high-risk generative model or scraped dataset. The new artifacts are prompts, generated specifications, evaluation code, and analysis artifacts.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes].

Justification: The paper cites CLEVER, the proof-assistant and library dependencies used in the benchmark instantiation, and related systems, and the release documentation records licenses and terms of use for benchmark data, proof-assistant dependencies, and remote model APIs.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes].

Justification: The paper introduces prompts, NLFP artifacts, evaluation scripts, proof/testcase artifacts, and generated specifications. The appendix documents representative NLFP and validation artifacts, and the project site documents schemas, prompt bundles, reproduction commands, generated artifacts, and expected outputs.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A].

Justification: The work does not include crowdsourcing experiments or human-subject studies.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A].

Justification: The work does not involve human subjects and therefore does not require IRB approval.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes].

Justification: LLMs are central to the method: they generate NLFPs and formal specs, judge semantic agreement, translate specs into Python assertions, and refine prompts from structured feedback. These uses are described in the Method and Experiments sections.