PREFACE - A Reinforcement Learning Framework for Code Verification via LLM Prompt Repair

Manvi Jha
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
manvij2@illinois.edu

Jiaxin Wan
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
wan25@illinois.edu

Huan Zhang
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
huanz@illinois.edu

Deming Chen
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
dchen@illinois.edu

Abstract

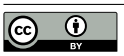
Large Language Models (LLMs) have emerged as powerful tools for code generation. Yet, they often struggle to produce code that is both syntactically and semantically correct, particularly when correctness must be formally verified. Addressing this gap, we present a novel, model-agnostic framework that couples LLMs with a lightweight reinforcement learning (RL) agent to enable scalable, robust, and formally verifiable code generation without the need for costly model fine-tuning. Centered on the generation of Dafny code, a formally verifiable programming language, our system initiates with LLM-generated code, which is rigorously evaluated within an integrated verification environment. Upon failure, we feed the erroneous code and error metadata to an RL agent trained to explore the prompt-code space (i.e., the set of possible prompt edits paired with their resulting generated code variants). This agent strategically selects corrective prompts to minimize verification iterations, effectively steering the LLM toward correct outputs using its latent capabilities. Once verified, Dafny code can be systematically translated to C for high-level synthesis (HLS), ensuring correctness-by-construction in downstream hardware design workflows. On a 100-task benchmark, PREFACE's error-guided prompt refinement raises verification success by up to 21% — 14% for ChatGPT-4o, 17% for ChatGPT-o1-mini, 10% for Qwen2.5-Coder-14B, 4% for Qwen2.5-7B, and 21% for Gemini-2-Flash—demonstrating substantial gains across diverse LLMs.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; Formal methods; • **Computing methodologies** → *Machine learning*; • **Theory of computation** → Logic.

Keywords

Dafny, Formal verification, Large Language Models (LLMs), Reinforcement learning, Prompt optimization, Automated reasoning



GLSVLSI '25, New Orleans, LA, USA

ACM ISBN 979-8-4007-1496-2/25/06
https://doi.org/10.1145/3716368.3735300

ACM Reference Format:
Manvi Jha, Jiaxin Wan, Huan Zhang, and Deming Chen. 2025. PREFACE - A Reinforcement Learning Framework for Code Verification via LLM Prompt Repair. In *Great Lakes Symposium on VLSI 2025 (GLSVLSI '25), June 30–July 02, 2025, New Orleans, LA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3716368.3735300

1 Introduction

Large Language Models (LLMs) have significantly advanced the field of automated code generation, enabling developers to synthesize functional code from natural-language prompts [3]. While LLMs excel at producing syntactically correct code in popular languages like Python, they often fail in domains where formal correctness is non-negotiable, such as cryptographic libraries, safety-critical systems, and hardware design [17] [10]. Languages like Dafny [4], which are specifically designed to support formal verification, pose a unique challenge, if such code is to be generated by LLMs: even minor specification mismatches or missing invariants can result in unverifiable or unsafe outputs.

A conventional approach to improving performance in such domains is fine-tuning [14], which adapts an LLM to a specific task distribution. However, this strategy is both resource-intensive and brittle. Fine-tuning large models requires massive computational resources and substantial annotated data. Furthermore, given the broad and already saturated knowledge encoded in modern LLMs, excessive fine-tuning may risk overwriting useful capabilities.

We argue that the latent reasoning potential of LLMs is sufficient to solve many existing domain-specific coding tasks—if properly guided. We observe that the effectiveness of LLMs in formal verification tasks hinges on the quality of the prompts they are given. Crafting such prompts manually is a labor-intensive process, especially when verifier feedback (this is the message associated with running the integrated verification engine for the given code) must be interpreted and integrated into future attempts. Yet this verifier feedback is precisely the kind of structured signal that could be used to learn how to steer the model more effectively.

This insight motivates our approach: rather than modifying the model itself, we propose a lightweight and scalable alternative of training a Small Language Model (SLM) within a reinforcement learning (RL) framework to generate optimized prompts for a fixed, general-purpose LLM. This RL-driven prompt adaptation enables accurate and verified code synthesis without direct fine-tuning,

unlocking the full potential of LLMs for specialized domains. The proposed approach is discussed in more detail in Section III.

We focus on Dafny, a verification-aware programming language that supports formal specifications through built-in constructs such as preconditions, postconditions, invariants, and lemmas. Dafny automatically translates these specifications into verification conditions, which are discharged by SMT solvers, enabling correctness-by-construction software development. Its design balances expressive power and automation, making it an ideal platform for synthesizing and verifying programs with minimal user intervention. Crucially, however, domain-specific datasets for Dafny are extremely limited (e.g., DafnyBench provides only ∼700 training tasks [7]), so conventional fine-tuning of large models is infeasible. Instead, our approach reuses a frozen, general-purpose LLM's existing capabilities and steers it via an RL-driven prompt agent—making it possible to tackle this low-resource verification task without costly, data-hungry model retraining. Verified Dafny programs can then be translated to C for high-level synthesis, bridging formally verified software and hardware design in a practical, trustworthy workflow.

1.1 Contributions

This work presents a novel reinforcement learning-based framework for guiding LLMs toward generating formally verified code without requiring fine-tuning. Our key contributions are:

- **A model-agnostic architecture for verifiable code generation:** We propose a modular framework that leverages a small RL agent to adapt prompts for a frozen, general-purpose LLM. This enables verified code synthesis in underrepresented languages like Dafny without modifying the LLM itself.
- **A formal verification feedback loop:** We design a generation-verification loop that uses structured feedback from an integrated Dafny verifier. The RL agent interprets error metadata and learns to iteratively refine prompts to improve verification success.
- **Prompt optimization as an RL task:** We formalize prompt construction as a reward-driven optimization problem and show how verifier outcomes can serve as control signals in an RL setup.
- **Significant empirical improvements:** The proposed model shows a significant improvement in the success rate for the generated code. An experiment with a test set of 100 examples doubles verification success across various models.

Together, these contributions offer a scalable and resource-efficient alternative to traditional LLM fine-tuning, paving the way for more reliable AI-assisted programming in various applications.

2 Background and Related Works

2.1 Dafny: Syntax and Verification Workflow

Dafny [4] is a verification-aware imperative language that integrates formal specifications directly into the program text: each method may declare its preconditions with "requires" and its post-conditions with "ensures", ensuring that callers and implementers agree on expected behavior. Loop constructs must carry "invariant" annotations that articulate and enforce properties preserved across iterations, while "ghost" variables and separate "lemma" functions enable purely logical reasoning without affecting runtime execution. Under the hood, Dafny translates these high-level specifications into first-order verification conditions, which are automatically discharged by an SMT back end [15], yielding correctness-by-construction guarantees. Its seamless integration of specification, proof, and execution makes Dafny an ideal target for LLM-driven code synthesis in domains demanding provable reliability.

2.2 Dafny Generation & Verification with LLMs

Large Language Models have increasingly been applied to assist formal verification and correct-by-construction code generation [6], [17], [8]. Misu et al. [9] tackled the challenge of verified synthesis by using LLMs to generate program implementations that satisfy formal specifications, leveraging verification feedback to filter or guide outputs. This work demonstrated that GPT-4 can synthesize Dafny methods that pass the verifier for a majority of benchmark tasks when guided by structured chain-of-thought prompts with retrieved examples, whereas naive prompting succeeds on only a small fraction. This underscores how prompt engineering can impart logical structure that improves the rate of verified solutions.

Poesia et al. [12] first introduced Dafny-Annotator, which leverages an LLM together with a greedy search over candidate logical annotations to satisfy the Dafny verifier. While this approach yielded modest gains in small-scale tests, its effectiveness was limited by the scarcity of Dafny examples. The same work proposed DafnySynth, a synthetic program generation pipeline that automatically creates Dafny training tasks to overcome this data bottleneck. When an LLM is fine-tuned on this augmented dataset, its pass rate on DafnyBench [7] jumps from 15.7% to 50.6%, showing the power of synthetic data for low-resource formal verification domains.

In parallel, Li et al. [5] introduced a code generation framework that treats Dafny as a verification-aware intermediate language. To generate target languages like Python, their prototype uses LLMs to generate Dafny code from natural language prompts, which is then verified and compiled to Python. Crucially, their method shields the user from Dafny syntax, instead relying on natural language interactions to iteratively refine both specifications and code.

In each case, the core challenge is similar: LLMs must output formally correct artifacts (code or proofs) and typically require guidance – either through carefully engineered prompts, iterative verification feedback, or additional training – to do so reliably. Our work situates itself at the intersection of these trends. Like Dafny-Annotator, we close the loop with a verifier, but rather than fine-tuning a general-purpose LLM to emit verifiable code, we train a small RL-based agent to steer a frozen general-purpose LLM via its prompts. This preserves the breadth and power of an LLM (no fine-tuning of it is needed) while injecting a learning mechanism to adapt its outputs to the domain of formally verified Dafny code. Conceptually, we treat prompt construction as a decision-making problem, where verification success is the reward bridging the formal verification approaches above with RL techniques, as discussed next.

2.3 LLMs for RL and Prompt Optimization

Recent research has explored incorporating LLMs into RL pipelines, both for single-agent decision making and multi-agent coordination.

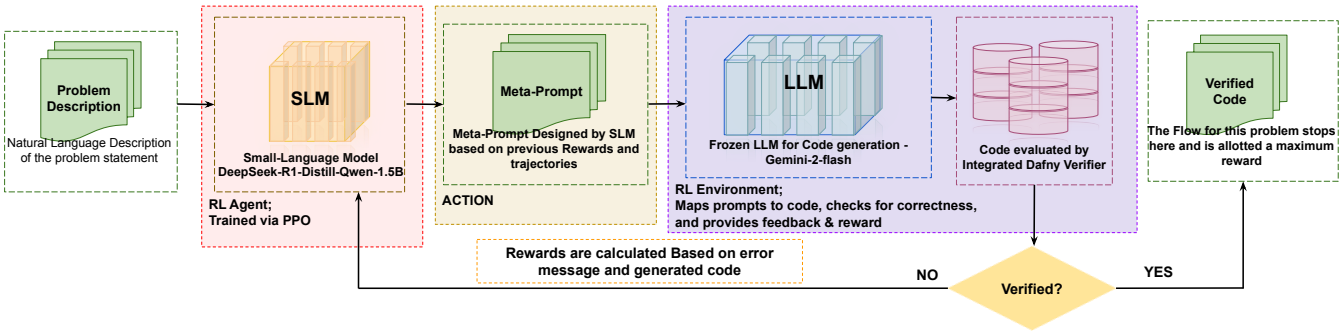


Figure 1: Overview of the proposed PREFACE framework.

Broadly, two paradigms have emerged: LLM-guided policy learning, where an RL agent benefits from the high-level reasoning of a language model, and prompt optimization via RL, where an agent learns to craft better prompts or instructions for an LLM.

In the single-agent setting, several works have treated an LLM as a source of high-level guidance for an RL agent tackling a complex task. For instance, Du et al. (ELLM) [1] use an LLM to suggest subgoals for the agent, providing a form of curriculum or exploration hint that improves learning efficiency. These approaches treat the LLM almost like an oracle or an expert demonstrator. In particular, the prompt template—for example, "Given the current state S, what sub-goal should the agent pursue next?" —is manually engineered once and remains unchanged throughout training, so the LLM always sees the same instruction structure. The model's outputs are then used as additional training data or rewards for the RL agent.

In multi-agent settings, LLMs have been used to improve coordination and learning efficiency without embedding the model in every agent. YOLO-MARL [19] exemplifies this by using a one-time LLM prompt to produce high-level strategies and policy sketches that are distilled into agents before training, offering efficiency but no adaptability if initial plans falter. In contrast, LERO [16] periodically re-queries the LLM within a feedback loop to generate and evolve hybrid reward functions (for finer credit assignment) and observation enhancement mappings (to infer global context), trading added complexity for adaptive flexibility.

Our work builds on these ideas by applying RL-guided prompt optimization to formal verification. A lightweight agent learns to adapt prompts based on verifier feedback, guiding a frozen LLM toward verified Dafny code. Unlike static prompting or fine-tuning, our approach evolves prompts through interaction, treating verification success as a reward signal. This enables efficient, correct-by-construction code generation without retraining large models.

3 Proposed Work

3.1 System Pipeline Overview

Our framework introduces an RL-guided pipeline for verified code synthesis in Dafny, a language equipped with formal specifications and automated verification. In this context, "RL-guided" means that the Small Language Model (SLM)—DeepSeek-R1-Distill-Qwen-1.5B—is itself trained with reinforcement learning: it learns to adapt its prompts based on feedback from Dafny's verifier. A frozen Large LLM (Gemini-2-flash or Qwen2.5-Coder-14B) then uses those adapted prompts to generate candidate code. The SLM and LLM interact iteratively until the generated Dafny program passes verification. Figure 1 illustrates the overall PREFACE workflow.

Initial Prompt & Code Generation. Given a task description with specifications like pre/post conditions - τ, an initial prompt p_0 is constructed using the manual prompt template φ and passed to the frozen Large LLM - $\mathcal{M}$, which generates a candidate Dafny implementation - c_t, where $t = 0, 1, ...T_{max} - 1$, where T_{max} denoted the maximum iterations which is talked about in more detail in later subsections.

$$p_0 = \varphi(\tau) \tag{1}$$

$$c_t = \mathcal{M}(p_t), t = 0, 1, .., T_{max} - 1 \tag{2}$$

While the model leverages broad programming knowledge, the initial output may not meet verification requirements.

Formal Verification Feedback. The Dafny verifier automatically checks the generated code against the provided specification ($c_t \models \tau$).

$$E_t, o_t = \text{Verify}(c_t, \tau) \quad (e_t = 0 \text{ on success}). \tag{3}$$

If the code verifies (i.e., all proof obligations are discharged), then the error count, $e_t = 0$, the process terminates and returns c_t. Otherwise, detailed error messages (o_t) serve as feedback, identifying failed conditions such as unproven invariants or post-conditions. These verifier outputs act as a correctness oracle and yield reward signals for the RL agent (we will discuss more about the RL agent in later subsections).

Prompt Adaptation via RL. If verification fails, the SLM agent, whose policy network weights are denoted by θ, processes the verifier's feedback to construct the current state as:

$$s_t = \psi(p_t, c_t, o_t) \tag{4}$$

And proposes a revised prompt as:

$$p_{t+1} \sim \pi_\theta(s_t), \tag{5}$$

Algorithm 1 PREFACE Prompt Adaptation Loop

Require: Specification τ Initial prompt $p_0 \leftarrow \varphi(\tau)$ Max iterations $T_{\max} = 7$

for $t = 0 \dots T_{\max} - 1$ **do**
  $c_t \leftarrow \mathcal{M}(p_t)$
  $e_t \leftarrow \text{Verify}(c_t, \tau)$
  if $e_t = 0$ **then**
    return c_t
  end if
  $s_t \leftarrow \psi(p_t, c_t, e_t)$
  Sample prompt edit: $\Delta p_t \sim \pi_\theta(\cdot \mid s_t)$
  $p_{t+1} \leftarrow p_t \oplus \Delta p_t$
end for

This interaction is modeled as a Markov Decision Process (MDP), where each state encodes the latest code, verifier output, and previous prompt. The SLM agent selects a prompt modification intended to guide the LLM toward correcting the failure. The system forms a trajectory, iterating until a verified solution is found or a step limit is reached. Unlike prior approaches relying on static retries or fine-tuning, our method learns adaptive refinements via Reward-Guided Proximal Optimization.

3.2 MDP Formulation for Prompt Adaptation

We cast prompt adaptation as a Markov Decision Process (MDP), a standard framework for sequential decision-making under uncertainty. In this formulation, each decision step corresponds to proposing a prompt edit and observing the resulting effect on generated code and verifier feedback. An MDP is well-suited here because the process of refining prompts is inherently sequential—each edit influences future code generations and the remaining verification effort, so we can formally optimize for long-term success rather than greedy, one-shot improvements. By defining states, actions, transitions, and rewards over the prompt-code space, we leverage reinforcement learning algorithms to learn a policy that maximizes the cumulative verification reward.

We cast prompt refinement as a finite Markov Decision Process

$$\mathcal{G} = (\mathcal{S},\ \mathcal{A},\ P,\ R,\ \gamma), \tag{6}$$

where each step t corresponds to proposing a small edit Δp_t to the current prompt p_t, then observing the generated code c_{t+1} and verifier feedback e_{t+1}. This sequential view lets us optimize for long-term verification success rather than greedy, one-shot gains.

- **State Representation**
  Each state $s_t \in \mathcal{S}$ encodes three pieces of information:
  Prompt Text p_t: We pass the raw prompt p_t into the frozen LLM, which internally maps tokens to vectors. This captures the semantic context of the current prompt.
  Generated Code c_t: Similarly, the generated Dafny code c_t is fed as text into the same LLM. This summarizes the structure and content of the most recent program.
  Verifier Log e_t, o_t: We parse the Dafny verifier's output into a count vector $\mathbf{e}_t$, an integer count for each Dafny error, parsed directly from the verifier output o_t.
- **Action Space.** The action space is defined as

  $$\mathcal{A} = \{\, p_t \mid p_t \text{ is a prompt token sequence},\ t = 0, 1, \dots, 7 \,\}, \tag{7}$$

  Each action Δp_t consists of a discrete sequence of tokens drawn from the same vocabulary used by the frozen LLM. At each decision step, the policy network computes logits $\pi_\theta(a_t \mid s_t)$ over this set $\mathcal{A}$, and sampling from these logits yields the selected edit Δp_t. This edit is then used as the new prompt for the next conversation between the SLM and LLM thereby steering the subsequent code generation and verifier feedback.
- **Transition Dynamics** The transition dynamics are governed by the interaction between the edited prompt and the Dafny code generator. Formally, given the current state s_t and action p_{t+1} obtained by Eq. 4 and Eq. 5 the next code snippet is produced by using Eq. 2
  This generated code c_{t+1} is then verified by Dafny to produce an error count e_{t+1} and error message o_{t+1}. Finally, the next state s_{t+1} is constructed exactly as in the state-representation section, by encoding the updated prompt p_{t+1}, the new code c_{t+1}, and the verifier feedback e_{t+1} and o_{t+1}. In this way, the distribution $P(s_{t+1} \mid s_t, a_t)$ is implicitly defined by the combined generative and verification steps.
- **Reward Function** The reward $R(s_t, a_t)$ is defined as

  $$R(s_t, a_t) = \begin{cases} +R_{\text{succ}}, & \text{if } e_{t+1} = 0, \\ -\alpha\, e_{t+1} - \beta, & \text{if } e_{t+1} > 0, \end{cases}$$

  where e_{t+1} is the number of Dafny verification errors in the generated code c_{t+1} A successful verification (zero errors) yields a reward of $+R_{\text{succ}}$. Otherwise, the penalty scales linearly with the error count via $-\alpha\, e_{t+1} - \beta$, with shaping coefficients $\alpha = 0.2$ and $\beta = 0.5$. To prevent degenerate "fake" successes, any attempt resulting in an empty Dafny file incurs a large additional penalty. The policy π_θ is then learned with the RL algorithm to maximize the expected discounted cumulative reward

  $$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t\, R(s_t, a_t)\right]. \tag{8}$$

  where the discount factor $\gamma \in [0, 1)$ weights future rewards. We use $\gamma = 0.99$, which balances long-term verification success against stable learning updates.

3.3 Prompt Refinement via Proximal Policy Optimization

We employ the standard Proximal Policy Optimization (PPO) algorithm with a clipped-surrogate objective, and our dense, shaped reward R. Given a trajectory $\{(s_t, a_t, r_t)\}_{t=0}^{T}$, we first compute the discounted return:

$$G_t \ = \ \sum_{k=0}^{T-t} \gamma^k\, r_{t+k}, \tag{9}$$

We then calculate the advantage estimate:

$$\hat{A}_t \ = \ G_t \ - \ V_\phi(s_t) \tag{10}$$

This measures how much better the observed return G_t is compared to the critic's value estimate $V_\phi(s_t)$. Here, $V_\phi(s_t)$ is the critic

Table 1: Baseline Results: Pass rate (number of successful verifications in a set of 100 samples) with prompting strategies across different LLMs, with and without verifier error feedback.

Model	One-line Description		Detailed Description		One-line and Detailed Description	
	W/O Feedback	With Feedback	W/O Feedback	With Feedback	W/O Feedback	With Feedback
ChatGPT-4o	22%	37%	23%	36%	25%	36%
ChatGPT-o1-mini	22%	34%	22%	37%	23%	35%
Qwen2.5-Coder-14B	10%	18%	5%	16%	15%	21%
Qwen2.5-7B	3%	5%	3%	3%	3%	7%
Gemini-2-Flash	20%	34%	16%	34%	20%	33%

network's prediction of expected discounted return from state s_t. The critic is implemented by augmenting the same small language model used for prompt generation with a lightweight "value head," rather than by training an entirely separate network.

The actor objective (loss) is the clipped-surrogate loss is calculated as:

$$L_{\text{actor}}(\theta) = -\mathbb{E}_t\Big[\min\big(r_t(\theta)\,\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big)\Big], \quad (11)$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \quad \epsilon = 0.2. \quad (12)$$

Here, $\pi_\theta(a_t \mid s_t)$ is the probability that our current policy—i.e. the small LLM with parameters θ—chooses action a_t in state s_t, $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ is that same probability under the old parameter of the small LLM before we optimize L_{actor}.

The ratio $r_t(\theta)$ accounts for changes between the old and new policy, and clipping it by ϵ prevents any single update from moving the policy too far outside a small trust region. This surrogate objective thus stabilizes learning by ensuring each policy step stays within safe bounds.

The critic minimizes the mean-squared error

$$L_{\text{critic}}(\phi) = \mathbb{E}_t\big[\big(V_\phi(s_t) - G_t\big)^2\big]. \quad (13)$$

This trains the value head V_ϕ to accurately predict future shaped returns. During optimization, we use a smaller learning rate to reduce variance and maintain stability.

Each update consists of two phases on the same batch of rollouts: first, we update θ by descending $\nabla_\theta L_{\text{actor}}(\theta)$; second, we update ϕ by descending $\nabla_\phi\big(0.3\,L_{\text{critic}}(\phi)\big)$. Both optimizers use Adam with learning rate 1×10^{-4} and gradient-norm clipping at 0.5. This decoupled update ensures that the actor and critic leverage identical experience without their gradients interfering.

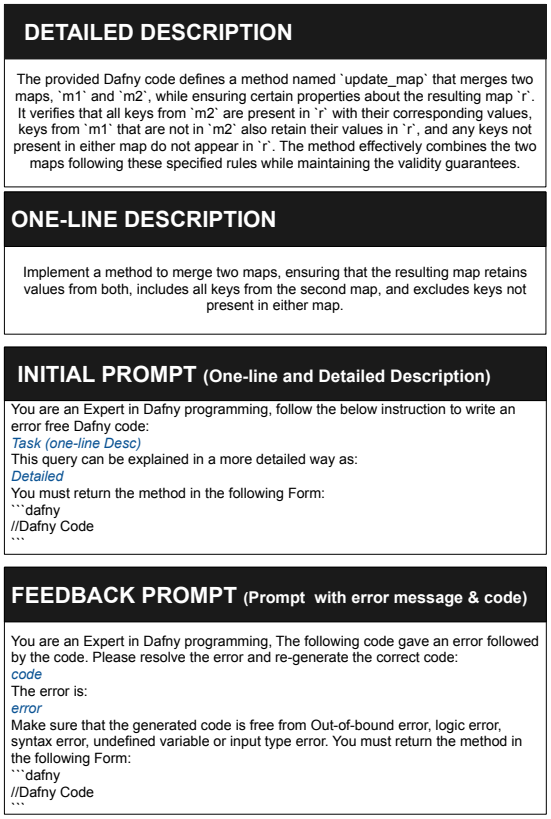


Figure 2: Example prompts and descriptions for Verifier-Feedback Baseline Results.

4 Results and Evaluation

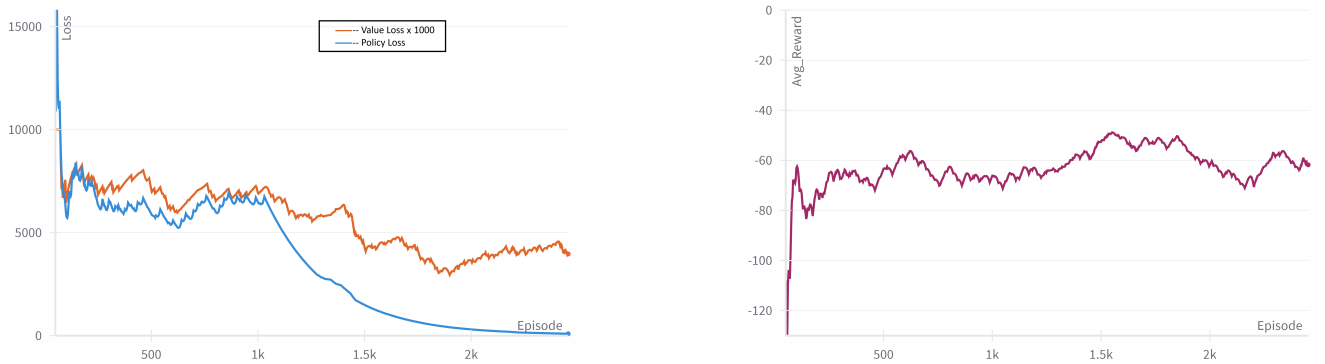
4.1 Verifier-Feedback Baseline Result

We evaluate the baseline performance of several state-of-the-art language models on a representative subset of 100 tasks sampled from DafnyBench [7]. The models tested include ChatGPT-4o [11], ChatGPT-o1 [18], Qwen2.5-Coder-14B [2], Qwen2.5-7B [13] and Gemini-2-Flash. Each model is evaluated under three prompting strategies: a one-line task description, a detailed description, and a concatenation of both. For each type of prompt, we report performance with and without verifier error feedback. Figure 2 shows example prompts and descriptions used for the evaluation of Verifier-Feedback baseline results.

As shown in Table 1, ChatGPT-4o and ChatGPT-o1 perform best, achieving up to 37% success with feedback. Gemini-2-Flash also performs competitively, while Qwen2.5-Coder-14B shows moderate gains with feedback. Qwen2.5-7B performs poorly, failing a lot of tasks due to syntax or logic errors in the generated code.

Table 2: Verification success rates on a 100-task benchmark for various LLMs using prompts through PREFACE pipeline from both untrained and 100-hr-trained small language model, comparing single-shot (W/O Feedback) versus iterative error-feedback (With Feedback) modes. We employ the "One-line and Detailed Description" prompting mechanism (cf. Table 1); values are color-coded green for improvements and red for degradations relative to their respective baseline values.

Model	Without Trained SLM		With Trained SLM	
	W/O Feedback	With Feedback	W/O Feedback	With Feedback
ChatGPT-4o	10% (-15%)	44% (+8%)	23% (-2%)	50% (+14%)
ChatGPT-o1-mini	8% (-14%)	42% (+7%)	23% (±0%)	52% (+17%)
Qwen2.5-Coder-14B	2% (-13%)	12% (-9%)	16% (+1%)	31% (+10%)
Qwen2.5-7B	0% (-3%)	4% (-3%)	3% (±0%)	11% (+4%)
Gemini-2-Flash	3 (-17%)	32 (-1%)	29% (+9%)	55% (+21%)



(a) Value Loss × 1000 and Policy Loss vs. episode

(b) Episode reward vs. episode

Figure 3: Training curves for the PREFACE pipeline over more than 2000 episodes: (a) Value Loss × 1000 and Policy Loss, we scale up the Value loss for a better visualization; (b) Episode Reward, illustrates the pipeline's verification performance improving over time.

These experiments demonstrate that the prompting strategy significantly affects LLM performance, particularly when comparing results with and without verifier feedback.

4.2 PREFACE Results

We ran the PREFACE pipeline continuously for 100 h on 2 NVIDIA H100 GPUs, logging per-episode shaped rewards. Figure 3 plots this trajectory for policy loss and value loss, and reward along the episodes. At the very beginning, the loss in Figure 3a spikes to high values, then drops sharply below 10,000 by episode 200. From episode 200 to around episode 1,000, it plateaus in the 10,000–5,000 range. We use a learning-rate scheduler that reduces the step size at scheduled intervals, and the loss enters a steady decline, approaching zero by the final episodes. This curve illustrates how our scheduler effectively accelerates convergence, steadily improving both policy and value estimates over time.

A similar trend is observed in the reward plot in Figure 3b, where the rewards start very negative but eventually increase, showing success cases within each episode. It is important to note that the plot shows the exponential moving average (EMA) of the rewards for each episode. Our reward function is dominated by per-step penalties and mostly grants positive bonuses after a few steps (when successful verification is achieved), so most raw episode sums—even successful ones—are still negative. The EMA smoothing heavily weights early, attributed to the more consistent negative results at early stages, and thus the positive spikes at later stages cannot pull the running average above zero. Despite this, the clear upward drift, even with occasional verification misses, demonstrates that our RL agent is steadily learning to produce prompts that yield verifiable Dafny code earlier in each episode.

Table 2 reports the verification success of our 100-task benchmark for five LLMs, comparing performance based on prompts generated by an untrained versus a 100 hr-trained small language model (SLM), each evaluated with only SLM-tuned prompt ("W / O Feedback") versus the complete error feedback loop ("With Feedback") based on the PREFACE pipeline. When driven by the untrained SLM, LLM verification success falls below static baselines, a consequence of the SLM's initial inability to craft precise instructions focused on the task that each model needs. Our RL-based training addresses this by teaching the SLM to interpret verifier feedback and error signals, iteratively sharpening its prompt generation policy. The result is the ability to generate contextually rich and detail-sensitive prompts that align with LLM's strengths, yielding substantial improvements in verification performance.

Even before feedback, LLM-optimized prompting by trained SLM yields modest improvements over static without feedback baselines: ChatGPT-4o achieves 23% success (-2%), ChatGPT-o1 23% (±0%), Qwen2.5- Coder-14B 16% (+1%), Qwen2.5-7B 3% (±0%), and Gemini-2-Flash 29% (+9%). Once the verifier feedback loop is enabled, each model shows substantial improvements relative to the static with-feedback baseline: ChatGPT 4o increases to 50% (+ 14%), ChatGPTo1 increases to 52 (+ 17%), Qwen2.5 Coder 14B increases to 31% (+ 10%), Qwen2.5-7B increases to 11% (+ 4%) and Gemini 2 Flash increases to 55% (+ 21%). These pronounced, model-agnostic improvements, especially the near doubling of success for Gemini-2-Flash and significant uplifts for both ChatGPT variants, demonstrate that iterative, error-guided prompt refinement via the PREFACE pipeline robustly amplifies formal verification performance across diverse LLM architectures.

4.3 Implementation Details

All experiments were conducted on a dedicated workstation equipped with two NVIDIA H100 GPUs connected via PCIe Gen4, running Ubuntu 20.04 and Python 3.10. Our framework is implemented in PyTorch 2.1 and Transformers 4.34, with Accelerate for multi-GPU orchestration and BitsAndBytes 0.40 + PEFT 0.6 for 8-bit quantization and LoRA adapters.

We evaluate on the DafnyBench suite— 623 training, and 100 test tasks covering algorithms such as sorting, arithmetic operations, and list manipulations. For our flow, we create a one-line and detailed description for each of the sample. The prompt agent (SLM) is deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B. We quantize the model to 8-bit using BitsAndBytes and shard it across our two H100 GPUs via a balanced device map, with full-precision state dictionaries offloaded to the CPU. Model and adapter weights are checkpointed every 500 episodes to ensure fault tolerance and facilitate ablation studies.

5 Conclusion

We have presented *PREFACE*, a novel reinforcement–learning–driven framework that guides a frozen large language model to produce formally verified Dafny code via prompt adaptation rather than model fine-tuning. By casting prompt refinement as a Markov decision process and employing our Reward-Guided Proximal Optimization algorithm, the small prompt agent learns to interpret rich verifier feedback and iteratively steer generation toward provably correct programs. On the DafnyBench suite, PREFACE yields a dramatic improvement over both naïve prompting baselines and untrained small language model prompting, all while consuming significantly less compute typically required to adjust the base LLM weights.

Beyond these empirical gains, our approach offers a resource-efficient, model-agnostic path to correctness-by-construction code synthesis. PREFACE can be readily extended to other verification back ends, specification languages, or downstream HLS workflows, making it a promising building block for trustworthy AI-assisted software and hardware development. In future work, we will explore automated translation of verified Dafny into optimized hardware descriptions, further bridging the gap between formal methods and large language models.

Acknowledgments

This work is supported by the AMD Center of Excellence grant at UIUC and Semiconductor Research Corporation (SRC) 2023-CT-3175 grant.

References

[1] Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. 2023. Guiding Pretraining in Reinforcement Learning with Large Language Models. arXiv:2302.06692 [cs.LG] https://arxiv.org/abs/2302.06692

[2] Binyuan Hui and Others. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[3] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2024. LLM-based and Retrieval-Augmented Control Code Generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (Lisbon, Portugal) *(LLM4Code '24)*. Association for Computing Machinery, New York, NY, USA, 22–29. doi:10.1145/3643795.3648384

[4] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

[5] Yue Chen Li, Stefan Zetzsche, and Siva Somayyajula. 2025. Dafny as Verification-Aware Intermediate Language for Code Generation. arXiv:2501.06283 [cs.SE] https://arxiv.org/abs/2501.06283

[6] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[7] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. DafnyBench: A Benchmark for Formal Software Verification. arXiv:2406.08467 [cs.SE] https://arxiv.org/abs/2406.08467

[8] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. arXiv:2401.08807 [cs.SE] https://arxiv.org/abs/2401.08807

[9] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE, Article 37 (July 2024), 24 pages. doi:10.1145/3643763

[10] William Murphy, Nikolaus Holzer, Feitong Qiao, Leyi Cui, Raven Rothkopf, Nathan Koenig, and Mark Santolucito. 2024. Combining LLM Code Generation with Formal Specifications and Reactive Program Synthesis. arXiv:2410.19736 [cs.SE] https://arxiv.org/abs/2410.19736

[11] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[12] Gabriel Poesia, Chloe Loughridge, and Nada Amin. 2024. dafny-annotator: AI-Assisted Verification of Dafny Programs. arXiv:2411.15143 [cs.SE] https://arxiv.org/abs/2411.15143

[13] Qwen and Others. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[14] Yi Ren and Danica J. Sutherland. 2025. Learning Dynamics of LLM Finetuning. arXiv:2407.10490 [cs.LG] https://arxiv.org/abs/2407.10490

[15] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98.

[16] Yuan Wei, Xiaohan Shan, and Jianmin Li. 2025. LERO: LLM-driven Evolutionary framework with Hybrid Rewards and Enhanced Observation for Multi-Agent Reinforcement Learning. arXiv:2503.21807 [cs.LG] https://arxiv.org/abs/2503.21807

[17] Sichao Yang and Ye Yang. 2024. FormalEval: A Method for Automatic Evaluation of Code Generation via Large Language Models. In *2024 2nd International Symposium of Electronics Design Automation (ISEDA)*. 660–665. doi:10.1109/ISEDA62518.2024.10617643

[18] Tianyang Zhong and Others. 2024. Evaluation of OpenAI o1: Opportunities and Challenges of AGI. arXiv:2409.18486 [cs.CL] https://arxiv.org/abs/2409.18486

[19] Yuan Zhuang, Yi Shen, Zhili Zhang, Yuxiao Chen, and Fei Miao. 2024. YOLO-MARL: You Only LLM Once for Multi-agent Reinforcement Learning. arXiv:2410.03997 [cs.MA] https://arxiv.org/abs/2410.03997