VeriSEA: Verified Synthesis of Self-Evolving Agents

ANONYMOUS AUTHOR(S)

Recent advances have demonstrated the effectiveness of self-evolving LLM agents on tasks such as program repair and scientific discovery. In this paradigm, a planner LLM synthesizes agent code that invokes parametric models, including probabilistic generative models such as LLMs, smaller neural networks, and external tools such as SMT solvers. These components are then tuned per task to improve performance. However, unlike traditional constraint-guided program synthesis, existing self-evolving agent frameworks provide no formal guarantees of safety or correctness. Because such synthesized programs are often executed autonomously on unseen inputs, the lack of formal guarantees raises serious reliability and security concerns. To address this gap, we formulate agentic code generation as a constrained learning problem that combines hard formal specifications with soft objectives capturing task utility. We introduce Formally Guarded Generative Models (FGGM), which allow the planner LLM to specify a formal output contract for each generative-model call using first-order logic. Each FGGM call automatically wraps the underlying parametric generative model in a rejection sampler with a verified fallback, treating model outputs as samples from a proposal distribution. As a result, every returned output satisfies the specified contract for any input and any parameter setting of the underlying model. Building on FGGM, we present VeriSEA (*Ver*ified *S*elf-*E*volving *A*gents), a three-stage framework for solving constrained learning problems arising from agent synthesis. In *Search*, the planner LLM synthesizes candidate parametric programs that may contain multiple FGGM calls. In *Verification*, we prove correctness with respect to the hard constraints for all parameter values, reducing the problem to unconstrained learning. In *Learning*, we apply scalable gradient-based optimization, including GRPO-style fine-tuning for LLMs, to improve the soft objective while preserving formal correctness. We evaluate VeriSEA on constrained symbolic regression, invariant generation for Dafny programs, symbolic mathematical expression synthesis, and policy-compliant agentic tool use (τ^2-bench). Across all tasks, VeriSEA achieves zero constraint violations while simultaneously improving task performance over unconstrained and state-of-the-art baselines. Our results demonstrate that formal behavioral constraints not only guarantee correctness but also prune the space of candidate programs, steering synthesis toward higher-quality agents.

CCS Concepts: • **LLM Agents** → **Automated Verification**.

Additional Key Words and Phrases: LLM Agents, Automated Verification, Deductive Program Synthesis.

ACM Reference Format:
Anonymous Author(s). 2026. VeriSEA: Verified Synthesis of Self-Evolving Agents. 1, 1 (March 2026), 42 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent works have leveraged the code-generation capabilities of large language models (LLMs) to automatically synthesize LLM-based agents, expressed as programs, from natural language task descriptions [13, 19, 47, 49, 53]. Given a user-defined task, a planner LLM generates a program that invokes parametric models, including LLMs, smaller neural networks, and external tools such as SMT solvers, and executes this program on user inputs to compute outputs. While this paradigm is powerful, it introduces serious safety and reliability concerns: the synthesized programs are executed autonomously on unseen inputs. Modern self-evolving frameworks [11, 43, 48, 51] further

ACM XXXX-XXXX/2026/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

complicate formal safety verification as they fine-tune the parameters of models embedded in the synthesized programs to improve task-specific performance. Consequently, any correctness guarantee provided on an agentic program must continue to hold after parameter updates.

The absence of formal safety verification leads to failures across diverse domains. In program verification, agents cheat by subtly modifying the input program (e.g., altering variable initializations) so that the modified program with proposed annotations passes verification, inflating task accuracy [3]. In code repair, agents delete failing tests rather than fixing the underlying bugs [50]. In agentic tool use, unconstrained agents violate domain-specific policies, such as refund eligibility and booking-modification rules, on 65-76% of interactions [20]. In deployment, rogue agents have been observed bypassing security protections in the wild [14, 15, 25]. These are not isolated incidents; they are a consequence of evaluating synthesized agents solely on soft performance metrics without formal behavioral specifications. Natural-language task descriptions and testing on fixed inputs is not sufficient to completely prevent such failures. Agentic programs need *formal hard constraints* to ensure safety. In this paper, we aim to achieve both the formal guarantees of constraint-guided synthesis [1] and the flexibility and empirical effectiveness of self-evolving agentic frameworks.

Key Challenges: We summarize the challenges below.

- **Balancing Safety and Performance:** A safe agent synthesis algorithm must ensure formal correctness guarantees while simultaneously improving task specific performance. Existing work on deductive program synthesis [1, 22, 38] provides formal guarantees but typically does not optimize task-specific performance objectives. Meanwhile, popular gradient-based optimization methods for LLMs such as GRPO [36] empirically improve output quality but provide no assurance that post-training outputs satisfy specified constraints.
- **Enforcing Constraints per Model Call:** Generative models (GMs), such as LLMs, may be used in an agentic program at different places with different constraints. For example, one call may require generating Python programs, while another may require producing symbolic mathematical expressions conforming to a specific formal grammar. Thus, we require a mechanism to specify formal output constraints for *each* GM call in an agentic program and a way to use these specifications to verify that an agentic program satisfies the user-provided behavioral specification. Recent work on constrained decoding [4, 35, 45] can enforce context-free grammar–based syntactic constraints and limited semantic constraints such as variable naming or simple type checks. However, these approaches are restricted in the types of constraints they support and require modifying the model's decoding procedure, which limits their applicability to open-source models. Moreover, constrained decoding strategies are known to distort the output distribution, producing outputs that satisfy constraints but degrade task performance [4, 35].

Our Contributions: We summarize our contributions below.

- We introduce the novel concept of Formally Guarded Generative Models (FGGM), which allow us to define and enforce local contracts on model calls through formal input-output specifications expressed in first-order logic. FGGM operates solely on LLM output strings and can be applied to both open and closed source models. Using FGGM, we retain the formal guarantees of deductive synthesis while utilizing the performance gained through gradient-based parameter optimization.
- We introduce VeriSEA: the first self-evolving agent synthesis algorithm with verifiable guarantees. The approach consists of three stages: (a) *Search*, where a planner LLM samples program strings in a verifier-aware language such as Dafny, and use the proposed FGGMs to establish local contracts on the embedded model calls; (b) *Verify*, where the language's built-in verifier is used to prove that the sampled programs satisfy all specified contracts over all parameters of the embedded parametric model calls; and (c) *Learn*, where verified parametric programs reduce the constrained

learning problem to an unconstrained optimization problem over model parameters, enabling scalable gradient-based training without sacrificing correctness.

- We prove that VeriSEA is *sound*: any agent returned satisfies the behavioral specification for all inputs and all parameter values (Theorem 5.4). We also establish a *sufficient condition* under which a verified agent exists that satisfies the hard constraints while incurring no greater task loss than any unconstrained generative model with initial parameters, with strict improvement whenever the unconstrained model violates the specification (Theorem 5.5).
- We evaluate VeriSEA on tasks spanning LLM-assisted program verification, symbolic math synthesis, agentic tool use, and constrained symbolic regression. Across all benchmarks, VeriSEA achieves provably zero constraint violations while outperforming state-of-the-art agents in task performance: 97.0% verification rate on HumanEvalDafny (vs. 86.9% for the best baseline), 66.0% accuracy on GSM-Symbolic (vs. 44.7% for the best constrained-decoding method), and 52.6% pass rate τ^2-bench's airline domain using Qwen3-8B. Notably, on τ^2-bench airline VeriSEA even beats Agent-C [20] with Claude Sonnet 4.5, a state-of-the-art constrained agent using a frontier model. These results demonstrate that behavioral constraints do not merely enforce safety but actively prune the search space of candidate programs and steer synthesis toward higher-quality agents.

2 BACKGROUND

Notations and Terminology: We use Σ to denote the alphabet, a finite set of characters. All LLMs output finite strings over Σ. We use lowercase letters (x) to denote constants, bold letters ($\boldsymbol{x}$) to denote strings, capital letters (X) to denote sets (including sets of functions), and $|\boldsymbol{x}|$ to denote string length. In the rest of the paper, we treat neural networks as a restricted form of generative model that produces a single output for a fixed input.

LLMs: LLMs $\mathcal{L} : \Sigma^* \to \Sigma^*$ are probabilistic string generators that, given a prompt $\boldsymbol{p} \in \Sigma^*$ over the alphabet Σ, sample an output string $\boldsymbol{y} \in \Sigma^*$. Typically, the output length $|\boldsymbol{y}|$ is bounded by a fixed user-specified n, i.e., $|\boldsymbol{y}| \leq n$ or equivalently $\boldsymbol{y} \in \Sigma^{\leq n}$. Fixed-length generation $\mathcal{L}_n : \Sigma^* \to \Sigma^{\leq n}$ is implemented by iteratively composing two high-level steps, repeated at most n times. Each iteration samples a single character (sometimes referred to as a token) from Σ or the special end-of-sequence symbol $\langle eos \rangle \notin \Sigma$ that marks termination of generation. The first step is the **distribution prediction step**, defined as $\mathcal{L}^p : \Sigma^* \to \mathcal{P}(\Sigma \cup \langle eos \rangle)$, which maps a prefix string $\boldsymbol{p} \in \Sigma^*$ to a probability distribution over $\Sigma \cup \langle eos \rangle$. This distribution captures how likely each character is to appear next. The second step is the **decoding step**, defined as $\mathcal{D} : \Sigma^* \times \mathcal{P}(\Sigma \cup \langle eos \rangle) \to \Sigma \cup \langle eos \rangle$, which draws the next character using the current prefix $\boldsymbol{p}$ and the predicted distribution. If the decoding step produces $\langle eos \rangle$, the generation terminates. Otherwise, the sampled character c is appended to the prefix, and generation continues. Henceforth, we drop the subscript n for simplicity.

Definition 2.1 (LLMs). For a fixed $n \in \mathbb{N}$, an LLM is a bounded string generator $\mathcal{L}_n : \Sigma^* \to \Sigma^{\leq n}$ that samples a output string $\boldsymbol{y} \in \Sigma^{\leq n}$ for input $\boldsymbol{x}$. $\boldsymbol{y}$ on $\boldsymbol{x}$ is computed recursively as $\boldsymbol{y}_0 = \boldsymbol{x}$, and $\boldsymbol{y}_i = \boldsymbol{y}_{i-1} \cdot c_i$ if $(\forall j \leq i . c_j \neq \langle eos \rangle)$ else $\boldsymbol{y}_{i-1}$ where $i \in [n]$, $c_i = \mathcal{D}(\boldsymbol{y}_{i-1}, \mathcal{L}^p(\boldsymbol{y}_{i-1}))$ and $\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{y}_n$.

Rejection Sampling: Rejection sampling is a popular Monte Carlo method for generating samples from a *target distribution* π_t when direct sampling from π_t is difficult. Instead, it relies on a *proposal distribution* π_p from which sampling is easy. The support set $S(\pi_p)$ (see Definition 2.2) must cover $S(\pi_t)$, i.e., $S(\pi_t) \subseteq S(\pi_p)$. Intuitively, π_p generates candidate samples, and the rejection sampler decides whether to accept each candidate, rejecting those that are unlikely under π_t (details in Appendix B). The rejection sampler guarantees that no accepted sample lies outside $S(\pi_t)$.

Definition 2.2. Let π be a probability distribution over Ω with density function $D_\pi : \Omega \to \mathbb{R}^+$. The support set $S(\pi)$ of π captures all points with positive probability $S(\pi) = \{x \in \Omega; |D_\pi(x) > 0\}$.

Self-evolving LLM agents: In this work, we focus on LLM agents modeled as programs $f : T_i \to T_o$ that, given any user input $x \in T_i$, compute the output $y = f(x)$. Typically, f is written in a popular imperative language such as Python. The program invokes one or more parametric models (formally defined in § 5.1), such as LLMs, as well as other tools such as an SMT solver, to compute the output y. All parametric models $\mathcal{F}_p$ and tool calls $\mathcal{F}_c$ are provided as a collection of pre-implemented library functions. We denote this library by $\mathcal{F} = \mathcal{F}_p \cup \mathcal{F}_c$ (formal definition in § 5.1). In the self-evolving paradigm, the program f is not handwritten. Instead, it is generated by a planner LLM based on task-specific instructions provided as a prompt. The goal is to synthesize a program f that, given a training dataset of input and optional ground-truth output pairs $\mathsf{D} \subseteq T_i \times T_o$ and a loss function $\mathsf{L} : T_i \times T_o \times T_o \to \mathbb{R}^+$, minimizes the aggregated loss $\frac{1}{|\mathsf{D}|} \sum_{(x_i, y_i) \in \mathsf{D}} \mathsf{L}(x_i, y_i, f(x_i))$. For certain tasks, the ground truth y_i may not be available, in such cases, the loss is denoted as $\mathsf{L}(x_i, \_, f(x_i))$. Assume that a context-free grammar (CFG) G defines the set of syntactically valid programs f as $L(G) \subseteq \Sigma^*$. Given the library functions $\mathcal{F}$, let $\mathsf{S}(G, \mathcal{F})$ denote the search space for $f : T_i \to T_o$ with type signature $(T_i \to T_o)$, including all parametric values of the models in $\mathcal{F}_p$. Self-evolving agentic frameworks aim to find the optimal solution f^* to the following optimization problem.

$$f^* = \underset{f \in \mathsf{S}(G,\mathcal{F})}{\arg\min} \; \frac{1}{|\mathsf{D}|} \times \sum_{(x_i, \_) \in \mathsf{D}} \mathsf{L}(x_i, \_, f(x_i)) \tag{1}$$

3 PROBLEM FORMULATION

High-level formulation of agent synthesis with formal constraints: Eq. 1 searches only for the optimal program f^* that minimizes the loss on the training set D. However, it provides no guarantee about how f^* performs on unseen user inputs outside D. In contrast, deductive program synthesis [1, 38] allows program generation to be controlled by formal behavioral input $\Phi : T_i \to \{T, F\}$ and output specifications $\Psi : T_i \times T_o \to \{T, F\}$. The use of formal behavioral specifications ❶ enables proving the correctness of the synthesized program for all inputs that satisfy the input specification, and ❷ can guide the search for f_0 by eliminating unverified program candidates. With (Φ, Ψ), the unconstrained learning problem in Eq. 1 can be reformulated as the constrained learning problem shown below.

$$f^* = \underset{f \in \mathsf{S}(G,\mathcal{F})}{\arg\min} \overbrace{\frac{1}{|\mathsf{D}|} \times \sum_{(x_i, \_) \in \mathsf{D}} \mathsf{L}(x_i, \_, f(x_i))}^{\text{soft learning objective}} \quad \text{s.t.} \quad \overbrace{\forall x \in T_i.\, \Phi(x) \implies \Psi(x, f(x))}^{\text{hard formal constraints}} \tag{2}$$

The central question is whether Eq. 2 can be solved while retaining the scalability of gradient-based methods designed for Eq. 1. Next, we illustrate how practical constraints from several application domains can be encoded as behavioral specifications (Φ, Ψ).

Examples of formal constraints (Φ, Ψ)**:** The output specification Ψ characterizes the agent's expected behavior on any valid input, not just on examples in D. Hard constraints therefore detect and mitigate errors during synthesis that would not be revealed by performance evaluation on a fixed dataset D. We show encodings of (Φ, Ψ) for four different tasks from distinct domains. ⓐ **Scientific Discovery:** We consider constrained symbolic regression, where the task is to recover an unknown mathematical formula from observed data. The formal constraints (Φ, Ψ) encode prior knowledge about the target formula, such as known symbolic bounds [7, 17, 28] or asymptotic behavior [27]. Any generated formula that violates these hard constraints is invalid, regardless of its empirical fit to the data. Moreover, in the presence of noise, the constraints (Φ, Ψ) help prevent overfitting, which cannot be addressed solely by optimizing a soft objective [24]. ⓑ **Program Verification:** Given a program in a verification-aware language such as Dafny with a predefined specification, the agent is expected to synthesize annotations (e.g., loop invariants, assertions, ranking functions) [29, 31, 39]. Task success depends on whether the annotated program verifies. Recent work [3] shows that agents using frontier models (e.g., Claude) may cheat by subtly modifying the input program, such as altering variable initializations, instead of producing correct annotations. [3] reports that such

modifications were not detected by prior string-based checkers, leading to inflated task accuracy. In contrast, formal AST-based diff checkers that syntactically enforce equivalence between the input program and the annotated output serve as hard constraints, ensuring the agent cannot cheat even on unseen inputs. ⓒ **Constrained LLM Generation:** Constrained generation captures a class of problems in which LLMs are required to generate strings that follow a formal structure. Typically, these formal structures are defined using formal grammars (regular [40] or context-free grammars [35, 45]), or static analyzers such as type checkers [32, 33]. In our setting, these formal structures define the output specification Ψ. We consider the task of generating symbolic math expressions from natural language questions, where hard constraints ensure that the synthesized agent always produces at least a structurally valid expression. Overall, hard behavioral constraints capture the minimal requirements that synthesized agents must satisfy on all inputs, thereby enabling the pruning of untrusted candidate programs during agent synthesis. ⓓ **Agentic Tool Use:** Conversational LLM agents deployed in customer-service settings must select and invoke API tools to resolve user requests while respecting domain-specific policies, such as refund eligibility, booking-modification rules, and authentication-before-access requirements [5]. These temporal and logical policy constraints can be expressed as formal specifications that govern the ordering and content of agent actions. Agent-C [20] provides a domain-specific language for specifying such temporal properties and translates them into linear temporal logic (LTL), enabling SMT-based checking of the generated tool calls. The Agent-C checker, given a sequence of tool calls represented as a string, verifies whether the generated sequence complies with the defined formal policy. In our formulation, Ψ encodes these policy-compliance rules, and the Agent-C checker defines the hard constraint, ensuring that synthesized agents never violate domain policies regardless of user input. Overall, these behavioral constraints capture the minimal requirements that synthesized agents must satisfy, enabling the synthesis process to prune untrustworthy candidates.

4 OVERVIEW

Fig. 1 gives a high-level overview of VeriSEA for solving the constrained learning problem in Eq. 2 over the search space $\mathsf{S}(G, \mathcal{F})$. In this setting, we need to simultaneously optimize task-specific performance while guaranteeing that the resulting program provably satisfies the given behavioral constraints. This is difficult because the search space contains both discrete program structure and continuous model parameters, and because although GMs such as LLMs are powerful, they are inherently unreliable, which makes it hard to ensure conformance with behavioral specifications. To address these challenges, VeriSEA retains the best of both worlds: constraint-guided program search from deductive synthesis and unconstrained gradient-based parameter optimization. Specifically, VeriSEA employs a CEGIS-style loop in which a planner LLM proposes candidate parametric programs and a verifier checks their correctness with respect to the behavioral specification. Once a parametric program is verified, VeriSEA applies unconstrained gradient-based optimization to tune the underlying GM parameters and reduce the task-specific loss L. The proposed Formally Guarded Generative Model (FGGM) serves as the critical link between these two strategies. The FGGM setup allows one to bind each GM call to explicit local input–output contracts and a verified non-parametric fallback program, ensuring that these contracts hold regardless of the underlying GM parameters. The verifier then leverages these local contracts to prove the correctness of the parametric program with respect to the behavioral specification. Beyond providing correctness guarantees, these contracts also extract local learning objectives by characterizing the expected outputs of each GM call. This enables gradient-based optimization to improve the GM's conformance to its local contracts, thereby connecting constrained discrete program search with unconstrained learning. **Design Considerations.** Alternative approaches for enforcing formal constraints on generative model calls exist, but they fall short of FGGM. Constrained decoding [4, 33, 35, 45]

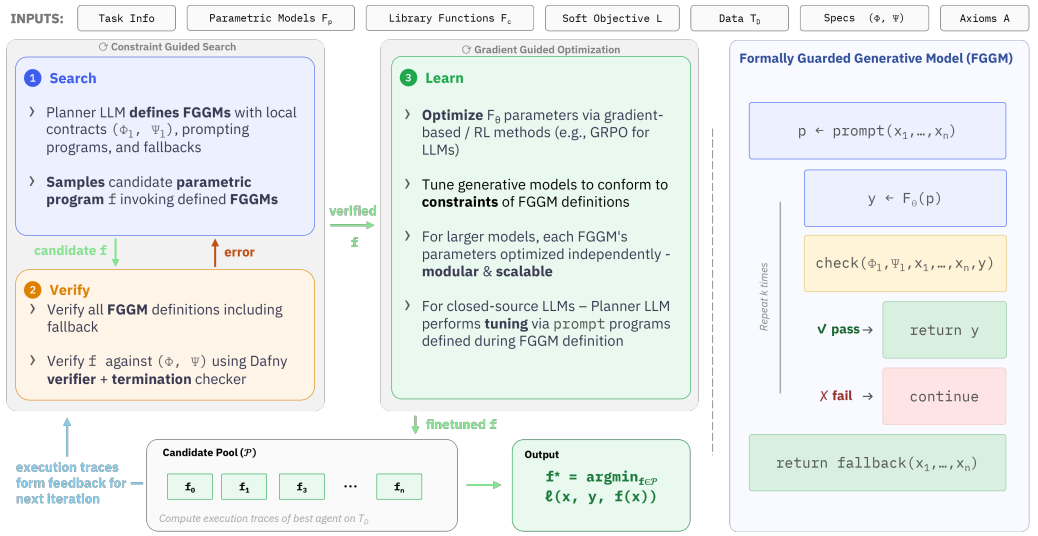

Fig. 1. Overview of VeriSEA, which operates in three key steps. (1) **Search:** Given the task information, library functions $\mathcal{F}_c$, a list of parametric generative models (GMs) $\mathcal{F}_p$, and specifications (Φ, Ψ), the planner LLM outputs a parametric agentic program in which all formal output specifications of GM calls are defined using the FGGM setup. (2) **Verify:** Given the parametric agentic code, this step first verifies all FGGM definitions proposed by the planner. Once verified, it uses their local contracts to check the program against (Φ, Ψ). If all checks pass, the program is accepted and the unconstrained learning step is invoked on the verified program; otherwise, an error message is returned to the planner. (3) **Learn:** This step optimizes the parameters of the underlying GMs within each FGGM to improve conformance with local conformance defined in step (1), while also reducing the task-loss L over the dataset D. After tuning, VeriSEA maintains a pool of verified fine-tuned agents and uses their execution traces on D to generate new candidates via step (1). All agents in the pool are valid candidate solutions, and the one with the lowest L on D is returned. Central to VeriSEA is the Formally Guarded Generative Model (FGGM), which binds each GM call to local input–output contracts (Φ_l, Ψ_l) and a verified non-parametric fallback, ensuring that the contracts hold irrespective of the underlying GM parameters. This enables unconstrained gradient-based parameter optimization without compromising formal correctness.

modifies the model's internal decoding procedure, which limits its applicability to open-source models and restricts constraints to syntactic forms such as context-free grammars. Post-hoc output filtering without a fallback provides no guarantees: if all sampled outputs are rejected, the program produces no valid output. Fully verifying an agentic program is computationally intractable, as it depends on large parametric components such as LLMs. FGGM addresses all three limitations: it operates solely on model outputs, making it compatible with closed-source models; it incorporates a verified fallback that guarantees a valid output on every execution; and it decomposes verification into parameter-independent local contracts. Moreover, these local contracts can be utilized during learning to guide parameter tuning, improving conformance, and reducing reliance on the fallback. Next, we discuss the components of VeriSEA through two representative and easy-to-understand problem instances from different domains: scientific discovery and program verification.

4.1 Example Instantiation of VeriSEA

4.1.1 Constrained Symbolic Regression: In symbolic regression, each problem instance consists of possibly noisy observations of input–output pairs generated by an unknown ground-truth function

f_{gt}. The goal is to recover this ground-truth function from the training data D, a popular task in scientific discovery and neuro-symbolic program synthesis [10, 37]. In constrained symbolic regression [7, 17], users encode known properties of f_{gt} beyond what is captured in D as formal constraints, ensuring that the recovered function f satisfies these constraints. This setup directly follows the constrained learning formulation in Eq. 2, where the synthesized agent f^* corresponds to the recovered function. For illustration, we consider the problem instance with ground-truth function $f_{gt}(x) = \sqrt{1.23 \times \max(x, 0.0)}$, taken from [28]. The dataset D contains input–output pairs $(x, y) \in \mathbb{R}^2$, where $y = f_{gt}(x) + \epsilon$ is a noisy observation of $f_{gt}(x)$ and $\epsilon \sim \mathcal{N}(0, \sigma)$ denotes additive Gaussian noise. The pointwise loss function L is the commonly used Normalized Mean Squared Error (NMSE) (see Eq. 3). For this instance, the agent's type signature τ is $(\mathbb{R} \to \mathbb{R})$, and the behavioral specifications (Φ, Ψ) define known symbolic bounds on f_{gt} (see Eq. 4) where $sqrt : \mathbb{R} \to \mathbb{R}$ is a provided library function.

$$\mathsf{L}(x_i, y_i, f(x_i)) = \frac{(f(x_i) - y_i)^2}{C} \quad \text{where } C = \textstyle\sum_{(x_i, y_i) \in \mathrm{D}} y_i^2 \tag{3}$$

$$\Phi(x) : \ (x \geq 0), \ \Psi(x, f(x)) : \ ((x \leq 1) \implies (f(x) \geq pow(x, 0.8)) \wedge ((x \geq 1) \implies (f(x) \geq sqrt((x))) \tag{4}$$

Search Space: VeriSEA defines the search space of f using a restricted subset of the verification-aware language Dafny. This subset supports four basic types (T): int ($\mathbb{Z}$), bool (T, F), real ($\mathbb{R}$), and str (Σ^*) over the alphabet Σ, as well as conditional blocks, while loops, and function calls. The agent's type signature τ, all variables appearing in first-order formulas, including behavioral specifications (Φ, Ψ), and the contracts of all FGGMs can only use variables of types in T. All concrete values in $\{int, bool, real\}$ can be converted to their string representations in Σ^*, and VeriSEA provides the corresponding conversion functions. For example, $convInt : (x : int) \to str$ converts any integer to its string representation.

Library Functions: For this task, the library function set $\mathcal{F}$ includes common transcendental functions (e.g., sin, cos, pow, $sqrt$) as well as small parametric neural networks (NNs). This ensures that the search space over library function calls can capture a wide range of neuro-symbolic programs. The user provides the type signatures and defines formal contracts through input–output specifications for all library functions, except for the parametric models. The planner LLM defines the formal contracts of each parametric model call through the FGGM setup described in § 4.2. We present the type signature and formal specification of a representative non-parametric and parametric function $NN_{\Theta}^{(n)} : \mathbb{R}^n \to \mathbb{R}$ from $\mathcal{F}$ in Eq. 5, with the complete set provided in Appendix C. As stated in § 2, we treat NNs as a specific instance of generative models (GMs) and handle them no differently from other GMs. Note that all library functions include textual descriptions, which are utilized by the planner LLM during the sampling of parametric programs.

$$sqrt(x : real) \to real, \ \Phi_{sqrt}(x) : (x \geq 0), \ \Psi_{sqrt}(x, sqrt(x)) : (sqrt(x) \geq 0); \ \ NN_{\theta}^{(1)}(x_1 : real) \to real \tag{5}$$

$$\forall x, d_1, d_2 \in \mathbb{R}.(x \geq 1) \wedge (d_1 \geq d_2) \implies pow(x, d_1) \geq pow(x, d_2) \tag{6}$$

$$\forall x, d_1, d_2 \in \mathbb{R}.(x \leq 1) \wedge (d_1 \geq d_2) \implies pow(x, d_1) \leq pow(x, d_2) \tag{7}$$

$$\varphi_{\mathcal{A}} \implies \forall x \in T_i.\big(\Phi(x) \implies \Psi(x, f(x))\big) \quad \text{where } \varphi_{\mathcal{A}} = (\textstyle\bigwedge_{\varphi \in \mathcal{A}} \varphi) \tag{8}$$

Axioms: The axioms $\mathcal{A}$ are a set of universally quantified first-order sentences encoding known properties of the library functions in $\mathcal{F}$. In particular, $\mathcal{A}$ includes the formal contracts of all functions in $\mathcal{F}_c$. VeriSEA also allows users to optionally specify additional known properties of these functions. For example, Eq. 7 shows a representative axiom capturing a property of the power function pow. In all such sentences, library functions are treated as uninterpreted functions. The axiom set assists automated verification of candidate agents, as shown in Eq. 8. We provide the formal syntax for defining $\mathcal{A}$ and the complete axiom set for this task in Appendix D, derived from the popular math

library in LEAN. Note that $\mathcal{A}$ is specific to the library functions and independent of a particular synthesis instance, allowing it to be reused across different tasks that use the same libraries. Like other synthesis frameworks with library functions [12], VeriSEA assumes that all library functions have correct input–output specifications, and that they are pure and terminating. Verifying the correctness of the library functions and axioms is outside the scope of VeriSEA.

4.1.2 LLM-Assisted Automated Verification. This task aims to synthesize annotations, such as inductive loop invariants, ranking functions, and assertions, that enable automatic verification and termination checking of input Dafny programs with pre-encoded input–output specifications [29, 39]. The agent has type signature $(p : str) \to str$, mapping an input Dafny program with pre-encoded specifications, represented as a string, to an annotated Dafny program. The pointwise loss function L evaluates whether the annotated program verifies within a pre-fixed time budget, assigning 1 to unverified programs and 0 to verified ones (Eq. 10). Hard constraints (Φ, Ψ) ensure that, for any parsable input program, the output remains parsable and does not change the input program outside adding annotation (Eq. 11). We use the Dafny built-in verifier to define L, and the Dafny parser together with the AST-based formal diff checker from [3] to define (Φ, Ψ). $noDiff$ returns True if the second input program is syntactically equivalent to the first (ignoring annotations), and False otherwise. The library function set $\mathcal{F}$ includes these components. Although parsing and diff checks could be incorporated into L by penalizing invalid outputs on D, these constraints must hold for all possible inputs, not only those in D. Therefore, any candidate agent that fails to satisfy them must be filtered out.

Library Functions: We use the same search space as constrained symbolic regression. $\mathcal{F}$ includes the verifier with a pre-fixed time budget $verify$, the Dafny parser $parse$, the diff-checker $noDiff$. Additionally, we provide parsers for different individual annotations (e.g. invariants, ranking functions) separately and a function for inserting annotations at a specific location of input code marked by a marker (see Appendix E). We show the type signature and formal contract of representative functions in Eq. 9 while providing the rest in Appendix E.

$$noDiff(\boldsymbol{x} : str, \boldsymbol{y} : str) \to bool,\ \Phi(\boldsymbol{x}, \boldsymbol{y}) : parse(\boldsymbol{x}),\ \Psi(\boldsymbol{x}, \boldsymbol{y}, noDiff(\boldsymbol{x}, \boldsymbol{y})) : T \tag{9}$$

$$\mathsf{L}(\boldsymbol{x}, \_, \boldsymbol{y}) = 1 - \mathbb{I}(parse(\boldsymbol{y}) \wedge noDiff(\boldsymbol{x}, \boldsymbol{y}) \wedge verify(\boldsymbol{y})) \tag{10}$$

$$\Phi(\boldsymbol{x}) : parse(\boldsymbol{x});\ \Psi(\boldsymbol{x}, f(\boldsymbol{x})) : parse(f(\boldsymbol{x})) \wedge noDiff(\boldsymbol{x}, f(\boldsymbol{x})) \tag{11}$$

Axioms: Example of a first-order sentence encoding known properties of $noDiff$ in Eq. 12. $noDiff$ is both reflexive and transitive w.r.t all parsable Dafny programs. Easy to see that same program will always have no difference with itself. We list all the axioms in Appendix F.

$$\textbf{Reflexivity: } \forall \boldsymbol{x} \in \Sigma^*.\ parse(\boldsymbol{x}) \implies noDiff(\boldsymbol{x}, \boldsymbol{x}) \tag{12}$$

4.2 Formally Guarded Generative Model

Formally Guarded Generative Model (FGGM) setup allows the planner LLM to bind each generative model (GM) call to local input–output contracts (Φ_l, Ψ_l) that formally characterize the output. Once the FGGM is verified to be well-formed, the defined contracts can be used to prove the correctness of the agentic program against the behavioral specifications (Φ, Ψ). Our FGGM design ensures the following essential properties. ❶ *Flexibility:* Unlike functions in $\mathcal{F}_c$, output contracts cannot be predefined by users because the expected output of a GM varies across calls depending on the prompt. In our design, the planner LLM synthesizes custom first-order formulas over terms involving non-parametric functions (e.g., sqrt) and predicates (e.g., $noDiff$) from $\mathcal{F}_c$ to specify local contracts. ❷ *Parameter-Independent Correctness:* Provided the defined FGGM is well-formed, the specified contracts hold irrespective of the parameters of the underlying GM. This property is crucial to ensure that gradient-based parameter optimization never breaks the correctness guarantees. ❸ *Local Learning Objective:* Although correctness holds irrespective of parameter values, the local

contracts (Φ_l, Ψ_l) extract the local learning objective for a particular GM call by characterizing its expected output. This objective can guide gradient-based optimization so that, after tuning, the GM's conformance with (Φ_l, Ψ_l) improves (§ 4.2.6).

4.2.1 Intuition. For each FGGM with local contracts (Φ_l, Ψ_l) and underlying parametric GM $\mathcal{L}_\Theta$, the planner LLM synthesizes two non-parametric programs using only functions from $\mathcal{F}_c$. The first is a fallback program f_d that satisfies (Φ_l, Ψ_l) and guards against edge cases in which the GM hallucinates or produces outputs that violate the specification. Without a verified fallback, the rejection sampler could exhaust all samples; the fallback is what makes the contract hold unconditionally for all parameter values. Intuitively, although f_d may lack the GM's task-solving capability (low task performance), it prevents the candidate agent from entering unsafe program states during execution (high constraint satisfaction). The second is a prompting program f_p, which constructs the input to the GM (which may be just a neural network). For LLMs, f_p encodes task-specific natural-language instructions, analogous to prompt-tuning techniques [6, 21] known to improve GM performance and instruction adherence.
Once planner LLM proposes Φ_l, Ψ_l, f_p, and f_d, VeriSEA automatically constructs a rejection sampler that treats outputs of underlying GM as samples from a proposal distribution π_p for an input $\boldsymbol{p}$. The local specifications (Φ_l, Ψ_l) define the support set (Definition 2.2) of the target distribution π_t on $\boldsymbol{p}$, assigning zero probability to outputs that violate (Φ_l, Ψ_l). The rejection sampler draws a fixed number $K \geq 1$ of samples from $\mathcal{L}_\Theta$, as specified by the user, and falls back to f_d if all samples are rejected. The purpose of f_p and optional conformance tuning is to improve the acceptance rate of the rejection sampler for any input, thereby reducing reliance on the fallback f_d.

$$\langle FGGM\rangle ::= \langle id\rangle\ ";"\ \langle GMid\rangle ";"\ \overbrace{\langle typeSig\rangle\ ";"\ \langle \Phi_l\rangle ";"\ \langle \Psi_l\rangle\ ";"}^{\text{signature + formal contract}}\ \underbrace{\langle f_p\rangle ";"}_{\text{prompt}}\ \underbrace{\langle f_d\rangle ";"}_{\text{fallback}}\langle info\rangle \tag{13}$$

4.2.2 FGGM definition. Eq. 13 specifies how the planner LLM defines a guarded GM, where: (a) *id:* is the unique identifier of the created FGGM; (b) *GMid:* identifies the underlying GM used as the proposal distribution; (c) *typeSig:* defines the type signature of the FGGM, including input variables with their types and the output type, e.g., $(x_1 : T_1, \ldots, x_n : T_n) \to T_o$; (d) (Φ_l, Ψ_l)*:* are first-order formulas whose terms can include non-parametric library functions from $\mathcal{F}_c$; (e) f_p*:* is the program that constructs the GM input from the typed input variables and the task description; (f) f_d*:* is a non-parametric fallback program that satisfies the local specifications (Φ_l, Ψ_l), (h) *info:* is a textual description of the defined FGGM, utilized by the planner LLM during the generation of the parametric program. VeriSEA parses these inputs, validates the definition, and automatically synthesizes a rejection sampler with fallback, as shown in Fig. 2 where $\mathcal{L}_\Theta$ is the parametric GM specified by GMid. The defined FGGM is then invoked in the candidate program via its identifier *id*. For valid definitions, VeriSEA synthesizes $check_{\mathcal{A},\Phi_l,\Psi_l}$, which checks whether a concrete input–output pair $(x_1, \ldots, x_n, y)$ satisfies the proposed contract (Φ_l, Ψ_l), as elaborated in § 4.2.4.

```
function id(x_1 : T_1, ..., x_n : T_n) : (T_o)
  requires Φ_l(x_1, ..., x_n) {
  var p : T_in := f_p(x_1, ..., x_n), y : T_o;
  for i = 1 ... K {                                  ┐
    y := L_Θ(p);                                     │
    if check_{A,Φ_l,Ψ_l}(x_1, ..., x_n, y)           │ ❶
      return y;                                      │
  }                                                  ┘
  return f_d(x_1, ..., x_n, y);                      ┐ ❷
}
```

Fig. 2. Auto-synthesized guarded GM with rejection sampler (1) and fallback (2).

We provide the BNF grammar for parsing the FGGM definition in Appendix L. Next, we discuss correctness checks for a proposed FGGM definition. Provided the VeriSEA validates definition including the fallback f_d, $\forall\theta \in \Theta.\forall x_1 \in T_1 \ldots \forall x_n \in T_n.\Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n, id(x_1,\ldots,x_n))$ holds ensuring formal contracts are preserved for all parametric values $\theta \in \Theta$.

4.2.3 Verifying well-formedness of FGGM definitions. Let the parametric GM identified by *GMid* be $\mathcal{L}_\Theta : T_{in} \to T_o$. VeriSEA first verifies that the declared signature *typeSig*, contracts (Φ_l, Ψ_l) are well-formed and that f_p is terminating and type-correct, i.e., $f_p : (T_1,\ldots,T_n) \to T_{in}$, ensuring compatibility with $\mathcal{L}_\Theta$. It then checks that $f_d : (T_1,\ldots,T_n,T_o) \to T_o$ matches the declared output type. The last argument to f_d corresponds to the most recent failed sample from $\mathcal{L}_\Theta$, allowing the fallback to optionally use this sample, as illustrated in Eq. 14. Finally, f_d is automatically checked for termination and verified against the local specifications (Φ_l, Ψ_l), i.e., $\forall x_1 \in T_1 \ldots \forall x_n \in T_n, \forall y \in T_o, \Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n, f_d(x_1,\ldots,x_n,y))$, using the built-in deductive verifier and termination checker. The verifier also incorporates user-specified axioms from $\mathcal{A}$ (Eq. 8). If all checks pass, VeriSEA guarantees that for all $\theta \in \Theta$ and all well-typed inputs satisfying Φ_l, the synthesized FGGM preserves the output contract Ψ_l. We provide formal details in § 5.2.1.

4.2.4 Computability of $check_{\mathcal{A},\Phi_l,\Psi_l}$ (Fig. 2) on concrete values. The auto-synthesized rejection sampler (Fig. 2) requires checking whether a sample y from $\mathcal{L}_\Theta$, given a concrete input $(x_1,\ldots,x_n)$, satisfies the planner LLM–defined Ψ_l. We show that for any *quantifier-free* Ψ_l, this check can be computed efficiently for all concrete values $(x_1,\ldots,x_n,y)$. Since all library functions in $\mathcal{F}_c$ are computable, $check_{\mathcal{A},\Phi_l,\Psi_l}$ can evaluate every term on the concrete inputs $(x_1,\ldots,x_n,y)$. Satisfiability can then be decided in linear time in the number of terms in Ψ_l. We provide formal completeness for the quantifier-free fragment in Lemma 5.1. For Ψ_l with quantifiers, we instantiate $check_{\mathcal{A},\Phi_l,\Psi_l}$ with the axioms $\mathcal{A}$. Then, $check_{\mathcal{A},\Phi_l,\Psi_l}$ substitutes the free variables in Ψ_l with their corresponding concrete values, encodes the axioms $\mathcal{A}$ and Φ_l similarly to Eq. 8, and submits the resulting formula to an SMT solver. First-order formulas over the basic types T and uninterpreted library functions may lie outside decidable fragments. Therefore, we bound the SMT solver with a user-specified timeout. $check_{\mathcal{A},\Phi_l,\Psi_l}$ returns true only if the solver succeeds within the timeout; otherwise, it returns false. This guarantees soundness; however, in the presence of quantifiers, $check_{\mathcal{A},\Phi_l,\Psi_l}$ may be incomplete and may incorrectly reject valid samples y.

4.2.5 Examples. We show a couple of example FGGM definitons that the planner LLM suggests: one from symbolic regression and one from Dafny program verification. The first example from symbolic regression (Eq. 14) with id *boundedParam*, type signature $(l : real, u : real) \to real$, underlying parametric GM which is a neural network $NN^{(2)}_\Theta : \mathbb{R}^2 \to \mathbb{R}$ for this case, $\Phi_l : (l \le u)$ Ψ_l (Eq. 14) restricts $boundedParam(l,u)$ to lie within $[l,u]$. The fallback clamps any violating sample y using library functions *min* and *max*, ensuring contract preservation.
The second example from Dafny program verification (Eq. 15) with id $dafnyAnnotate$, type signature $(p : str) \to str$, underlying parametric GM which is a LLM $\mathcal{L}_\Theta : \Sigma^* \to \Sigma^*$, $\Phi_l : parse(p)$, Ψ_l (Eq. 15) ensures $dafnyAnnotate(p)$ always outputs a syntactically valid program and only adds annotations to the input program p. If all samples are rejected, the fallback safely returns the original program p, guaranteeing Ψ_l under Φ_l utilizing the ***reflexivity axiom*** (Eq 12). These examples show how a planner LLM can define FGGMs with fallbacks to avoid incorrect program states for diverse tasks.

$$\Psi_l : (l \le f(l,u)) \wedge (f(l,u) \le u), \quad f_d : \textbf{function } f_d\text{(l:real, u:real, y:real) \{return } min(max(l,y),u);\} \quad (14)$$

$$\Psi_l : parse(p) \wedge noDiff(p, f(p)), \quad f_d : \textbf{function } f_d\text{(p:str, y:str) : (}str\text{) \{return } p;\} \quad (15)$$

4.2.6 Conformance Tuning. A FGGM that always uses the fallback is formally correct but practically no better than a non-parametric program f_d, conformance tuning helps avoid this by teaching

the GM to satisfy the defined contract directly. The local contracts (Φ_l, Ψ_l) define the local learning objective to avoid the fallback and assign higher probability to outputs that pass the check $check_{\mathcal{A},\Phi_l,\Psi_l}$. The objective in Eq. 16 minimizes the expected contract violation of GM samples over an input set $\mathbb{P}$. For any $\theta \in \Theta$, the inner term $\mathsf{L}^{\theta}_{\Phi_l,\Psi_l}(p)$ measures the probability that a sampled output from $\mathcal{L}_\theta(p)$ violates $check_{\mathcal{A},\Phi_l,\Psi_l}(\Phi_l, \Psi_l, x_1, \ldots, x_n, y)$, while $\mathsf{L}_{\Phi_l,\Psi_l}(\theta)$ aggregates this violation across inputs in $\mathbb{P}$. By optimizing $\theta \in \Theta$ to reduce this loss, the GM post-tuning is encouraged to concentrate the probability mass on the support defined by (Φ_l, Ψ_l) (details in § 5.2.3). Overall conformance tuning improves the acceptance rate of the rejection sampler, reducing reliance on f_d.

$$\underbrace{\mathsf{L}_{\Phi_l,\Psi_l}(\theta) = \frac{1}{|\mathbb{P}|} \times \sum_{p\in\mathbb{P}} \mathsf{L}^{\theta}_{\Phi_l,\Psi_l}(p),}_{\text{Aggregated constraint violation over } \mathbb{P}} \quad \text{where } \mathsf{L}^{\theta}_{\Phi_l,\Psi_l}(p) = \underbrace{\mathbb{E}_{y\sim\mathcal{L}_\theta(p)} 1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y))}_{\text{Expected constraint violation on single GM input } p} \tag{16}$$

4.3 VeriSEA

4.3.1 Search. VeriSEA uses a three-step strategy (search → verify → learn) to solve the constrained learning problem in Eq. 2. In the search step, VeriSEA employs a planner LLM to define FGGMs along with their prompting programs f_p, enabling per-call prompt tuning to discover textual instructions that improve task performance [6, 21]. The planner then samples candidate parametric programs that invoke the defined FGGMs (search step in Fig. 1). Direct calls to parametric GMs in $\mathcal{F}_p$ are disallowed; instead, each such call must be wrapped within a verified FGGM.

4.3.2 Verify. In the verification step, VeriSEA first checks all FGGM definitions. Once validated, their local contracts are used to verify the sampled program against the behavioral specifications (Φ, Ψ) and termination, using the Dafny built-in verifier and termination checker. If verification fails, either due to an invalid FGGM definition or an incorrect candidate program, error feedback is returned to the planner LLM to refine the FGGM or resample a new program (verification step in Fig. 1). Together, the search and verification steps instantiate a popular CEGIS-style [38] loop for discrete search over parametric programs. Due to the FGGM setup, any verified program satisfies the behavioral specifications for all parameter values of the underlying GMs inside the FGGMs. This reduces the constrained learning problem to unconstrained continuous parameter optimization.

4.3.3 Learn. The learning step solves the unconstrained optimization problem using scalable gradient-based methods, without requiring formal guarantees of optimization success. The optimization is guided by the global task-specific loss L and the local conformance loss $\mathsf{L}_{\Phi_l,\Psi_l}$ for each FGGM call. In § 5.2.3, we show that parameters for each FGGM call can be learned independently, ensuring scalability. For GMs such as LLMs, the resulting

```
function agent(x : real) : (real) {
  var a : real := boundedParam(0.0, 1.0);
  var b : real := boundedParam(0.0, 1.0);
  var linear_x : real := a * x;
  var y : real := linear_x + b;
  return y;
}
```

(a) Unverified parametric program pruned out.

```
function agent(x : real) : (real) {
  if (x ≤ 0.0) {return 0.0;}
  var a : real := boundedParam(1.0, 1.5);
  var d : real := boundedParam(0.5, 0.8);
  assert x ≥ 0.0;
  var pow_x : real := pow(x, d);
  assert (d ≤ 0.8);   assert (d ≥ 0.5);
  assert (x ≤ 1) ⟹ (pow_x ≥ pow(x, 0.8));
  assert (x ≥ 1) ⟹ (pow_x ≥ pow(x, 0.5));
  assert (x ≥ 0) ⟹ (pow(x, 0.5) = sqrt(x));
  var y : real := a * pow_x;
  return y;
}
```

(b) Verified parametric program.

Fig. 3. Two example candidate parametric programs generated for symbolic regression.

optimization can be solved using popular RL-based fine-tuning methods such as GRPO [36]. In this setting, the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ acts as a reward function that assigns zero reward to outputs violating the local specification, while the global pointwise loss L rewards the agent's final output. Further details are discussed in § 5.2.3. For closed-source LLMs without accessible parameters, the learning step is skipped. In this case, improvements rely on prompt tuning through prompting functions f_p defined within the FGGMs.

4.3.4 Candidate Pool and Execution Feedback. VeriSEA maintains a pool of verified candidate agentic programs with fine-tuned parameters (Fig. 1). VeriSEA selects the best-performing candidate agent from the pool, i.e., the one with the minimal task loss L, and uses its execution traces on data D to iteratively search for new candidates via the planner LLM. The number of parametric candidate sampling steps is capped by the user. Once this limit is reached, VeriSEA returns the agent in the pool with the lowest task loss L. Fig. 3 shows two candidate parametric programs sampled by the planner LLM. Each program invokes the FGGM defined in Eq. 14 with id *boundedParam* from two different locations. Fig. 3a shows the first parametric program, which fails to verify against the behavioral specification and is therefore pruned by the verifier. This candidate represents the output as a parametric affine function of the input x, which does not match the ground truth, illustrating how strict behavioral constraints eliminate poor candidates. The second candidate verifies for all parameters of the neural networks $NN_\Theta^{(2)}$ wrapped inside the *boundedParam* calls. After parameter tuning, it recovers the ground truth with negligible error, as discussed below. Currently, VeriSEA does not share parameters across FGGM calls at different locations and instead assigns separate parameter sets to each *boundedParam* call. This design allows the system to potentially learn different parameters at different call sites to improve task-specific performance. Since both *boundedParam* calls receive constant inputs, after fine-tuning they return constant values, i.e., $a = 1.11$ and $d = 0.503$. This recovers the ground-truth function $\sqrt{1.23 \times \max(x, 0)}$ with negligible error. Note that VeriSEA currently neither shares parameters nor caps the number of FGGM calls. However, VeriSEA can be extended to support both these checks to reduce the computational budget required for parameter learning, as discussed in § 6.5. The parametric candidate programs and all FGGM definitions with synthesized prompting programs f_p for the program verification task are included in Appendix K.

5 TECHNICAL DETAILS

We formalize the search space in § 5.1, provide the details of the steps search, verify, and learn along with FGGM in § 5.2. We provide formal proof of soundness and a sufficient condition for agent synthesis success with non-trivial utility in § 5.3.

5.1 Search Space

Program Search Space: We use a restricted subset of the popular verification-aware language Dafny [26] to define the search space of candidate programs. In this restricted subset, we allow conditional blocks, while loops, assignment statements, assertion statements, function calls, and annotations including invariant clauses and ranking functions (decreases clauses), along with first-order input–output specifications. We present the BNF grammar G_D capturing this subset in Appendix A. We allow four basic types $\mathsf{T} = \{bool, int, real, string\}$, basic built-in arithmetic functions $\mathcal{B}_a = \{+, -, \times, /\}$, boolean functions $\mathcal{B}_b = \{\&\&, ||, !, \implies\}$, and relations $\mathcal{B}_r = \{==, \leq, \geq, <, >\}$ with their usual semantics. We use a planner LLM $\mathcal{L}_p : \Sigma \to \Sigma$ to search the program space by sampling candidate programs as strings. We assume that $\mathcal{L}_p$'s output space can represent all valid Dafny programs $L(G_D) \subseteq \Sigma$ and that the type *string* can represent all strings Σ^* over Σ. VeriSEA provides built-in predicates $lex_T(x : string) \to bool$ to check whether a string can be parsed as a

specific type $T \in \{bool, int, real\}$. Using Dafny as the language allows us to leverage its built-in verifier to check candidate programs against a first-order input–output specification, as well as its built-in termination checker.

Library Functions: We categorize the set of library functions $\mathcal{F}$ into two disjoint sets: $\mathcal{F}_c$, the set of non-parametric functions, and $\mathcal{F}_p$, the set of parametric functions. $\mathcal{F}_c$ includes all built-in functions. Each user specified non-parametric function of arity n is specified by a tuple $(id(f_n), \tau_{f_n}, {f_n}^{\Phi}, {f_n}^{\Psi})$, where $id(f_n)$ is the function name, $\tau_{f_n} = (T_1 \times \cdots \times T_n \to T_0)$ is the type signature with $T_i \in \mathsf{T}$, and $({f_n}^{\Phi}, {f_n}^{\Psi})$ are first-order input–output specifications defining a formal over-approximate semantics of f_n. The function $f_n : T_1 \times \cdots \times T_n \to T_0$ halts for any inputs $x_1 \in T_1, \ldots, x_n \in T_n$ and computes $r = f_n(x_1, \ldots, x_n)$. The computed output r, together with $(x_1, \ldots, x_n)$, always satisfies $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\ {f_n}^{\Phi}(x_1, \ldots, x_n) \implies {f_n}^{\Psi}(x_1, \ldots, x_n, r)$. $\mathcal{A}$ have all these formal contracts.

Parametric Functions: Each f_n^{Θ} in $\mathcal{F}_p$ is specified by the tuple $(id(f_n^{\Theta}), \tau_{f_n^{\Theta}}, \theta_0, \Theta)$, where $id(f_n^{\Theta})$ denotes the function name, $\tau_{f_n^{\Theta}}$ is the type signature, Θ is a continuous set of parameters, and $\theta_0 \in \Theta$ denotes the default parameters. Here, $f_n^{\Theta} = \{f_n^{\theta} \mid \theta \in \Theta\}$ represents an infinite set of functions with the same type signature $\tau_{f_n^{\Theta}}$, where each individual function $f_n^{\theta} \in f_n^{\Theta}$ is instantiated by a concrete parameter value $\theta \in \Theta$. Although f_n^{Θ} represents a set of functions, we sometimes abuse this notation to denote a single function corresponding to $f_n^{\theta_0}$ with the default parameters.

The input $\mathcal{I} = (\mathcal{F}, \mathcal{A}, \mathsf{L}, \mathsf{D}, \Phi, \Psi, I)$ to VeriSEA includes the library functions $\mathcal{F}$, axioms $\mathcal{A}$, pointwise differentiable loss L, dataset D, behavioral specifications (Φ, Ψ), and textual task information I used by the planner LLM $\mathcal{L}_p$. We use $\mathcal{P}_{\mathsf{C}} : \Sigma^* \to \{T, F\}$ to denote the parser for checking the syntactic validity of input programs, $\mathcal{V}_{\mathsf{C}}[\Phi, \Psi] : \Sigma^* \to \{T, F\}$ to denote the verifier that checks a program against the specifications (Φ, Ψ), and $\mathcal{T}_{\mathsf{C}} : \Sigma^* \to \{T, F\}$ to denote the termination checker. Each of $\mathcal{P}_{\mathsf{C}}$, $\mathcal{V}_{\mathsf{C}}[\Phi, \Psi]$, and $\mathcal{T}_{\mathsf{C}}$ is instantiated with user-provided context $\mathsf{C} = (G_D, \mathcal{F}, \mathcal{A})$. Both $\mathcal{V}_{\mathsf{C}}[\Phi, \Psi]$ and $\mathcal{T}_{\mathsf{C}}$ are restricted to a fixed timeout and return *false* if they cannot verify (Φ, Ψ) or prove termination within the allotted time budget.

5.2 Key Steps

5.2.1 FGGM Well-formedness. Given an FGGM definition $\mathsf{G} = (id_{\mathsf{G}}, f_{\mathsf{G}}^{\Theta}, \tau_{\mathsf{G}}, \Phi_l, \Psi_l, f_p, f_d, info)$ following Eq. 13, this step checks whether the definition is well-formed. As described in § 4.2.3, VeriSEA first checks the well-formedness of the type signature $\tau_{\mathsf{G}} = (T_1 \times \cdots \times T_n) \to T_o$ and the well-formedness of the local contracts (Φ_l, Ψ_l). We then verify that all terms in Φ_l and subsequently in Ψ_l that involve non-parametric functions from $\mathcal{F}_c$ are valid (details in Appendix M). The G definition is considered valid provided both f_p and f_d are type-checked and terminating, and the fallback f_d satisfies the contract $\mathcal{V}_{\mathsf{C}}[\Phi_l, \Psi_l](f_d)$. After the validity check, VeriSEA constructs $check_{\mathcal{A},\Phi_l,\Psi_l} : (T_1 \times \cdots \times T_n \times T_o) \to \{T, F\}$ that checks whether a concrete input–output tuple $(x_1, \ldots, x_n, y)$ satisfies the local output contract Ψ_l (see § 4.2.4). We provide the pseudocode of $check_{\mathcal{A},\Phi_l,\Psi_l}$ along with a soundness proof showing $\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1, \ldots, \forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1, \ldots, x_n) \implies \big(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \implies \Psi_l(x_1, \ldots, x_n, y)\big)\big)$ in Appendix R. Furthermore, for quantifier-free Ψ_l, the checker is complete (Lemma 5.1).

Lemma 5.1 (Completeness of Checker). *For any valid FGGM* G *with quantifier-free* Ψ_l, $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1, \ldots, x_n) \implies \big(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi_l(x_1, \ldots, x_n, y)\big)$.

Proof Sketch. We prove this by structural induction on the quantifier-free formula Ψ_l. *Base case:* If Ψ_l is an atomic predicate over terms built from computable library functions in $\mathcal{F}_c$, then under any concrete substitution $(x_1, \ldots, x_n, y)$, $check_{\mathcal{A},\Phi_l,\Psi_l}$ evaluates the same computable terms and returns the same boolean value as Ψ_l. *Inductive step:* Assume the claim holds for formulas ϕ and ψ. For compound formulas $\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$, $check_{\mathcal{A},\Phi_l,\Psi_l}$ recursively evaluates the

subformulas and applies the corresponding boolean operator, and returns the result. Thus, by induction on the structure of Ψ_l, $check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)$ returns true iff $\Psi_l(x_1,\ldots,x_n,y)$ holds whenever $\Phi_l(x_1,\ldots,x_n)$ holds. Hence, the checker is complete. Formal details in Appendix R. □

For any valid FGGM G, the rejection sampler $id_{\mathrm{G}} : T_1 \times \cdots \times T_n \to T_o$ with fallback synthesized by VeriSEA (Fig. 2) satisfies the defined contract (Φ_l, Ψ_l) for all well-typed inputs $(x_1,\ldots,x_n)$. Crucially the contract preservation holds over all parameters $\theta \in \Theta$ of the underlying parametric model f_{G}^{Θ}. The function id_{G} terminates on all inputs since the number of samples K is finite and f_p, f_d, and $check_{\mathcal{A},\Phi_l,\Psi_l}$ are all terminating. We provide a soundness proof sketch in Theorem 5.2.

Theorem 5.2 (Valid Local contract). *For any valid FGGM* G *with* (Φ_l, Ψ_l) *and parametric GM* $f_{\mathrm{G}}^{\Theta}, \varphi_{\mathcal{A}} \implies \big(\forall\theta \in \Theta, \forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\ \Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n, id_{\mathrm{G}}(x_1,\ldots,x_n))\big)$.

Proof Sketch. Let $r = id_{\mathrm{G}}(x_1,\ldots,x_n)$. Based on the rejection sampler with the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$, the following condition always holds for the output r (Eq. 17). Eq. 18 follows from the validity of the fallback f_d, and Eq. 19 follows from the soundness of the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$. To simplify the notation, we used $(\forall x_i \in T_i)$ to denote $(\forall x_1 \in T_1, \ldots, \forall x_n \in T_n)$.

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r) \vee (r = f_d(x_1,\ldots,x_n))) \quad (17)$$

$$\varphi_{\mathcal{A}} \implies \forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \wedge (r = f_d(x_1,\ldots,x_n)) \implies \Psi_l(x_1,\ldots,x_n,r) \quad (18)$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).(\Phi_l(x_1,\ldots,x_n) \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r)) \implies \Psi_l(x_1,\ldots,x_n,r)) \quad (19)$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n,r)) \quad \text{Using Eq. (17, 18, 19)}$$

We provide the detailed formal proof in Appendix R. □

5.2.2 Search and Verify. The search and verification steps jointly implement a CEGIS-style loop that explores candidate parametric programs while enforcing the behavioral specification. In the **search** phase, the planner LLM $\mathcal{L}_p$ first synthesizes a set of FGGM definitions $\mathcal{G} = \{\mathrm{G}_1,\ldots,\mathrm{G}_m\}$ following Eq. 13. Each FGGM wraps a parametric model with local contracts and a verified fallback. Using these FGGMs as callable functions, the planner then samples a candidate parametric program $\mathrm{P}_{\{\Theta\}}$ where $\{\Theta\} = \{\Theta_1,\ldots,\Theta_k\}$ represents the set of all optimizable parameters in $\mathrm{P}_{\{\Theta\}}$. As mentioned in § 4.3.4, FGGM calls at different program locations do not share parameters and add their own parameter in $\{\Theta\}$. In the verification phase, VeriSEA first checks the well-formedness of every $\mathrm{G}_i \in \mathcal{G}$. If any of the definitions fail to verify it returns an error message. If all definitions are valid, the verification context $\mathrm{C} = (G_D, \mathcal{F}, \mathcal{A})$ is extended with the synthesized FGGMs and their local contracts. The updated context $\mathrm{C}' = (G_D, \mathcal{F} \cup \mathcal{G}, \mathcal{A} \cup \mathcal{A}_{\mathcal{G}})$ adds all FGGMs $\mathcal{G}$ along with $\mathcal{A}_{\mathcal{G}}$ containing local contracts $(\mathrm{G}_i^{\Phi_l}, \mathrm{G}_i^{\Psi_l})$ of each FGGM $\mathrm{G}_i \in \mathcal{G}$. The candidate program $\mathrm{P}_{\{\Theta\}}$ is then checked for syntactic validity, termination, and compliance with the behavioral specification (Φ, Ψ). If verification succeeds, VeriSEA accepts the parametric program $\mathrm{P}_{\{\Theta\}}$ and ensures that $\mathrm{P}_{\{\Theta\}}$ satisfies (Φ, Ψ) over all parameters (Theorem 5.3). Algorithm 1 summarizes the combined search–verification

Algorithm 1 searchVerify

Input: $\mathcal{I} = (\mathcal{F}, \mathcal{A}, \Phi, \Psi, I)$, planner $\mathcal{L}_p$, budget δ
Input: Previous attempts feed-back I'
Output: Verified Parametric Program $\mathrm{P}_{\{\Theta\}}$ else $\perp$
$i \leftarrow 0$, err $\leftarrow \{\}$
while $i < \delta$ **do**
  $\mathcal{G} \leftarrow defineFGGM(\mathcal{L}_p, I, \mathcal{F}, \mathcal{A}, I', \mathrm{err})$
  err $\leftarrow validateFGGM(\mathcal{G}, \mathcal{F}_c, \mathcal{A})$
  if err $\neq \{\}$ **then**
    $i \leftarrow i + 1$; **continue**
  end if
  $\mathrm{P}_{\{\Theta\}} \leftarrow sampleProgram(\mathcal{L}_p, I, \mathcal{F}, \mathcal{A}, I', \mathcal{G}, \mathrm{err})$
  $\mathrm{C} \leftarrow (G_D, \mathcal{F} \cup \mathcal{G}, \mathcal{A} \cup \mathcal{A}_{\mathcal{G}})$
  err $\leftarrow validateProgram(\mathrm{P}_{\{\Theta\}}, \mathcal{P}_{\mathrm{C}}, \mathcal{T}_{\mathrm{C}}, \mathcal{V}_{\mathrm{C}}[\Phi, \Psi])$
  if err $= \{\}$ **then**
    return $\mathrm{P}_{\{\Theta\}}$
  end if
  $i \leftarrow i + 1$
end while
return $\perp$

loop. Given the input specification $\mathcal{I}$ and planner $\mathcal{L}_p$, the algorithm iteratively proposes candidate solutions within a fixed budget δ (Lines 1–4). In each iteration, the planner first synthesizes a set of FGGM definitions $\mathcal{G}$ (Line 6), which are then checked for well-formedness using *validateFGGM* (Line 7). Within $\mathcal{G}$, the planner LLM $\mathcal{L}_p$ also synthesizes the prompting functions f_p, enabling each FGGM-specific customization. If the definitions are valid, the planner samples a candidate parametric program $\mathsf{P}_{\{\Theta\}}$ that may invoke these FGGMs (Line 11), and the verification context is extended with their contracts (Line 12). The candidate program is then validated for syntactic correctness, termination, and compliance with the behavioral specification using *validateProgram* (Line 13). If verification succeeds, the program is returned (Lines 14–15); otherwise, the loop continues with the error feedback until the search budget is exhausted (Lines 16–18).

Theorem 5.3. *If* $\mathsf{P}_{\{\Theta\}} \neq \bot$ *with FGGM set* $\mathcal{G}$*, then* $\forall\theta_1 \in \Theta_1, \ldots, \forall\theta_k \in \Theta_k.\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1, \ldots, x_n))\big)$.

Proof Sketch. Follows from Theorem 5.2 with details in Appendix R. □

5.2.3 Learn. Once a candidate parametric program $\mathsf{P}_{\{\Theta\}}$ passes the search–verification loop (Algorithm 1), the verification step guarantees that $\mathsf{P}_{\{\Theta\}}$ satisfies the behavioral specification (Φ, Ψ) for *all* parameter values (Theorem 5.3). Hence, the learning step can freely select parameters without violating (Φ, Ψ). Finding the optimal solution to the general formulation in Eq. 20 over $\{\Theta\}$ for a general loss L is practically intractable. Instead, VeriSEA employs a scalable gradient-descent–based optimization to efficiently search over $\{\Theta\}$.

$$(\theta_1{}^*, \ldots, \theta_k{}^*) = \underset{(\theta_1,\ldots,\theta_k)\in\{\Theta\}}{\arg\min} \sum_{(x,y)\in \mathrm{D}} \mathsf{L}(x, y, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x)) \tag{20}$$

Conformance loss $\mathsf{L}_{\Phi_l,\Psi_l}$ for each local contract (Φ_l, Ψ_l) encourages the parameters to produce outputs that satisfy their local contracts. By improving conformance with these contracts, VeriSEA increases the success rate of the corresponding rejection sampler and thereby avoids triggering the static non-parametric fallback program. Since the fallback cannot be tuned, reducing its usage allows the gradient-based search to more effectively guide the parameters toward solutions that reduce the final loss L. Our experiments in § 6.3 substantiate this observation, showing that reducing reliance on the fallback improves task performance. The augmented objective, which includes a local conformance loss for each FGGM call, is defined in Eq. 21, where $\lambda \in \mathbb{R}^+$ is a user-provided constant.

$$\sum_{(x,y)\in \mathrm{D}} \mathsf{L}(x, y, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x)) \;+\; \lambda \times \sum_{i\in k,(x,y)\in \mathrm{D}} \mathsf{L}^i_{\Phi_l,\Psi_l}(\theta_i) \tag{21}$$

Here, the first term is the task-specific loss over the dataset D, and the second term aggregates the local conformance losses (Eq. 16) of the FGGMs appearing in the program. Optimizing all parameters jointly according to Eq. 21, while feasible for smaller neural networks, becomes practically expensive for agents with multiple interdependent LLM calls. To address this challenge, we leverage the fact that the conformance loss for each FGGM depends only on its local inputs and outputs. As a result, it can be optimized independently for each FGGM call. Following this observation, for larger models we approximate the augmented loss by optimizing each FGGM id_{θ_i} with parameter θ_i independently, as shown in Eq. 22. Here, the first term includes the task-specific loss L only when the output y_l of id_{θ_i} matches the final output y of the agent on input x, where $(x, \_) \in \mathrm{D}$. The modularized loss (Eq. 22) enables efficient parallelized training across different θ_i.

$$\mathsf{L}_a(\theta_i) = \tfrac{1}{|\mathbb{P}|} \times \textstyle\sum_{p\in\mathbb{P}} \mathsf{L}_a^{\theta_i}(p); \quad \mathsf{L}_a^{\theta_i}(p) = \mathbb{E}_{y_l\sim id_{\theta_i}(p)} \mathsf{L}(x, \_, y) \times \mathbb{I}(y = y_l) + \lambda \times \mathsf{L}^{\theta_i}_{\Phi_l,\Psi_l}(p) \tag{22}$$

$$\mathcal{R}(p, y_l) = 1 - \mathit{Sigmoid}\big(\mathsf{L}(x, \_, y) \times \mathbb{I}(y = y_l) + \lambda \times (1 - \mathbb{I}(\mathit{check}_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l)))\big) \tag{23}$$

For generative models with accessible parameters, the objective in Eq. 22 can be optimized using reinforcement learning–based fine-tuning methods such as GRPO [36] (details in Appendix N). Eq. 23 defines the reward for an input–output pair (p, y_l), with the goal of learning an output

distribution over y_l for inputs $p \in \mathbb{P}$ that maximizes total reward $\mathcal{R}(p, y_l)$. The reward function need not be differentiable w.r.t θ_i, as long as the outputs y_l remain differentiable w.r.t θ_i [36].

5.2.4 VeriSEA. Algorithm 2 summarizes the overall workflow of VeriSEA. The algorithm maintains a pool $\mathcal{P}$ of verified candidate agents and iteratively improves it through a search–verify–learn loop. In each iteration, VeriSEA first invokes the searchVerify procedure (Algorithm 1), which performs CEGIS-style discrete search to synthesize FGGM definitions and candidate parametric program $\mathsf{P}_{\{\Theta\}}$ that satisfy the behavioral specification (Φ, Ψ). If verification succeeds, the program is passed to the LEARN procedure, which optimizes the parameters of the underlying generative models using the dataset D and task loss L. For closed-source models with inaccessible parameters, this step is skipped, and performance improvements rely solely on prompt tuning through the synthesized f_p programs in each FGGM. The resulting tuned agent is added to the candidate pool, and its execution traces on D are used to construct feedback I' (see Appendix P) that guides the planner in subsequent search iterations. After the search budget Δ is exhausted, VeriSEA returns the agent in the pool that achieves the lowest task loss on the dataset. If $\mathcal{P}$ is empty, VeriSEA returns $\perp$.

Algorithm 2 VeriSEA

Input: $\mathcal{I} = (\mathcal{F}, \mathcal{A}, \mathsf{L}, \mathsf{D}, \Phi, \Psi, I)$, planner $\mathcal{L}_p$
Input: total budget Δ, budget per candidate δ
Output: Best agent f^* or $\perp$ if search fails
$\mathcal{P} \leftarrow \{\}$ ▷ Pool of verified candidates
$I' \leftarrow \emptyset$
while $\delta \leq \Delta$ **do**
  $\mathsf{P}_{\{\Theta\}} \leftarrow searchVerify(\mathcal{F}, \mathcal{A}, \Phi, \Psi, I, \mathcal{L}_p, \delta, I')$
  $\Delta \leftarrow \Delta - \delta$
  if $\mathsf{P}_{\{\Theta\}} = \perp$ **then**
    continue
  end if
  if $\{\Theta\} \neq \{\}$ **then**
    $(\theta_1{}^*, \ldots, \theta_k{}^*) \leftarrow learn(\mathsf{P}_{\{\Theta\}}, \mathsf{L}, \mathsf{D})$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}_{\{(\theta_1{}^*, \ldots, \theta_k{}^*)/\Theta\}}\}$
  else
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}_{\{\Theta\}}\}$
  end if
  $I' \leftarrow collectFeedback(\mathsf{P}_{\{\Theta\}}, \mathsf{D})$
end while
$f^* \leftarrow \arg\min_{f \in \mathcal{P}} \sum_{(x,y) \in \mathsf{D}} \mathsf{L}(x, y, f(x))$
return f^*

5.3 Theoretical Results

The soundness of VeriSEA guarantees that any agent program $f^* \neq \perp$ returned satisfies the behavioral specifications (Φ, Ψ). The searchVerify step generates candidate programs $\mathsf{P}_{\{\Theta\}}$ that satisfy (Φ, Ψ) for all $(\theta_1, \ldots, \theta_k) \in \{\Theta\}$. Consequently, the parameters $(\theta_1{}^*, \ldots, \theta_k{}^*)$ predicted by the learning step preserve constraint satisfaction. The pool $\mathcal{P}$ is either empty or contains only programs that satisfy the behavioral specifications.

THEOREM 5.4 (SOUNDNESS). *If* $f^* \neq \perp$ *then,* $\varphi_{\mathcal{A}} \implies \forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, f^*(x_1, \ldots, x_n))$.

PROOF SKETCH. $(f^* \neq \perp) \implies (f^* \in \mathcal{P})$ and all program $f \in \mathcal{P}$ satisfies $\varphi_{\mathcal{A}} \implies \forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$ following Theorem 5.3. We provide the formal details in Appendix R. □

Next, we characterize a sufficient condition that ensures the existence of a candidate solution $f \in \mathsf{S}(G, \mathcal{F})$ that satisfies (Φ, Ψ) and achieves no worse loss L than any generative model $f_n^\theta \in \mathcal{F}_p$ with any prompting function $f_p \in \mathsf{S}(G, \mathcal{F}_c)$ and initial parameters θ_0. The high-level idea is as follows. If there exists a non-parametric program $f_d \in \mathsf{S}(G, \mathcal{F}_c)$ that satisfies (Φ, Ψ), we can use it as a fallback together with any type-correct generative model f_n^θ (i.e., with output type T_o) to construct an FGGM. A program that invokes this FGGM on the input and returns its output satisfies (Φ, Ψ). Moreover, if $check_{\mathcal{A},\Phi,\Psi}$ is complete and does not incorrectly reject valid samples from f_n^θ, then the FGGM always returns valid samples from f_n^θ. Lemma 5.1 shows that $check_{\mathcal{A},\Phi,\Psi}$ is complete when

Ψ is quantifier-free. Assume: (i) L penalizes constraint-violating outputs more than constraint-satisfying ones, i.e., $\forall x \in T_i,\ \forall y, y' \in T_o.\ (\Phi(x) \wedge \neg\Psi(x,y) \wedge \Psi(x,y')) \implies \mathsf{L}(x,\_,y') < \mathsf{L}(x,\_,y)$, (ii) Ψ is quantifier-free, and (iii) there exists a valid non-parametric program f_d satisfying (Φ,Ψ). Then there exists a program $f \in \mathsf{S}(G,\mathcal{F})$ that satisfies (Φ,Ψ) and incurs no greater loss than any type-correct generative model $f_n^\theta : T_i \to T_o$ with initial parameters. Moreover, if $f_n^\theta(x)$ violates $\Psi(x, f_n^\theta(x))$ for some $(x,\_) \in \mathrm{D}$, the improvement in loss is strict.

Theorem 5.5 (Sufficient Success Condition). *If (i) L penalizes constraint violations, i.e.,$\forall x \in T_i,\ \forall y, y' \in T_o.\ (\Phi(x) \wedge \neg\Psi(x,y) \wedge \Psi(x,y')) \implies \mathsf{L}(x,\_,y') < \mathsf{L}(x,\_,y)$,, (ii) Ψ is quantifier-free, (iii) there exists a non-parametric program $f_d \in \mathsf{S}(G,\mathcal{F}_c)$ such that f_d satisfies (Φ,Ψ). Then, for any type-correct generative model $f_n^\theta : T_i \to T_o$ with initial parameters and any prompting program f_p, there exists a program $f \in \mathsf{S}(G,\mathcal{F})$ such that: (1) f satisfies (Φ,Ψ), and (2) $\mathsf{L}(f) \le \mathsf{L}(f_n^\theta)$. Moreover, if there exists $(x,\_) \in \mathrm{D}$ such that $\neg\Psi(x, f_n^\theta(x))$, then $\mathsf{L}(f) < \mathsf{L}(f_n^\theta)$ where $L(f) = \sum_{(x,\_)\in \mathrm{D}} \mathsf{L}(x,\_,f(x))$.*

Proof Sketch. Define $\mathrm{G} = (id, f_n^\theta, \tau, \Phi_l, \Psi_l, f_p, f_d, info)$ where $(\Phi_l,\Psi_l) := (\Phi,\Psi)$. Since f_d satisfies (Φ,Ψ), the FGGM is valid. Define the program:

$$\textbf{function}\ f(x_1 : T_1, \ldots, x_n : T_n) : T_o\{\textbf{return}\ id(x_1,\ldots,x_n);\}$$

For any input satisfying Φ, by completeness of $check_{\mathcal{A},\Phi_l,\Psi_l}$ (Lemma 5.1), id returns $f_n^\theta(x)$ whenever $\Psi(x, f_n^\theta(x))$ holds, and otherwise returns $f_d(x)$, which satisfies Ψ. For each $(x,\_) \in \mathrm{D}$, $f(x)$ either equals $f_n^\theta(x)$ (if valid) or $f_d(x)$. By assumption (i), such replacement cannot increase loss, so $\mathsf{L}(f) \le \mathsf{L}(f_n^\theta)$. If f_n^θ violates Ψ on some $(x,\_) \in \mathrm{D}$, the improvement is strict. Formal details in Appendix R. □

6 EVALUATION

We evaluate VeriSEA on four tasks spanning scientific discovery, program verification, mathematical reasoning, and agentic tool use. The evaluation seeks to answer three main questions: whether formal constraints improve safety, whether learning remains effective under those constraints, and how conformance tuning with local constraints improves performance.

6.1 Experimental Setup

6.1.1 Tasks and Datasets. We instantiate VeriSEA on four tasks, each representative of a different kind of constrained agent synthesis problem.

LLM-Assisted Program Verification (DafnyBench). The agent receives a Dafny program with pre-encoded input-output specifications and must synthesize annotations that enable verification against the pre-encoded specification inside the input program [29]. The behavioral specification of the agent requires the output to remain parsable and equivalent to the input modulo added annotations, while task performance is measured by verification success within a fixed time budget. The underlying LLM is Claude Sonnet 4.5 [2], a closed-source model whose weights are not accessible; consequently, the learning step (i.e., parameter tuning) is not applicable to this task.

Symbolic Math Synthesis (GSM-Symbolic). GSM-Symbolic is a benchmark of grade-school math word problems generated from symbolic templates [30]. We use it to evaluate the LLM's ability to synthesize correct symbolic mathematical expressions. Behavioral specification of the agent enforces that each generated expression is syntactically valid with respect to a formal grammar, and task performance measures whether the answer is equivalent to the ground-truth expression.

Agentic Tool Use (τ^2-bench). τ^2-bench [5] evaluates conversational LLM agents in realistic customer-service scenarios. We use the retail and airlines domains. The agent must select and invoke API tools to resolve user requests while respecting domain-specific policies. Hard constraints, enforced via the Agent-C checker [20], encode temporal policy-compliance rules, such as refund

eligibility and booking-modification policies, that must hold for all possible user requests. Task performance measures pass rate as a percentage of interactions where the agent successfully resolves the user request.

Constrained Symbolic Regression (SymReg). In symbolic regression, each synthesis instance consists of noisy input–output observations generated by an unknown ground-truth function f_{gt}, along with a behavioral specification encoding prior knowledge about f_{gt}. The goal is to synthesize a neuro-symbolic program (i.e., the agent) that fits the observed input–output pairs while provably satisfying the behavioral specifications. We consider a total of 35 synthesis tasks, each with 2 different training sets corresponding to varying noise levels (5%, 10%), and behavioral specifications that encode symbolic bounds on the ground truth. Each training set includes 600 noisy samples and 400 test samples. All synthesis instances are drawn from popular benchmarks [7, 17].

6.1.2 Implementation Details. All experiments are conducted on NVIDIA A100 GPUs (40 GB). Across all tasks we use the same high-level pipeline: planner-guided program search with a budget of $\Delta = 10$ iterations, FGGM-based rejection sampling with a fixed sample budget $K = 5$, deductive verification with bounded verifier and SMT timeouts, and hyperparameter tuning when model parameters are accessible. The planner LLM is provided with 3 ground-truth examples as in-context examples (2 for τ^2-bench, due to the long traces); these examples are excluded from all train and test splits. The full planner prompt template is given in Appendix Q, and details on how execution feedback is constructed and provided to the planner between iterations are given in Appendix P.

6.1.3 Learning Implementation. Once a program verifies, VeriSEA uses Dafny's built-in transpiler [9] to translate the verified code into Python. Parameter tuning and experiments are then conducted on the transpiled code. VeriSEA assumes that Dafny's transpiler is sound for this restricted subset of the language.

6.1.4 Metrics. For each task, we report: (a) the *constraint violation rate* on a held-out test set of unseen inputs, measuring the fraction of outputs that violate the behavioral specification (Φ, Ψ); (b) *task performance*, measured by mean squared error for SymReg, verification success rate for Dafny, answer accuracy for GSM-Symbolic, and pass rate for τ^2-Bench; and (c) *wall-clock time*.

6.1.5 Baseline. For all four tasks outside GSM-symbolic, we consider state-of-the-art (SOTA) hand-crafted agents. For GSM-Symbolic, we use the SOTA constrained decoder [4]. Some hand-crafted agents (e.g., Agent-C on τ^2-Bench) manually enforce hard constraints; however, being static, they achieve lower task performance. We also consider self-evolving frameworks without constraints, which do not verify the generated programs against behavioral specifications and accept all planner programs provided they are syntactically valid and type-checked. Consequently, such frameworks provide no guarantees about the correctness or safety of the generated code. To ensure fairness, unconstrained self-evolving frameworks are only allowed to use the same set of library functions.

6.1.6 Verification Success. For program verification, tool calling, and GSM-Symbolic, VeriSEA consistently finds a valid program satisfying (Φ, Ψ) within 10 attempts. For symbolic regression, it produces a verified program in 33 out of 35 cases, whereas the baselines generate at least 11 cases where the generated program violates the behavioral specification on test inputs.

Dataset	Total	Train
HumanEvalDafny	135	30
DafnyBench	760	50
τ^2-bench Retail	114	15
τ^2-bench Airline	50	10
GSM-Symbolic	100	50
Symbolic Regression	600	400

Table 1. Dataset sizes and train/test splits.

6.1.7 Dataset splits. Table 1 summarizes the dataset sizes for each synthesis task. For each task, a small training set is used for the learning step and the remaining examples form the held-out test set

Table 2. Dafny program verification results on HumanEvalDafny and DafnyBench. *Verif. & NoDiff* counts a program as correct only if it both verifies and passes the AST-based diff check against the original; *Verif.* counts verification alone.

Dataset	Method	Ver. & NoDiff (%) ↑	Ver. (%) ↑	Vio. (%) ↓	Time (s)
HumanEvalDafny	LLM (Claude Sonnet 4.5)	73.7	76.8	8.1	9.8
	DafnyBench baseline	86.9	87.9	4.0	16.1
	VeriSEA (w/o constraints)	84.8	88.9	5.1	15.7
	VeriSEA	97.0 (+10.1)	97.0 (+9.1)	0.0 (-4.0)	18.2
DafnyBench	LLM (Claude Sonnet 4.5)	68.7	71.1	10.3	10.3
	DafnyBench baseline	81.6	84.0	8.2	20.1
	VeriSEA (w/o constraints)	79.2	84.8	7.9	18.4
	VeriSEA	89.1 (+7.5)	89.1 (+5.1)	0.0 (-8.2)	25.6

on which all reported metrics are computed. GSM-Symbolic uses a larger training fraction (50%) because its learning step performs GRPO-based fine-tuning, which requires more training data than the other tasks.

6.1.8 Task-specific details. For Dafny, the underlying LLM is Claude Sonnet 4.5 [2], a closed-source model; because model weights are inaccessible, Dafny experiments use only the search and verification stages of VeriSEA and do not perform parameter tuning. For τ^2-bench, parameter tuning is also omitted because the long, multi-turn execution traces make gradient-based fine-tuning prohibitively expensive. For GSM-Symbolic, the learning step uses GRPO [36] with LoRA adapters [18] on Qwen3-8B [42]. Additional hyperparameters (LoRA rank, solver timeouts, etc.) are reported in Appendix O.

6.2 Main Results

We begin with the central question motivating VeriSEA:

> **RQ1 (Safety).** *What goes wrong without formal constraints, and does VeriSEA eliminate these failures?*

We compare unconstrained agent synthesis against VeriSEA across all four tasks to show that hard behavioral specifications are necessary: without them, agents silently produce invalid outputs on unseen inputs that are undetectable by soft performance metrics.

6.2.1 LLM-Assisted Program Verification (Dafny). Table 2 presents results on two Dafny program-verification benchmarks and offers the clearest illustration of why formal constraints are indispensable. The gap between the *Verif.* and *Verif. & NoDiff* columns reveals the core safety issue (**RQ1**): without hard constraints, agents inflate their verification rate by modifying the original program. The LLM baseline reports 76.8% verification on HumanEvalDafny, yet only 73.7% of those outputs actually preserve the input program. Overall, 8.1% of all outputs are violations that would go undetected by a metric that checks verification alone. The DafnyBench baseline and VeriSEA without constraints reduce but do not eliminate this problem (4.0% and 5.1% violations, respectively). Only the full VeriSEA pipeline, which enforces the *NoDiff* behavioral specification, achieves a 0% violation rate on both benchmarks while simultaneously attaining the highest Verif. & NoDiff rate (97.0% on HumanEvalDafny, 89.1% on DafnyBench). This confirms that FGGM-based rejection sampling with a verified fallback does not merely filter outputs but actively steers the planner toward higher-quality candidates. This result is particularly notable because it is achieved without any parameter tuning (the underlying model, Claude Sonnet 4.5, is closed-source), the

Table 3. τ^2-bench results with Qwen3-8B on the retail and airline domains. Higher pass rate is better; lower violation rate is better.

Domain	Method	Pass Rate (%) ↑	Violation (%) ↓	Time (s)
Retail	LLM (Qwen3-8B)	11.3	76.3	146.6
	Agent-C (Qwen3-8B)	42.2	0.0	234.7
	VeriSEA (w/o constraints)	49.4	10.3	238.3
	VeriSEA	53.6 (+11.4)	0.0 (0.0)	212.4
Airline	LLM (Qwen3-8B)	13.2	68.4	184.8
	Agent-C (Qwen3-8B)	39.4	0.0	272.6
	VeriSEA (w/o constraints)	44.7	25.5	268.4
	VeriSEA	52.6 (+13.2)	0.0 (0.0)	241.1

Table 4. GSM-Symbolic results with Qwen3-8B. Higher accuracy is better; lower violation rate is better.

Method	Accuracy (%) ↑	Violation (%) ↓	Time (s)
LLM (Qwen3-8B)	38.3	10.6	10.9
CRANE [4]	44.7	2.1	12.4
VeriSEA (no parameter tuning)	53.2 (+8.5)	0.0 (-2.1)	18.8
VeriSEA (with parameter tuning)	66.0 (+21.3)	0.0 (-2.1)	16.7

gains come entirely from making constraints visible to the planner and enforcing them at synthesis time. The overhead introduced by VeriSEA is modest. The full pipeline takes 18.2 s per instance on HumanEvalDafny compared to 9.8 s for LLM generation, roughly a 1.9× factor that includes Dafny verification and up to k LLM calls. On DafnyBench the factor is 2.5× (25.6 s vs. 10.3 s). Given that the LLM baseline produces untrustworthy outputs on 8 to 10% of inputs, we believe this overhead is justified for the formal guarantees it provides.

6.2.2 Agentic Tool Use (τ^2-bench). Table 3 reports performance on τ^2-bench. The LLM baseline (Qwen3-8B without constraints) violates domain policies on 76.3% of retail interactions and 68.4% of airline interactions, yielding pass rates of only 11.3% and 13.2%, respectively. Both Agent-C and VeriSEA reduce the violation rate to 0%, confirming that formal policy enforcement is necessary. VeriSEA without constraints illustrates an intermediate point: by running the search loop without informing the planner of the policy specification, VeriSEA (w/o constraints) improves pass rates to 49.4% (retail) and 44.7% (airline), but still incurs 10.3% and 25.5% violation rates, respectively. Enforcing constraints eliminates all violations while further improving pass rates to 53.6% and 52.6%. This gap reflects VeriSEA's search-verify-learn design. By synthesizing agent programs who verifiable follow policy specifications, VeriSEA can explore a wider space of compliant strategies rather than relying solely on runtime constraint checking. Impressively, VeriSEA with the small open-weight Qwen3-8B outperforms Agent-C [20] with Claude Sonnet 4.5 (52.6 vs 47.3).
VeriSEA is also faster than Agent-C in both domains (212.4s vs. 234.7s in retail and 241.1s vs. 272.6s in airline). The LLM baseline is fastest because it performs no policy checking, but its outputs are unusable in practice given the violation rates.

6.2.3 Symbolic Math Synthesis (GSM-Symbolic). GSM-Symbolic uses Qwen3-8B as the underlying LLM. Qwen3-8B is open-weight, making parameter tuning feasible. This allows us to ask:

> **RQ2 (Training Impact).** *Does parameter tuning improve task performance under formal constraints?*

Table 5. % of synthesised instances where the synthesized program violates behavioral specifications on the test set.

Method	ϵ (5%) ↓	ϵ (10%) ↓
PySR	62.86%	62.86%
LLM-SR	31.43%	34.29%
VeriSEA	0.00%	0.00%

Table 6. NMSE values averaged over all instances that do not violate constraints on test data.

Comparison	$\epsilon = 0.05$		$\epsilon = 0.1$	
	VeriSEA	Baseline	VeriSEA	Baseline
vs PySR	0.0593	0.3113	0.0879	0.1217
vs LLM-SR	0.0200	0.1580	0.0205	0.1583

Table 4 compares VeriSEA against the LLM baseline and CRANE [4], a state-of-the-art constrained-decoding method, on GSM-Symbolic using Qwen3-8B. The LLM baseline (Qwen3-8B) achieves only 38.3% accuracy with a 10.6% constraint violation rate (**RQ1**), confirming that a vanilla LLM frequently produces outputs that are syntactically or semantically invalid. CRANE improves accuracy to 44.7% and reduces violations to 2.1% by augmenting the output grammar with reasoning tokens. [1]VeriSEA without parameter tuning already achieves 53.2% accuracy and eliminates all violations, an 8.5% accuracy improvement over CRANE with strictly stronger correctness guarantees. This improvement stems from the FGGM mechanism: the rejection sampler with a verified fallback ensures that every output satisfies the formal grammar and the semantic specification, while the synthesized prompting program f_p guides the underlying LLM toward correct answers.
We also compare the performance of VeriSEA with and without parameter tuning (**RQ2**). We fine-tune Qwen3-8B via GRPO with LoRA adapters under the same formal constraints. This provides a further 12.8% accuracy gain (from 53.2% to 66.0%) while maintaining the zero-violation guarantee. The tuned model produces outputs that more often pass the FGGM checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ on the first sample, reducing reliance on the fallback and thereby improving answer quality. Notably, the tuned variant is also faster than the untuned one (16.7 s vs. 18.8 s), because higher conformance reduces the number of rejection-sampling iterations needed per call.

6.2.4 Constrained Symbolic Regression. In constrained symbolic regression, we consider the SOTA genetic programming–based method PySR [8], as well as the existing self-evolving method LLM-SR [37], which samples parametric programs with optimizable PyTorch parameters and refines them to minimize loss. We denote the noise level by ϵ. As shown in Table 5, both baselines frequently produce programs that violate the behavioral specifications on the test data, with violations occurring in up to 62.86% of synthesis instances for PySR and up to 34.29% for LLM-SR. For both noise levels, VeriSEA finds a verified solution in 33 out of 35 instances. Table 6 presents a pairwise comparison between VeriSEA and the baselines in terms of normalized mean squared error (NMSE) on the test dataset. We compute the average NMSE over only those instances where the baseline-generated programs satisfy the behavioral constraints, pruning instances that correspond to invalid solutions. VeriSEA achieves significantly lower NMSE values compared to both baselines.

6.3 Constraint Decomposition

The previous section showed that formal constraints improve both safety and performance. We now ask a finer-grained question:

> **RQ3 (Constraint Decomposition).** *What are the individual contributions of parameter tuning with local FGGM contracts versus global loss* L*?*

[1]CRANE only ensures that at every decoding step, the generated partial output is a valid prefix of the target grammar.

Table 7. Constraint decomposition ablation on GSM-Symbolic (Qwen3-8B). *Local* = FGGM contracts only; *Global* = behavioral specification only; *Full* = both.

Configuration	Accuracy (%) ↑	Violation (%) ↓
LLM (Qwen3-8B)	38.3	10.6
VeriSEA (no parameter tuning)	53.2 (+14.9)	0.0 (-10.6)
VeriSEA (local only parameter tuning)	55.3 (+17.0)	0.0 (-10.6)
VeriSEA (global only parameter tuning)	61.7 (+23.4)	0.0 (-10.6)
VeriSEA (full)	66.0 (+27.7)	0.0 (-10.6)

We ablate each component independently on GSM-Symbolic. Table 7 decomposes the 12.8% gain that parameter tuning provides (from 53.2% to 66.0%, as shown in Table 4) into its two sources. Starting from VeriSEA without parameter tuning (53.2%), adding *local-only* tuning, which optimizes the FGGM conformance loss so that the model's outputs more frequently satisfy per-call contracts, yields a modest improvement to 55.3%. This 2.1% gain indicates that parameter tuning alone helps the model produce syntactically valid outputs more reliably, reducing reliance on rejection sampling. *Global-only* tuning, which optimizes the task loss L without the FGGM conformance component, provides a substantially larger gain to 61.7%. This 8.5% improvement reflects the direct benefit of training the model to produce semantically correct answers. The full VeriSEA pipeline combines both losses and achieves 66.0%, a 4.3% gain. This demonstrates that the two training signals are complementary: global tuning improves answer correctness while local conformance tuning ensures outputs satisfy per-call contracts, jointly yielding the best overall accuracy.

6.4 Example Synthesized Agent

We provide a complete agent program synthesised by VeriSEA for the DafnyBench task in Appendix K. The agent defines three FGGMs: $initialFGGM$, $diffErrorFGGM$, and $verifierErrorFGGM$. Each sharing the local output contract $\Psi_l := diffChecker(base\_program, \cdot)$ and differing only in their prompting function f_p, which specialises the LLM prompt for initial annotation, diff-checker repair, and verifier-error repair, respectively. The resulting agent program is verified to provably comply with the diff-checker specification for all inputs and all parameter values, ensuring that no synthesised output modifies the original program beyond adding annotations.

6.5 Discussion

Across all four tasks, VeriSEA achieves zero constraint violations on held-out test inputs while simultaneously improving task performance over every baseline. These results confirm that formal behavioral specifications do not merely enforce safety but actively prune the space of candidate programs, steering synthesis toward higher-quality agents. VeriSEA's runtime overhead is competitive: on τ^2-bench it is *faster* than Agent-C, and on Dafny and GSM-Symbolic the full pipeline introduces a modest 1.9–2.5× slowdown relative to the LLM baseline, attributable to the verification and rejection-sampling steps. One current limitation is that the formulation is *resource-unaware*: behavioral specifications constrain functional correctness but do not account for computational resources such as LLM calls, token usage, or wall-clock budget. Extending VeriSEA with resource-bound specifications [23], for example by encoding token or call budgets as additional hard constraints within the FGGM framework, is a promising direction for future work.

7 RELATED WORK

Self-Evolving Agents and Their Risks. Recent work has explored automatic agent synthesis and self-improvement, from code-based action unification [49] and open-ended skill libraries [47] to architecture search [13, 19, 53], gradient-free symbolic updates [51], co-evolution with world models [11], and reinforcement-learning-driven skill refinement [43, 48]. However, none of these provide formal behavioral guarantees, a gap made urgent by demonstrated risks of coding-agent exploits [14, 15, 25], test-case exploitation [50], and verification cheating [3]. VeriSEA addresses this by imposing and verifying hard formal specifications before deployment.

Runtime Monitoring and Shielding. Rather than verifying at synthesis time, runtime approaches enforce safety on observed executions. Shield synthesis from neural policies [52] and temporal-logic runtime checking of tool-call sequences [20] are representative examples, but both are limited to observed traces and cannot guarantee safety on unseen inputs. VeriSEA instead verifies the synthesized program for *all* inputs and parameter values at synthesis time. The monitoring checks can be used to define the specification as done with [20].

Neuro-Symbolic Program Synthesis. Neuro-symbolic methods, including neural program generation from examples [34], type-directed differentiable programming [46], wake-sleep library learning [10], vision-language program composition [41], and LLM-guided symbolic regression [37], synthesize programs that invoke neural components but lack formal correctness guarantees over the composed system. VeriSEA extends this paradigm with FGGMs that bind each model call to a verified local contract, enabling end-to-end verification.

Constrained Decoding. Constrained decoding restricts LLM outputs to a formal language through grammar-based [35, 44, 45], diffusion-based [40], programming-abstraction [6, 21], reasoning-preserving [4], type-based [32], and semantic [33] techniques. These approaches require access to the model's decoding internals (precluding closed-source models) and enforce per-token or per-output constraints, not program-level behavioral specifications. VeriSEA's FGGM mechanism operates on model outputs via rejection sampling, supports rich first-order logic specifications verified at the program level, and is model-agnostic.

Deductive Program Synthesis. Classical deductive synthesis, including syntax-guided [1], semantics-guided [22], counterexample-guided [38], abstraction-guided [16], and component-based [12] approaches, provides strong correctness guarantees but targets deterministic, non-parametric programs. VeriSEA bridges this gap with a CEGIS-style search–verify loop for program structure while using FGGMs to encapsulate parametric generative models within verified contracts.

8 CONCLUSION

We presented VeriSEA, a framework for synthesizing self-evolving LLM agents with formal behavioral guarantees. By introducing Formally Guarded Generative Models (FGGMs), VeriSEA binds each generative model call to a verified local contract backed by a rejection sampler with a provably correct fallback, ensuring that specifications hold for all inputs and all parameter values. This design decomposes the constrained learning problem into a CEGIS-style discrete search over program structure followed by unconstrained gradient-based parameter optimization, retaining the scalability of modern fine-tuning methods such as GRPO while providing end-to-end correctness guarantees. Our evaluation across constrained symbolic regression, LLM-assisted Dafny verification, symbolic math synthesis, and policy-compliant agentic tool use demonstrates that VeriSEA achieves zero constraint violations on all tasks while simultaneously improving task performance over unconstrained and state-of-the-art baselines. These results show that formal behavioral constraints are not merely a safety mechanism but an active guide that prunes the search space and steers synthesis toward higher-quality agents.

DATA-AVAILABILITY STATEMENT

Our implementation and experimental artifacts will be made publicly available and submitted for Artifact Evaluation. We will release code, datasets, and scripts to reproduce results upon acceptance.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Anthropic. 2025. System Card: Claude Sonnet 4.5. https://www-cdn.anthropic.com/963373e433e489a87a10c823c52a0a013e9172dd.pdf. Accessed: 2026-03-18.

[3] Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. 2026. DafnyPro: LLM-Assisted Automated Verification for Dafny Programs. arXiv:2601.05385 [cs.SE] https://arxiv.org/abs/2601.05385

[4] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. CRANE: Reasoning with constrained LLM generation. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=wKs9fHYxCV

[5] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. arXiv:2506.07982 [cs.AI] https://arxiv.org/abs/2506.07982

[6] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (June 2023), 24 pages. https://doi.org/10.1145/3591300

[7] Iwo Błądek and Krzysztof Krawiec. 2019. Solving symbolic regression problems with formal constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 977–984. https://doi.org/10.1145/3321707.3321743

[8] Miles Cranmer. 2023. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl. arXiv:2305.01582 [astro-ph.IM] https://arxiv.org/abs/2305.01582

[9] Dafny Language Community. 2023. Integrating Dafny and Python Code. https://dafny.org/v3.10.0/DafnyRef/integration-py/IntegrationPython. Accessed: 2026-03-17.

[10] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[11] Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. 2025. WebEvolver: Enhancing Web Agent Self-Improvement with Coevolving World Model. *arXiv preprint arXiv:2504.21024* (2025).

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[13] Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. 2025. FlowReasoner: Reinforcing Query-Level Meta-Agents. arXiv:2504.15257 [cs.AI] https://arxiv.org/abs/2504.15257

[14] The Guardian. 2026. Rogue AI agents published passwords and bypassed security protections. News investigation.

[15] Chengquan Guo, Xun Liu, Chulin Xie, Andy Zhou, Yi Zeng, Zinan Lin, Dawn Song, and Bo Li. 2024. RedCode: Risky Code Execution and Generation Benchmark for Code Agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=mAG68wdggA

[16] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 171 (June 2023), 24 pages. https://doi.org/10.1145/3591285

[17] Christian Haider and Gabriel Kronberger. 2022. Shape-Constrained Symbolic Regression with NSGA-III. In *Computer Aided Systems Theory – EUROCAST 2022: 18th International Conference, Las Palmas de Gran Canaria, Spain, February 20–25, 2022, Revised Selected Papers* (Las Palmas de Gran Canaria, Spain). Springer-Verlag, Berlin, Heidelberg, 164–172. https://doi.org/10.1007/978-3-031-25312-6_19

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *Iclr* 1, 2 (2022), 3.

[19] Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=t9U3LW7JVX

[20] Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. 2025. Enforcing Temporal Constraints for LLM Agents. arXiv:2512.23738 [cs.PL] https://arxiv.org/abs/2512.23738

[21] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=sY5N0zY5Od

[22] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 30 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434311

[23] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. https://doi.org/10.1145/3314221.3314602

[24] G. Kronberger, F. O. de Franca, B. Burlacu, C. Haider, and M. Kommenda. 2022. Shape-Constrained Symbolic Regression—Improving Extrapolation with Prior Knowledge. *Evolutionary Computation* 30, 1 (03 2022), 75–98. https://doi.org/10.1162/evco_a_00294 arXiv:https://direct.mit.edu/evco/article-pdf/30/1/75/1995582/evco_a_00294.pdf

[25] Eunkyu Lee, Donghyeon Kim, Wonyoung Kim, and Insu Yun. 2025. Takedown: How It's Done in Modern Coding Agent Exploits. arXiv:2509.24240 [cs.CR] https://arxiv.org/abs/2509.24240

[26] K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[27] Li Li, Minjie Fan, Rishabh Singh, and Patrick Riley. 2019. Neural-Guided Symbolic Regression with Asymptotic Constraints. arXiv:1901.07714 [cs.LG] https://arxiv.org/abs/1901.07714

[28] Wenqiang Li, Weijun Li, Linjun Sun, Min Wu, Lina Yu, Jingyi Liu, Yanjie Li, and Songsong Tian. 2023. Transformer-based model for symbolic regression via joint supervised learning. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ULzyv9M1j5

[29] Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2025. DafnyBench: A Benchmark for Formal Software Verification. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=yBgTVWccIx

[30] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229* (2024).

[31] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 134 (April 2025), 27 pages. https://doi.org/10.1145/3720499

[32] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 171 (June 2025), 26 pages. https://doi.org/10.1145/3729274

[33] Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. 2026. ChopChop: A Programmable Framework for Semantically Constraining the Output of Language Models. *Proc. ACM Program. Lang.* 10, POPL, Article 66 (Jan. 2026), 28 pages. https://doi.org/10.1145/3776708

[34] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. arXiv:1611.01855 [cs.AI] https://arxiv.org/abs/1611.01855

[35] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=5G7ve8E1Lu

[36] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

[37] Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. 2025. LLM-SR: Scientific Equation Discovery via Programming with Large Language Models. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=m2nmp8P5in

[38] Armando Solar-Lezama. 2013. Program Sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[39] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 134–155. https://doi.org/10.1007/978-3-031-65112-0_7

[40] Tarun Suresh, Debangshu Banerjee, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. 2025. DINGO: Constrained Inference for Diffusion LLMs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=KaYMGsnZ4R

[41] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 11854–11864. https://doi.org/10.1109/ICCV51070.2023.01092

[42] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[43] Wanxin Tian, Shijie Zhang, Kevin Zhang, Xiaowei Chi, Chunkai Fan, Junyu Lu, Yulin Luo, Qiang Zhou, Yiming Zhao, Ning Liu, Siyu Lin, Zhiyuan Qin, Xiaozhu Ju, Shanghang Zhang, and Jian Tang. 2025. SEEA-R1: Tree-Structured Reinforcement Fine-Tuning for Self-Evolving Embodied Agents. arXiv:2506.21669 [cs.AI] https://arxiv.org/abs/2506.21669

[44] Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. 2025. IterGen: Iterative Semantic-aware Structured LLM Generation with Backtracking. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=ac93gRzxxV

[45] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. SynCode: LLM Generation with Grammar Augmentation. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=HiUZtgAPoH

[46] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. HOUDINI: lifelong learning as program synthesis. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 8701–8712.

[47] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291 [cs.AI] https://arxiv.org/abs/2305.16291

[48] Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. 2026. Reinforcement Learning for Self-Improving Agent with Skill Library. arXiv:2512.17102 [cs.AI] https://arxiv.org/abs/2512.17102

[49] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 2054, 25 pages.

[50] Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. 2025. ImpossibleBench: Measuring LLMs' Propensity of Exploiting Test Cases. *arXiv preprint arXiv:2510.20270* (2025).

[51] Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. 2024. Symbolic Learning Enables Self-Evolving Agents. arXiv:2406.18532 [cs.CL] https://arxiv.org/abs/2406.18532

[52] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 686–701. https://doi.org/10.1145/3314221.3314638

[53] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. GPTSwarm: Language Agents as Optimizable Graphs. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=uTC9AFXIhg

A RESTRICTED DAFNY GRAMMAR

$$
\begin{array}{rcl}
\langle program \rangle & ::= & \langle method\_decl \rangle \mid \langle function\_decl \rangle \\
\langle method\_decl \rangle & ::= & \texttt{method}\ \langle ident \rangle (\langle params \rangle)\ \langle specs \rangle\ \{\langle stmt\_list \rangle\} \\
\langle function\_decl \rangle & ::= & \texttt{function}\ \langle ident \rangle (\langle params \rangle) : \langle type \rangle\ \langle specs \rangle\ \langle expr \rangle \\
\langle params \rangle & ::= & \epsilon \mid \langle param \rangle \mid \langle param \rangle, \langle params \rangle \\
\langle param \rangle & ::= & \langle ident \rangle : \langle type \rangle \\
\langle type \rangle & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{real} \\
\langle specs \rangle & ::= & \epsilon \mid \langle spec \rangle \langle specs \rangle \\
\langle spec \rangle & ::= & \texttt{requires}\ \langle formula \rangle \mid \texttt{ensures}\ \langle formula \rangle \\
\langle stmt\_list \rangle & ::= & \epsilon \mid \langle stmt \rangle \langle stmt\_list \rangle \\
\langle stmt \rangle & ::= & \langle assign\_stmt \rangle \mid \langle if\_stmt \rangle \mid \langle while\_stmt \rangle \mid \langle assert\_stmt \rangle \mid \langle call\_stmt \rangle \\
\langle assign\_stmt \rangle & ::= & \langle ident \rangle := \langle expr \rangle; \\
\langle call\_stmt \rangle & ::= & \langle ident \rangle (\langle args \rangle); \\
\langle args \rangle & ::= & \epsilon \mid \langle expr \rangle \mid \langle expr \rangle, \langle args \rangle \\
\langle if\_stmt \rangle & ::= & \texttt{if}\ (\langle formula \rangle)\{\langle stmt\_list \rangle\} \\
 & & \mid \texttt{if}\ (\langle formula \rangle)\{\langle stmt\_list \rangle\}\ \texttt{else}\ \{\langle stmt\_list \rangle\} \\
\langle while\_stmt \rangle & ::= & \texttt{while}\ (\langle formula \rangle)\ \langle loop\_annots \rangle\ \{\langle stmt\_list \rangle\} \\
\langle loop\_annots \rangle & ::= & \epsilon \mid \langle loop\_annot \rangle \langle loop\_annots \rangle \\
\langle loop\_annot \rangle & ::= & \texttt{invariant}\ \langle formula \rangle \mid \texttt{decreases}\ \langle expr \rangle \\
\langle assert\_stmt \rangle & ::= & \texttt{assert}\ \langle formula \rangle; \\
\langle expr \rangle & ::= & \langle aexpr \rangle \mid \langle sexpr \rangle \\
\langle aexpr \rangle & ::= & \langle int\_lit \rangle \mid \langle ident \rangle \mid \langle aexpr \rangle\ \langle arith\_op \rangle\ \langle aexpr \rangle \mid (\langle aexpr \rangle) \mid \langle ident \rangle (\langle args \rangle) \\
\langle sexpr \rangle & ::= & \langle string\_lit \rangle \mid \langle ident \rangle \mid (\langle sexpr \rangle) \mid \langle ident \rangle (\langle args \rangle) \\
\langle formula \rangle & ::= & \langle expr \rangle\ \langle rel\_op \rangle\ \langle expr \rangle \\
 & \mid & !\ \langle formula \rangle \\
 & \mid & (\langle formula \rangle) \\
 & \mid & \langle ident \rangle (\langle args \rangle) \\
 & \mid & \texttt{forall}\ \langle bound\_vars \rangle :: \langle formula \rangle \\
 & \mid & \texttt{exists}\ \langle bound\_vars \rangle :: \langle formula \rangle \\
\langle bound\_vars \rangle & ::= & \langle ident \rangle : \langle type \rangle \mid \langle ident \rangle : \langle type \rangle, \langle bound\_vars \rangle \\
\langle arith\_op \rangle & ::= & + \mid - \mid * \mid / \\
\langle rel\_op \rangle & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq \\
\langle literal \rangle & ::= & \langle int\_lit \rangle \mid \langle string\_lit \rangle \mid \texttt{true} \mid \texttt{false} \\
\langle int\_lit \rangle & ::= & \texttt{0} \mid 1 \mid \cdots \mid 9 \text{ (and sequences thereof)} \\
\langle string\_lit \rangle & ::= & \text{"}\langle char \rangle^*\text{"} \\
\langle ident \rangle & ::= & \langle letter \rangle \mid \langle letter \rangle \langle ident \rangle \\
\langle letter \rangle & ::= & a \mid b \mid \cdots \mid z \mid A \mid B \mid \cdots \mid Z \\
\langle digit \rangle & ::= & 0 \mid 1 \mid \cdots \mid 9
\end{array}
$$

B REJECTION SAMPLING DETAILS

Rejection Sampling Details. Given a target distribution π_t and a proposal distribution π_p with $S(\pi_t) \subseteq S(\pi_p)$, assume there exists $M \geq 1$ such that $\forall z \in \Omega, \quad D_{\pi_t}(z) \leq M \cdot D_{\pi_p}(z)$. The procedure samples $z \sim \pi_p$ and $u \sim \text{Uniform}(0,1)$, and accepts z iff $u \leq \frac{D_{\pi_t}(z)}{M \cdot D_{\pi_p}(z)}$. Otherwise, the sample is rejected and the process repeats. Accepted samples follow π_t. Moreover, any $z \notin S(\pi_t)$ is always rejected since $D_{\pi_t}(z) = 0$, ensuring all accepted samples lie in $S(\pi_t)$. The acceptance probability is $1/M$, so efficiency depends on how tightly π_p upper-bounds π_t. For this paper, we assume $M = 1$.

C LIBRARY FUNCTIONS FOR SYMBOLIC REGRESSION

Library Function Set $\mathcal{F}$.

- $abs(x : real) \longrightarrow r : real$ **ensures** $r \geq 0$ **ensures** $r = x \vee r = -x$
- $max(x : real, y : real) \longrightarrow r : real$ **ensures** $(x \leq r) \wedge (y \leq r)$ **ensures** $(r = x) \vee (r = y)$
- $min(x : real, y : real) \longrightarrow r : real$ **ensures** $(x \geq r) \wedge (y \geq r)$ **ensures** $(r = x) \vee (r = y)$
- $sin(x : real) \longrightarrow r : real$ **ensures** $-1 \leq r \leq 1$ **ensures** $(x = 0) \Rightarrow (r = 0)$
- $cos(x : real) \longrightarrow r : real$ **ensures** $-1 \leq r \leq 1$ **ensures** $(x = 0) \Rightarrow (r = 1)$
- $pi(x : real) \longrightarrow r : real$ **requires** $x \geq 0$ **ensures** $3.141592653589790 \cdot x \leq r$ **ensures** $r \leq 3.141592653589793 \cdot x$
- $pow(x : real, d : real) \longrightarrow r : real$ **requires** $x \geq 0$ **requires** $(x \neq 0) \vee (d \geq 0)$ **ensures** $r \geq 0$ **ensures** $(x > 0) \Rightarrow (r > 0)$ **ensures** $(d = 0) \Rightarrow (r = 1)$ **ensures** $(x \geq 1 \wedge d \geq 0) \Rightarrow (r \geq 1)$ **ensures** $(x \geq 1 \wedge d \leq 0) \Rightarrow (r \leq 1)$ **ensures** $(x \leq 1 \wedge d \geq 0) \Rightarrow (r \leq 1)$ **ensures** $(x \leq 1 \wedge d \leq 0) \Rightarrow (r \geq 1)$
- $sqrt(x : real) \longrightarrow r : real$ **requires** $x \geq 0$ **ensures** $r \geq 0$ **ensures** $r \cdot r = x$
- $exp(x : real) \longrightarrow r : real$ **ensures** $r \geq 0$ **ensures** $(x \leq 0) \Rightarrow (r \leq 1)$ **ensures** $(x \geq 0) \Rightarrow (r \geq 1)$ **ensures** $(x = 1) \Rightarrow (2.71 \leq r \leq 2.72)$ **ensures** $r \geq (1 + x)$
- $log(x : real) \longrightarrow r : real$ **requires** $x > 0$ **ensures** $(x \geq 1) \Rightarrow (r \geq 0)$ **ensures** $(x \leq 1) \Rightarrow (r \leq 0)$ **ensures** $(x = 1) \Rightarrow (r = 0)$
- $neural1(x : real) \longrightarrow r : real$
- $neural2(x1 : real, x2 : real) \longrightarrow r : real$
- $neural3(x1 : real, x2 : real, x3 : real) \longrightarrow r : real$

D ADDITIONAL AXIOMS FOR SYMBOLIC REGRESSION

D.1 Axiom Syntax

$$\langle \text{axiom} \rangle ::= \langle formula \rangle$$

Axiom Set $\mathcal{A}$.

- **Interval Multiplication Rule:** $\forall\, x, y, x_{lb}, x_{ub}, y_{lb}, y_{ub} \in \mathbb{R}.\ (x_{lb} \leq x \leq x_{ub}) \wedge (y_{lb} \leq y \leq y_{ub}) \implies$

$min(x_{lb} \times y_{lb},\ x_{lb} \times y_{ub},\ x_{ub} \times y_{lb},\ x_{ub} \times y_{ub}) \leq x \times y \leq max(x_{lb} \times y_{lb},\ x_{lb} \times y_{ub},\ x_{ub} \times y_{lb},\ x_{ub} \times y_{ub})$.

- $\forall\, x, d_1, d_2 \in \mathbb{R}.\ (0 \leq x \leq 1) \wedge (d_1 \leq d_2) \implies pow(x, d_1) \geq pow(x, d_2)$.
- $\forall\, x, d_1, d_2 \in \mathbb{R}.\ (x \geq 1) \wedge (d_1 \leq d_2) \implies pow(x, d_1) \leq pow(x, d_2)$.
- $\forall\, x \in \mathbb{R}.\ exp(x) = pow(exp(1),\ x)$.
- $\forall\, x \in \mathbb{R}.\ (x > 0) \implies \big(\forall\, r \in \mathbb{R}.\ r = log(x) \iff exp(r) = x\big)$.
- $\forall\, x \in \mathbb{R}.\ (x \geq 0) \implies (sqrt(x) = pow(x, 0.5))$.

E LIBRARY FUNCTIONS FOR DAFNY PROGRAM VERIFICATION

Library Function Set $\mathcal{F}$.

- $claude(prompt : str) \to r : str$
- $diffChecker(base\_code : str, annotated\_code : str) \to r : bool$
- $diffCheckerWithErrorMsg(base\_code : str, annotated\_code : str) \to r : str$
- $dafnyVerifier(dafny\_code : str) \to r : bool$
- $dafnyVerifierWithErrorMsg(dafny\_code : str) \to r : str$
- $extractDafnyCode(text : str) \to r : str$
- $isSubstring(sub : str, full : str) \to r : bool$

F ADDITIONAL AXIOMS FOR DAFNY PROGRAM VERIFICATION

Axiom Set $\mathcal{A}$.

- **Reflexivity:** $\forall\, a, b \in \Sigma^*.\ (a = b) \implies diffChecker(a, b)$.
- **Transitivity:** $\forall\, a, b, c \in \Sigma^*.\ diffChecker(a, b) \wedge diffChecker(b, c) \implies diffChecker(a, c)$.
- $\forall\, a, b \in \Sigma^*.\ diffCheckerWithErrorMsg(a, b) = \text{“”} \iff diffChecker(a, b)$.
- $\forall\, c \in \Sigma^*.\ dafnyVerifierWithErrorMsg(c) = \text{“”} \iff dafnyVerifier(c)$.

G LIBRARY FUNCTIONS FOR τ^2-BENCH

Library Function Set $\mathcal{F}$.

- $qwen(prompt : str) \to r : str$
- $agentCCheck(tool : str, trace : str, domain : str) \to r : bool$
- $isToolCall(response : str) \to r : bool$
- $parseToolCallName(response : str) \to r : str$

H ADDITIONAL AXIOMS FOR τ^2-BENCH

Axiom Set $\mathcal{A}$.

- **Non-tool-call safety:** $\forall\, r, t, d \in \Sigma^*.\ \neg isToolCall(r) \implies agentCCheck(r, t, d)$.
- **Transfer-to-human satisfiability:** $\forall\, r, t, d \in \Sigma^*.\ isToolCall(r) \wedge parseToolCallName(r) =$ “transfer_to_human_agents” $\implies agentCCheck(r, t, d)$.

I LIBRARY FUNCTIONS FOR GSM-SYMBOLIC

Library Function Set $\mathcal{F}$.

- $qwen(prompt : str) \to r : str$
- $gsmParser(expression : str) \to r : bool$
- $gsmParserWithErrorMsg(expression : str) \to r : str$
- $extractExpression(text : str) \to r : str$

J ADDITIONAL AXIOMS FOR GSM-SYMBOLIC

Axiom Set $\mathcal{A}$.

- $gsmParser(\text{“« »”})$.
- $\forall\, e \in \Sigma^*.\ gsmParserWithErrorMsg(e) = \text{“”} \iff gsmParser(e)$.

K EXAMPLE AGENTS FOR LLM ASSISTED PROGRAM VERIFICATION

The following is an example verified agent program for the Dafny program verification task. The agent defines and invokes three FGGMs—$initialFGGM$ (Fig. 4), $diffErrorFGGM$ (Fig. 5), and $verifierErrorFGGM$ (Fig. 6)—each with local contract $\Psi_l : diffChecker(base\_program, \cdot)$ and a fallback that returns the original program (justified by the reflexivity axiom). All three share $GMid := claude$ and differ only in the prompting function f_p: the first generates an initial annotation with context-specific guidance, the second repairs after a diff-checker failure, and the third repairs

after a verification failure with error-specific feedback. The agent program (Fig. 7) dispatches to the appropriate FGGM on each loop iteration.

FGGM Definition:

$id := initialFGGM$
$GMid := claude$
$typeSig := (base\_program : str) \to str$
$\Phi_I := \textbf{true}$
$\Psi_I := diffChecker(base\_program,\ f(base\_program))$
$f_p := \textbf{function}\ f_p(base\_program : str) : (str)\ \{$
  var $g : str :=$
    "You are an expert in Dafny verification. Given the following unannotated Dafny code, "
    + "add appropriate loop invariants, assertions, and decreases clauses to make it verify.\n\n"
    + "IMPORTANT RULES:\n"
    + "1. Do NOT modify the method signatures, requires clauses, or ensures clauses\n"
    + "2. Only add invariants, assertions, and decreases clauses\n"
    + "3. Do NOT change the logic or control flow\n"
    + "4. Keep invariants MINIMAL - only what's necessary for postconditions\n"
    + "5. Add decreases clauses for termination\n";

  if $(isSubstring(\text{"lemma "}, base\_program))$ {
    $g := g$
      + "6. LEMMAS: If a lemma has a non-trivial postcondition, add explicit proof steps:\n"
      + " - Use assertions to unfold function definitions\n"
      + " - Add case analysis or pattern matching if needed\n"
      + " - Call other lemmas to establish intermediate facts\n"
      + " - Do NOT leave lemma bodies empty unless postcondition is trivial\n"; }

  if $(isSubstring(\text{"<"}, base\_program) \wedge isSubstring(\text{">"}, base\_program))$ {
    $g := g$
      + "7. GENERICS: When calling generic functions/lemmas, propagate type constraints:\n"
      + " - If a called function requires T(00), declare it in the caller too\n"
      + " - Explicitly instantiate type parameters when needed\n"; }

  if $(isSubstring(\text{"set<"}, base\_program) \vee isSubstring(\text{"set "}, base\_program))$ {
    $g := g$
      + "8. SETS: Use simple, direct assertions for set reasoning:\n"
      + " - Assert set equality decompositions: A == B + (A - B)\n"
      + " - Use cardinality constraints: |A| == |B| + |C|\n"
      + " - Avoid complex witness variables or case analysis\n"
      + " - Let Dafny's built-in set theory automation work\n"; }

  if $(isSubstring(\text{"multiset"}, base\_program))$ {
    $g := g$
      + "9. MULTISETS: Match the structure of postconditions directly\n"
      + " - Use simple decomposition assertions\n"
      + " - Avoid unnecessary intermediate steps\n"; }

  return g + "\nBase program:\n" + $base\_program$
    + "\n\nReturn ONLY the annotated Dafny code in a ```dafny code block."; }

$f_d := \textbf{function}\ f_d(base\_program : str,\ y : str) : (str)$
  ensures $diffChecker(base\_program,\ f_d(base\_program, y))$
{
  return $base\_program$;
}

Fig. 4. FGGM definition for *initialFGGM*. The prompting function builds context-specific guidance by checking for lemmas, generics, sets, and multisets in the base program. The fallback returns the original program, satisfying Ψ_I by the reflexivity axiom.

FGGM Definition:

$id := diffErrorFGGM$
$GMid := claude$
$typeSig := (base\_program : str,\ diff\_error : str,\ prev\_attempt : str) \rightarrow str$
$\Phi_l := \textbf{true}$
$\Psi_l := diffChecker(base\_program,\ f(base\_program,\ diff\_error,\ prev\_attempt))$
$f_p :=$ **function** $f_p(base\_program : str,\ diff\_error : str,\ prev\_attempt : str) : (str)$ {
return
"The previous Dafny annotation modified the base program logic incorrectly.\n\n"
+ "DiffChecker error:\n" + $diff\_error$ + "\n\n"
+ "Base program:\n" + $base\_program$ + "\n\n"
+ "Previous attempt:\n" + $prev\_attempt$ + "\n\n"
+ "Please add annotations WITHOUT changing:\n"
+ "- Method signatures\n"
+ "- Requires/ensures clauses\n"
+ "- Program logic or control flow\n"
+ "Only add invariants, assertions, and decreases clauses.\n"
+ "Return ONLY the corrected Dafny code in a ```dafny code block."; }

$f_d :=$ **function** $f_d(base\_program : str,\ diff\_error : str,\ prev\_attempt : str,\ y : str) : (str)$
ensures $diffChecker(base\_program,\ f_d(base\_program,\ diff\_error,\ prev\_attempt,\ y))$
{
return $base\_program$;
}

Fig. 5. FGGM definition for $diffErrorFGGM$. Invoked when the previous iteration's output failed the diff checker. The prompt includes the diff error and previous attempt, instructing the model to preserve the base program logic.

FGGM Definition:

$id := verifierErrorFGGM$
$GMid := claude$
$typeSig := (base\_program : str,\ verify\_error : str,\ prev\_attempt : str) \to str$
$\Phi_l := \textbf{true}$
$\Psi_l := diffChecker(base\_program,\ f(base\_program,\ verify\_error,\ prev\_attempt))$
$f_p :=$ **function** $f_p(base\_program : str,\ verify\_error : str,\ prev\_attempt : str) : (str)$ {
  var $sg : str :=$
    "Focus on:\n"
    + "- Strengthening invariants MINIMALLY\n"
    + "- Adding simple decomposition assertions\n"
    + "- Ensuring decreases clauses are correct\n";
  if $(isSubstring(\text{"timeout"},\ verify\_error) \lor isSubstring(\text{"resource"},\ verify\_error))$ {
    $sg :=$ "TIMEOUT/RESOURCE ISSUE: Break down the proof into smaller steps:\n"
      + "- Add strategic intermediate assertions\n"
      + "- Call helper lemmas to establish sub-goals\n"
      + "- Simplify complex invariants\n"; }
  if $(isSubstring(\text{"lemma"},\ verify\_error) \lor isSubstring(\text{"postcondition"},\ verify\_error))$ {
    $sg :=$ "LEMMA PROOF ISSUE: Add explicit proof steps in lemma body:\n"
      + "- Assert intermediate facts that lead to postcondition\n"
      + "- Unfold recursive function definitions\n"
      + "- Use case analysis if needed\n"; }
  if $(isSubstring(\text{"invariant"},\ verify\_error))$ {
    $sg :=$ "INVARIANT ISSUE: Adjust loop invariants:\n"
      + "- Ensure invariants are maintained after each iteration\n"
      + "- Add bounds and relationships incrementally\n"
      + "- Keep invariants minimal but sufficient\n"; }
  if $(isSubstring(\text{"type"},\ verify\_error) \lor isSubstring(\text{"constraint"},\ verify\_error))$ {
    $sg :=$ "TYPE CONSTRAINT ISSUE: Check generic type parameters:\n"
      + "- Propagate nonemptiness constraints like T(00)\n"
      + "- Ensure type parameters match between caller and callee\n"; }
  return
    "The previous Dafny annotation preserved the base logic but failed verification.\n\n"
    + "Verification error:\n" + $verify\_error$ + "\n\n"
    + sg + "\n"
    + "Base program:\n" + $base\_program$ + "\n\n"
    + "Previous attempt:\n" + $prev\_attempt$ + "\n\n"
    + "Please fix the annotations. Do NOT modify method signatures, requires, or ensures clauses.\n"
    + "Return ONLY the corrected Dafny code in a ```dafny code block."; }

$f_d :=$ **function** $f_d(base\_program : str,\ verify\_error : str,\ prev\_attempt : str,\ y : str) : (str)$
  ensures $diffChecker(base\_program,\ f_d(base\_program,\ verify\_error,\ prev\_attempt,\ y))$
{
  return $base\_program$;
}

Fig. 6. FGGM definition for $verifierErrorFGGM$. Invoked when the previous iteration passed the diff checker but failed verification. The prompting function analyzes the verification error to provide targeted guidance (timeout, lemma, invariant, or type issues).

```
method agent(base_program : str) returns (r : str)
  ensures diffChecker(base_program, r)
{
  var max_attempts : int := 5;
  var attempt : int := 0;
  var best : str := base_program;
  var best_verified : bool := false;

  var prev_diff_err : str := "";
  var prev_ver_err : str := "";
  var prev_code : str := "";

  while (attempt < max_attempts)
    invariant diffChecker(base_program, best)
    decreases max_attempts − attempt
  {
    var y : str;

    if (attempt = 0) {
      y := initialFGGM(base_program);
    } else {
      if (prev_diff_err ≠ "") {
        y := diffErrorFGGM(base_program, prev_diff_err, prev_code);
      } else {
        y := verifierErrorFGGM(base_program, prev_ver_err, prev_code);
      }
    }

    prev_diff_err := "";
    prev_ver_err := "";
    prev_code := y;

    var diff_err : str := diffCheckerWithErrorMsg(base_program, y);

    if (diff_err = "") {
      var v_err : str := dafnyVerifierWithErrorMsg(y);

      if (v_err = "") {
        r := y; return;
      } else {
        if (¬best_verified) { best := y; }
        prev_ver_err := v_err;
      }
    } else {
      prev_diff_err := diff_err;
    }
    attempt := attempt + 1;
  }
  r := best; return;
}
```

Fig. 7. Verified agent program for Dafny annotation synthesis. On the first iteration, *initialFGGM* (Fig. 4) generates an initial annotation. On subsequent iterations, the agent dispatches to *diffErrorFGGM* (Fig. 5) or *verifierErrorFGGM* (Fig. 6) depending on the previous error type. Each FGGM guarantees *diffChecker*(*base_program*, *y*), maintaining the loop invariant. Upon exhausting all attempts, the agent returns *best*, which satisfies the postcondition by the loop invariant.

L FGGM SYNTAX

The syntax of a first-order guarded generative model (FGGM) can be described using the following BNF grammar. We use the production rules (⟨ident⟩, ⟨type⟩, ⟨spec⟩, ⟨program⟩, ⟨string_lit⟩, ⟨formula⟩) from Appendix A.

$$
\begin{aligned}
\langle id \rangle &::= \langle \text{ident} \rangle \\
\langle GMid \rangle &::= \langle \text{ident} \rangle \\
\langle typeSig \rangle &::= \text{"("} \langle \text{typedVars} \rangle \text{")""} \rightarrow \text{"} \langle \text{type} \rangle \\
\langle \text{typedVars} \rangle &::= \langle \text{typedVar} \rangle \mid \langle \text{typedVar} \rangle, \langle \text{typedVars} \rangle \\
\langle \text{typedVar} \rangle &::= \langle \text{ident} \rangle : \langle \text{type} \rangle \\
\langle \Phi_l \rangle &::= \texttt{"requires"}\ \langle \text{formula} \rangle \\
\langle \Psi_l \rangle &::= \texttt{"ensures"}\ \langle \text{formula} \rangle \\
\langle f_p \rangle &::= \langle \text{program} \rangle \\
\langle f_d \rangle &::= \langle \text{program} \rangle \\
\langle \text{info} \rangle &::= \langle \text{string\_lit} \rangle
\end{aligned}
$$

M FGGM VALIDITY CHECK

Algorithm 3 checks whether an FGGM definition is valid. It verifies that the type signature and local contracts are syntactically well-formed and that all terms using library functions satisfy their input specifications. It then checks that the prompting program and fallback program are type-correct and terminating. Finally, the fallback program is verified using the deductive verifier to ensure it satisfies the local contract for all inputs. The FGGM definition is accepted only if no errors are found.

Algorithm 3 validateFGGM

Input: FGGM definition G = $(id, GMid, typeSig, \Phi_l, \Psi_l, f_p, f_d)$
Input: Library functions $\mathcal{F}_c$, axioms $\mathcal{A}$
Output: {} if valid else set of errors err
err ← {}
if $\neg WellTyped(typeSig)$ **then**
    err ← err ∪ {Invalid type signature}
end if
if $\neg WellFormedFormula(\Phi_l)$ **then**
    err ← err ∪ {Invalid input contract}
end if
if $\neg WellFormedFormula(\Psi_l)$ **then**
    err ← err ∪ {Invalid output contract}
end if
if $\neg CheckTerms(\Phi_l, \mathcal{F}_c)$ **then**
    err ← err ∪ {Invalid library terms in Φ_l}
end if
if $\neg CheckTerms(\Psi_l, \mathcal{F}_c)$ **then**
    err ← err ∪ {Invalid library terms in Ψ_l}
end if
if $\neg TypeCheck(f_p)$ **or** $\neg Terminates(f_p)$ **then**
    err ← err ∪ {Invalid prompting program}
end if
if $\neg TypeCheck(f_d)$ **or** $\neg Terminates(f_d)$ **then**
    err ← err ∪ {Invalid fallback program}
end if
if $\neg Verify(\forall x_1 \dots x_n, y.\ \Phi_l(x_1 \dots x_n) \Rightarrow \Psi_l(x_1 \dots x_n, f_d(x_1 \dots x_n, y)))$ **then**
    err ← err ∪ {Fallback violates contract}
end if
return err

Algorithm 4 checks whether a concrete input–output tuple satisfies the FGGM output contract. It first substitutes the concrete values into the specification Ψ_l. If the resulting formula is quantifier-free, the checker directly evaluates it by computing all terms using the library functions. If the specification contains quantifiers, the substituted formula is combined with the axioms and input specification and submitted to an SMT solver with a timeout. The procedure returns true only if the solver confirms satisfiability within the time bound.

Algorithm 4 $check_{\mathcal{A},\Phi_l,\Psi_l}$ Contract Checker

Input: Concrete values $(x_1, \ldots, x_n, y)$
Input: Local contracts (Φ_l, Ψ_l), axioms $\mathcal{A}$
Output: T if contract satisfied else F
$\phi \leftarrow Substitute(\Psi_l, \{x_1, \ldots, x_n, y\})$
if $QuantifierFree(\phi)$ **then**
  if $\phi = \neg\phi'$ **then return** $\neg check_{\mathcal{A},\Phi,\phi'}(x_1, \ldots, x_n, y)$
  else if $\phi = \phi' \vee \psi$ **then return** $check_{\mathcal{A},\Phi,\phi'}(x_1, \ldots, x_n, y) \vee check_{\mathcal{A},\Phi,\psi}(x_1, \ldots, x_n, y)$ ▷ Recursively evaluate on subformulas
  else if $\phi = \phi' \wedge \psi$ **then return** $check_{\mathcal{A},\Phi,\phi'}(x_1, \ldots, x_n, y) \wedge check_{\mathcal{A},\Phi,\psi}(x_1, \ldots, x_n, y)$
  else if $isAtomic(\phi)$ **then return** $\phi(x_1, \ldots, x_n, y)$ ▷ Compute the atomic predicate ϕ value on $(x_1, \ldots, x_n, y)$ T or F
  end if
end if
$\psi \leftarrow \varphi_{\mathcal{A}} \wedge \Phi_l(x_1, \ldots, x_n) \wedge \phi$
$res \leftarrow SMTSolve(\psi, time)$
if $res = SAT$ **then**
  return T
else
  return F
end if

N BACKGROUND ON GRPO

Group Relative Policy Optimization (GRPO): GRPO is a reinforcement learning–based fine-tuning method for generative models that optimizes a policy by comparing outputs *relative to other samples from the same input*, rather than relying on an explicit value function. Let $\pi_\theta(y \mid p)$ denote the parametric policy for an FGGM id_{θ_i}, where $p \in \mathbb{P}$ is the input prompt and $y_l \sim \pi_\theta(\cdot \mid p)$ is a sampled output. For each input p, GRPO samples a *group* of G candidate outputs $\{y^{(1)}, \ldots, y^{(G)}\}$ from the current policy. Each sample is assigned a scalar reward $r^{(j)} = \mathcal{R}(p, y^{(j)})$. Instead of using absolute rewards directly, GRPO computes *relative advantages* within the group:

$$A^{(j)} = \frac{r^{(j)} - \mu_p}{\sigma_p}, \quad \mu_p = \frac{1}{G}\sum_{j=1}^{G} r^{(j)}, \quad \sigma_p = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(r^{(j)} - \mu_p)^2 + \epsilon} \tag{24}$$

Here μ_p and σ_p are the mean and standard deviation of rewards within the group.

$$\max_{\theta} \ \mathbb{E}_{p\sim\mathbb{P}}\left[\frac{1}{G}\sum_{j=1}^{G}\min\left(\rho^{(j)}A^{(j)},\ \text{clip}(\rho^{(j)}, 1-\epsilon, 1+\epsilon)A^{(j)}\right)\right] \tag{25}$$

where $\rho^{(j)} = \frac{\pi_\theta(y^{(j)}|p)}{\pi_{\theta_{\text{old}}}(y^{(j)}|p)}$ is the importance ratio. This clipped objective, similar to PPO, ensures stable policy updates. In our setting, each FGGM id_{θ_i} defines a policy $\pi_{\theta_i}(y_l \mid p)$ over outputs y_l for input p. The reward function defined in Eq. 23:

$$\mathcal{R}(p, y_l) = 1 - Sigmoid\big(\mathsf{L}(x, \_, y) \times \mathbb{I}(y = y_l) + \lambda \times (1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l)))\big) \tag{26}$$

O HYPERPARAMETERS

Table 8 lists all hyperparameters used across the four evaluation tasks. The top block reports the shared pipeline settings applied uniformly across tasks; the middle blocks report task-specific training configurations for GSM-Symbolic and Symbolic Regression; the bottom block reports the per-task verifier and solver timeouts.

Table 8. Hyperparameter settings for all evaluation tasks.

Category	Hyperparameter	Value
Shared pipeline	Planner search budget (Δ)	10
	FGGM rejection-sample budget (K)	5
	In-context examples	3 (2 for τ^2-bench)
GRPO / LoRA (GSM-Symbolic)	Base model	Qwen3-8B
	LoRA rank	16
	LoRA alpha (α)	32
	GRPO training epochs	5
	GRPO batch size	64
	GRPO learning rate	1×10^{-5}
	GRPO generations per prompt (G)	8
Parameter tuning (Symbolic Regression)	Optimizer	Adam
	Learning rate	0.05
	Backpropagation steps	40
Verifier timeouts	Dafny — `dafny verify` timeout	240 s per subprocess call
	τ^2-bench — Z3 `solver.check()` timeout	10 s per call (up to 2 retries)
	GSM-Symbolic — Z3 `solver.check()` timeout	5 s per call

P PLANNER FEEDBACK

After each search–verify–learn iteration, VeriSEA constructs structured feedback I' from the execution traces of the current best candidate agent on the training data D (Algorithm 2, line 31). The planner LLM receives the task-performance score of the previous program along with per-example error diagnostics. Specifically, for each training example on which the previous candidate failed, VeriSEA provides the planner with the example input, the agent's output, and the corresponding error message (e.g., a verification error, a grammar violation, or an incorrect answer). The planner is then asked to generate a one-to-two sentence description and a suggested fix for each failed example. The resulting list of per-example descriptions and suggestions is collected and fed back to the planner as part of the context I' when it samples the next candidate program. This feedback mechanism enables the planner to identify recurring failure patterns and adjust its synthesized FGGM definitions and prompting programs f_p accordingly in subsequent iterations.

Q PLANNER LLM PROMPT

The following is the prompt template used by the planner LLM to synthesize candidate agent programs. Placeholders in braces (e.g., `{task_description}`) are populated at each iteration with the task specification, postcondition, available operator library, agent signature, and feedback from the previous iteration's execution traces (Appendix P).

```
You are an expert programmer tasked with writing a Dafny agent function.

## TASK
{task_description}

## CRITICAL POSTCONDITION
{postcondition_description}

## AVAILABLE OPERATORS
You can use these operators in your agent:

```dafny
{library_functions}
```

AGENT SIGNATURE AND POSTCONDITION
```dafny
{agent_signature}
```

CONSTRAINTS
- Fully Typed: All variables must have type annotations
- No imports: All operators are already available in scope
- No recursion: Don't call agent() recursively
- Always satisfy postcondition: Every return path must satisfy the postcondition
- No f-strings: Do not use format strings (f-strings); use the addition operator for concatenation
{task_specific_constraints}

FEW-SHOT EXAMPLES
{few_shot_examples}

PREVIOUS AGENT CODE
{previous_agent_code}

PREVIOUS ATTEMPT FEEDBACK
{feedback}

SCORING INFORMATION
{scoring_info}

OUTPUT
Write ONLY the agent function inside a ```dafny``` block.

Make sure to:
- Check the postcondition before every return
- If the task fails, return a safe fallback value
- LLM calls are stateless, calling the LLM again does not maintain previous context
- Predefine variables before using them to avoid syntax errors
{task_specific_output_tips}
```

Fig. 8. Planner LLM prompt template. The planner receives this prompt at each search–verify–learn iteration, with placeholders filled from the task specification and feedback from the previous iteration.

R PROOF DETAILS

Lemma 5.1 (Completeness of Checker). *For any valid FGGM* G *with quantifier-free* Ψ_l, $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1, \ldots, x_n) \implies \big(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi_l(x_1, \ldots, x_n, y)\big)$.

Proof of Lemma 5.1. We prove the claim by structural induction on the quantifier-free formula Ψ_l. Fix arbitrary $(x_1, \ldots, x_n, y)$. We show:

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi_l(x_1, \ldots, x_n, y).$$

Base case. Suppose Ψ_l is an atomic predicate $P(x_1, \ldots, x_n, y)$ over terms constructed from computable library functions in $\mathcal{F}_c$. Under the substitution $(x_1, \ldots, x_n, y)$, $check_{\mathcal{A},\Phi,P}$ evaluates each term

by executing the corresponding functions in $\mathcal{F}_c$. Since these functions are computable $check_{\mathcal{A},\Phi,P}$ exactly computes the atomic predicate $P(x_1,\ldots,x_n,y)$ on $(x_1,\ldots,x_n,y)$ which reduces to T or F.

Inductive step. Assume the claim holds for formulas ϕ and ψ i.e. $check_{\mathcal{A},\Phi,\phi}(x_1,\ldots,x_n,y) \iff \phi(x_1,\ldots,x_n,y)$ and $check_{\mathcal{A},\Phi,\psi}(x_1,\ldots,x_n,y) \iff \psi(x_1,\ldots,x_n,y)$ We show it holds for compound formulas:

- If $\Psi_l = \neg\phi$, then by definition of $check_{\mathcal{A},\Phi_l,\Psi_l}$,

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) = \neg check_{\mathcal{A},\Phi,\phi}(x_1,\ldots,x_n,y).$$

  By the inductive hypothesis,

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \iff \neg\big(check_{\mathcal{A},\Phi,\phi}(x_1,\ldots,x_n,y)\big) \iff \neg\big(\phi(x_1,\ldots,x_n,y)\big) \iff \Psi_l(x_1,\ldots,x_n,y)$$

- If $\Psi_l = \phi \wedge \psi$, then

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) = check_{\mathcal{A},\Phi,\phi}(x_1,\ldots,x_n,y) \wedge check_{\mathcal{A},\Phi,\psi}(x_1,\ldots,x_n,y),$$

  which, by the inductive hypothesis,

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \iff \big(check_{\mathcal{A},\Phi,\phi}(x_1,\ldots,x_n,y) \wedge check_{\mathcal{A},\Phi,\psi}(x_1,\ldots,x_n,y)\big) \iff \big(\phi(x_1,\ldots,x_n,y) \wedge \psi(x_1,\ldots,x_n,y)\big) \iff \Psi_l(x_1,\ldots,x_n,y)$$

- If $\Psi_l = \phi \vee \psi$, the argument is analogous tp $\Psi_l = \phi \wedge \psi$.

□

Lemma R.1 (Checker Soundness). $\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1,\ldots,x_n) \implies \big(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \implies \Psi_l(x_1,\ldots,x_n,y)\big)\big)$.

Proof. For quantifier-free Ψ_l soundness follows from Lemma 5.1. For Ψ_l with qunatifier $check_{\mathcal{A},\Phi_l,\Psi_l}$ returns true if $(\varphi_{\mathcal{A}} \wedge \Phi_l(x_1,\ldots,x_n)) \wedge \Psi_l(x_1,\ldots,x_n,y)$ is evalautes to true within timeout. Hence for any $(x_1,\ldots,x_n,y)$

$$\begin{aligned}
check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) &\implies (\varphi_{\mathcal{A}} \wedge \Phi_l(x_1,\ldots,x_n)) \wedge \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) &\implies \Phi_l(x_1,\ldots,x_n) \wedge \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \wedge \Phi_l(x_1,\ldots,x_n)) &\implies \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge \Phi_l(x_1,\ldots,x_n)) &\implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) \implies \Psi_l(x_1,\ldots,x_n,y))\\
\varphi_{\mathcal{A}} \implies (\Phi_l(x_1,\ldots,x_n)) &\implies ((check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) \implies \Psi_l(x_1,\ldots,x_n,y))) \qquad (27)
\end{aligned}$$

From Eq 27 it follows , $\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1,\ldots,x_n) \implies \big(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \implies \Psi_l(x_1,\ldots,x_n,y)\big)\big)$. □

Theorem 5.2 (Valid Local contract). *For any valid FGGM* G *with* (Φ_l,Ψ_l) *and parametric GM* f_{G}^{Θ}, $\varphi_{\mathcal{A}} \implies \big(\forall\theta \in \Theta, \forall x_1 \in T_1,\ldots,\forall x_n \in T_n.\ \Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n, id_{\mathrm{G}}(x_1,\ldots,x_n))\big)$.

Proof. Let $r = id_{\mathrm{G}}(x_1,\ldots,x_n)$. Based on the rejection sampler with the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ (Fig 2). r is returned from the if branch **if** $check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)\{$**return** $y;\}$ or from the fallback $r = f_d(x_1,\ldots,x_n,y)$. Using the semantics of an if block, the following condition always holds for the output r.

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r) \vee (r = f_d(x_1,\ldots,x_n))) \qquad (28)$$

Eq. 29 follows from the validity of the fallback f_d, and Eq. 19 follows from the soundness of the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ (lemma R.1). To simplify the notation, we used $(\forall x_i \in T_i)$ to denote $(\forall x_1 \in T_1, \ldots, \forall x_n \in T_n)$.

$$\varphi_{\mathcal{A}} \implies \forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \wedge (r = f_d(x_1,\ldots,x_n)) \implies \Psi_l(x_1,\ldots,x_n,r) \tag{29}$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).(\Phi_l(x_1,\ldots,x_n) \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r)) \implies \Psi_l(x_1,\ldots,x_n,r)) \tag{30}$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n,r)) \quad \text{Using Eq. (28, 29, 30)}$$

□

Theorem 5.3. *If* $\mathsf{P}_{\{\Theta\}} \neq \bot$ *with FGGM set* $\mathcal{G}$*, then* $\forall\theta_1 \in \Theta_1,\ldots,\forall\theta_k \in \Theta_k.\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n.\Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,\mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1,\ldots,x_n))\big)$.

Proof. Let, $\mathcal{G} = \{\mathsf{G}_1,\ldots,\mathsf{G}_k\}$ and $\mathcal{A}_{\mathcal{G}}$ denote the set of first-order sentences defining all the local contracts as shown below.

$$\mathcal{A}_{\mathcal{G}} = \{\varphi_{\mathsf{G}_1},\ldots,\varphi_{\mathsf{G}_k}\} \text{ where } \varphi_{\mathsf{G}_i} \text{ defined below}$$

$$\varphi_{\mathsf{G}_i} : \forall x_1 \in T_1,\ldots,x_n \in T_n.\mathsf{G}_i^{\Phi_l}(x_1,\ldots,x_n) \implies \mathsf{G}_i^{\Psi_l}(x_1,\ldots,x_n,\mathsf{G}_i(x_1,\ldots,x_n))$$

$$(\varphi_{\mathcal{A}} \implies (\forall\theta_1 \in \Theta_1.\varphi_{\mathsf{G}_1})) \wedge \cdots \wedge (\varphi_{\mathcal{A}} \implies (\forall\theta_k \in \Theta_k.\varphi_{\mathsf{G}_k})) \quad \text{from Theorem 5.2}$$

$$\big(\varphi_{\mathcal{A}} \implies \wedge_{i=1}^{k}(\forall\theta_i \in \Theta_i.\varphi_{\mathsf{G}_i})\big) \tag{31}$$

If $\mathsf{P}_{\{\Theta\}} \neq \bot$ then $\mathcal{V}_{\mathsf{C}}[\Phi,\Psi]$ with $\mathsf{C} = ()$ enusres

$$\begin{aligned}(\mathsf{P}_{\{\Theta\}} \neq \bot) \implies ((\varphi_{\mathcal{A}} \wedge (\wedge_{i=1}^{k}\forall\theta_i \in \Theta_i.\varphi_{\mathsf{G}_i})) \implies \\ \forall x_1 \in T_1 \ldots \forall x_n \in T_n.\Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,\mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1,\ldots,x_n))\end{aligned} \tag{32}$$

$$\begin{aligned}(\mathsf{P}_{\{\Theta\}} \neq \bot) \implies ((\varphi_{\mathcal{A}} \implies \\ \forall\theta_1 \in \Theta_1,\ldots,\forall\theta_k \in \Theta_k,\forall x_1 \in T_1 \ldots \forall x_n \in T_n. \\ \Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,\mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1,\ldots,x_n)) \text{ Using Eq. 31 and Eq. 32}\end{aligned}$$

□

Theorem 5.4 (Soundness). *If* $f^* \neq \bot$ *then,* $\varphi_{\mathcal{A}} \implies \forall x_1 \in T_1,\ldots,\forall x_n \in T_n.\Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,f^*(x_1,\ldots,x_n))$.

Proof. $f^* \leftarrow \arg\min_{f\in\mathcal{P}} \sum_{(x,y)\in\mathsf{D}} \mathsf{L}(x,y,f(x))$. If $f^* \neq \bot$, then by definition $(\mathcal{P} \neq \{\}) \wedge (f^* \in \mathcal{P})$. By Theorem 5.3, every program $f \in \mathcal{P}$ that satisfies $\varphi_{\mathcal{A}}$ also satisfies

$$\forall x_1 \in T_1,\ldots,\forall x_n \in T_n,\ \Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,f(x_1,\ldots,x_n)).$$

□

Theorem 5.5 (Sufficient Success Condition). *If (i)* L *penalizes constraint violations, i.e.,*$\forall x \in T_i,\ \forall y,y' \in T_o.\ (\Phi(x) \wedge \neg\Psi(x,y) \wedge \Psi(x,y')) \implies \mathsf{L}(x,\_,y') < \mathsf{L}(x,\_,y)$*,, (ii)* Ψ *is quantifier-free, (iii) there exists a non-parametric program* $f_d \in \mathsf{S}(G,\mathcal{F}_c)$ *such that* f_d *satisfies* (Φ,Ψ)*. Then, for any type-correct generative model* $f_n^\theta : T_i \to T_o$ *with initial parameters and any prompting program* f_p*, there exists a program* $f \in \mathsf{S}(G,\mathcal{F})$ *such that: (1)* f *satisfies* (Φ,Ψ)*, and (2)* $\mathsf{L}(f) \leq \mathsf{L}(f_n^\theta)$*. Moreover, if there exists* $(x,\_) \in \mathsf{D}$ *such that* $\neg\Psi(x,f_n^\theta(x))$*, then* $\mathsf{L}(f) < \mathsf{L}(f_n^\theta)$ *where* $L(f) = \sum_{(x,\_)\in\mathsf{D}} \mathsf{L}(x,\_,f(x))$.

Proof. Let $f_n^\theta : T_1 \times \cdots \times T_n \to T_o$ be any type-correct generative model. By assumption (iii), there exists $f_d \in \mathsf{S}(G,\mathcal{F}_c)$ such that

$$\varphi_{\mathcal{A}} \implies \forall x_1,\ldots,x_n.;\Phi(x_1,\ldots,x_n) \implies \Psi(x_1,\ldots,x_n,f_d(x_1,\ldots,x_n)).$$

Construction. Define

$$\mathsf{G} = (id, f_n^\theta, \tau, \Phi_l, \Psi_l, f_p, f_d, info),$$

with $(\Phi_l, \Psi_l) := (\Phi, \Psi)$ and any well-typed f_p. Since f_d satisfies the contract and Ψ is quantifier-free, G is valid and $check_{\mathcal{A},\Phi_l,\Psi_l}$ is complete (Lemma 5.1).

Define the program $f \in \mathsf{S}(G, \mathcal{F})$ as:

$$\textbf{function } f(x_1 : T_1, \ldots, x_n : T_n) : T_o \ \{\textbf{return } id(x_1, \ldots, x_n);\}.$$

Correctness. For any $(x_1, \ldots, x_n)$ such that $\Phi(x_1, \ldots, x_n)$ holds, by Theorem 5.2,

$$\Psi(x_1, \ldots, x_n, f(x_1, \ldots, x_n)).$$

Output of f. Fix $(x_1, \ldots, x_n)$ with $\Phi(x_1, \ldots, x_n)$. By completeness,

$$check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi(x_1, \ldots, x_n, y).$$

Hence,

$$f(x_1, \ldots, x_n) = \begin{cases} f_n^\theta(x_1, \ldots, x_n) & \text{if } \Psi(x_1, \ldots, x_n, f_n^\theta(x_1, \ldots, x_n)), \\ f_d(x_1, \ldots, x_n) & \text{otherwise.} \end{cases}$$

Loss comparison. For any $(x, \_) \in \mathrm{D}$:

Case 1: $\Psi(x, f_n^\theta(x))$. Then $f(x) = f_n^\theta(x)$ and

$$\mathsf{L}(x, \_, f(x)) = \mathsf{L}(x, \_, f_n^\theta(x)).$$

Case 2: $\neg\Psi(x, f_n^\theta(x))$. Then $f(x) = f_d(x)$ and $\Psi(x, f(x))$ holds. By assumption (i),

$$\mathsf{L}(x, \_, f(x)) < \mathsf{L}(x, \_, f_n^\theta(x)).$$

Thus,

$$\forall (x, \_) \in \mathrm{D}. \quad \mathsf{L}(x, \_, f(x)) \le \mathsf{L}(x, \_, f_n^\theta(x)),$$

which implies

$$\mathsf{L}(f) \le \mathsf{L}(f_n^\theta).$$

Strict improvement. If $\exists (x, \_) \in \mathrm{D}$ such that $\neg\Psi(x, f_n^\theta(x))$, then Case 2 holds for at least one point, yielding

$$\mathsf{L}(f) < \mathsf{L}(f_n^\theta).$$

Conclusion. The constructed program f satisfies (Φ, Ψ) and achieves no greater loss than f_n^θ, with strict improvement when violations occur. Note the program verification setup satisfies this sufficient condition. □