SEVerA: Verified Self-Evolving Agents with Specification Guidance

Debangshu Banerjee [*1] **Changming Xu** [*1] **Eugene Ie** [2] **Ming Zhang** [2] **Daiyi Peng** [2] **Chu-Cheng Lin** [2]
Gagandeep Singh [1]

Abstract

Recent works have demonstrated the effectiveness of self-evolving LLM agents across a range of tasks, including program repair and scientific discovery. In this paradigm, a planner LLM synthesizes an agent program (e.g., a model harness) that invokes parametric models, including LLMs, smaller neural networks, and external tools, which are then tuned for each task to improve performance. However, existing self-evolving frameworks provide no formal guarantees of safety or correctness, even though the synthesized programs are executed autonomously on unseen inputs. 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), in which each model call is wrapped in a rejection sampler with a verified fallback, ensuring that the model always satisfies a first-order output contract for any input and parameter setting. Building on FGGM, we present SEVerA (*S*elf-*E*volving *Ver*ified *A*gents), a three-stage *Search→Verify→Learn* framework. We evaluate SEVerA on policy-compliant agentic tool use (τ^2-bench), automated program verification, scientific discovery, and symbolic math synthesis. SEVerA achieves zero constraint violations while improving task performance over unconstrained and state-of-the-art baselines, demonstrating that formal behavioral constraints both prune bad candidate programs and steer synthesis toward higher-quality agents.

[*]Equal contribution [1]University of Illinois Urbana-Champaign, USA [2]Google, USA. Correspondence to: Debangshu Banerjee <db21@illinois.edu>, Changming Xu <cx23@illinois.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

1. Introduction

Recent works leverage the code-generation capabilities of large language models (LLMs) to automatically synthesize LLM-based agents, expressed as programs, from natural language task descriptions (Wang et al., 2024; 2023; Hu et al., 2025; Zhuge et al., 2024; Gao et al., 2025). Given a user-defined task, a planner LLM generates a program (e.g., a model harness) 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 powerful, this paradigm raises serious safety and reliability concerns: the synthesized programs are executed autonomously on unseen inputs without human supervision. Self-evolving frameworks (Zhou et al., 2024; Fang et al., 2025; Wang et al., 2026; Tian et al., 2025) further complicate formal safety verification by fine-tuning the parameters of models embedded in the synthesized programs, so any correctness guarantee 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 the modified program with proposed annotations passes verification, inflating task accuracy (Banerjee et al., 2026). In code repair, agents delete failing tests rather than fixing the underlying bugs (Zhong et al., 2025). In agentic tool use, unconstrained agents violate domain-specific policies, such as refund eligibility and booking-modification rules, on 65–76% of interactions (Kamath et al., 2025). In deployment, rogue agents have been observed bypassing security protections in the wild (Guo et al., 2024; Lee et al., 2025; Guardian, 2026). 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 are not sufficient to 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 (Alur et al., 2013) and the flexibility and performance benefits of self-evolving agentic frameworks.

Key Challenges:

- **Balancing Safety and Performance:** A safe agent synthesis algorithm must ensure formal correctness while improving task-specific performance. Existing work on deductive program synthesis (Alur et al., 2013; Kim et al., 2021; Solar-Lezama, 2013) provides formal guarantees but typically does not optimize task-specific performance objectives. Meanwhile, popular gradient-based optimization methods for LLMs, such as GRPO (Shao et al., 2024), 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 and to verify that an agentic program satisfies the user-provided behavioral specification. Recent work on constrained decoding (Ugare et al., 2025b; Park et al., 2024; Banerjee et al., 2025) can enforce context-free grammar-based syntactic constraints and limited semantic constraints, but is restricted in expressivity and requires modifying decoding internals, limiting applicability to open-source models. Constrained decoding is also known to distort the output distribution (Park et al., 2024; Banerjee et al., 2025).

Our Contributions:

- We introduce Formally Guarded Generative Models (FGGM), which define and enforce local contracts on model calls through formal input–output specifications expressed in first-order logic. FGGMs operate solely on LLM output strings and apply to both open- and closed-source models. They retain the formal guarantees of deductive synthesis while exploiting the performance gains from gradient-based parameter optimization.

- We introduce SEVerA: the first self-evolving agent synthesis algorithm with verifiable guarantees. The approach has three stages: (a) *Search*, where a planner LLM samples program strings in a verifier-aware language such as Dafny, with FGGMs establishing local contracts on the embedded model calls; (b) *Verify*, where the language's built-in verifier proves that sampled programs satisfy all specified contracts over all parameters of the embedded parametric models; and (c) *Learn*, where verified parametric programs reduce the constrained learning problem to unconstrained optimization, enabling scalable gradient-based training without sacrificing correctness.

- We prove SEVerA is *sound*: any agent returned satisfies the behavioral specification for all inputs and all parameter values (Theorem 3.2). We also establish a *sufficient condition* (Theorem 3.3) 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.

- We evaluate SEVerA on tasks spanning agentic tool use, LLM-assisted program verification, symbolic math, and constrained symbolic regression. SEVerA 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 on τ^2-bench's airline domain using Qwen3-8B. Notably, on τ^2-bench airline, SEVerA with Qwen3-8B even beats Agent-C (Kamath et al., 2025) with Claude Sonnet 4.5 (52.6 vs. 47.3). **These results demonstrate that behavioral constraints not merely enforce safety but actively prune the search space and steer synthesis toward higher-quality agents.**

2. Background and Problem Formulation

Notation. We use Σ to denote a finite alphabet. LLMs output finite strings over Σ. We use lowercase letters (x) for constants, bold letters $(\boldsymbol{x})$ for strings, and capital letters (X) for sets. We treat neural networks as a restricted form of generative model that produces a single unit-length output.

Rejection Sampling. Rejection sampling is a Monte Carlo method for sampling from a target distribution π_t when direct sampling is hard. Instead, it uses a proposal distribution π_p whose support covers $S(\pi_t)$. Candidates are sampled from π_p and accepted/rejected so that no accepted sample lies outside $S(\pi_t)$. The support set is $S(\pi) = \{x \in \Omega \mid D_\pi(x) > 0\}$ where D_π is the density of π (Appendix E).

Self-Evolving LLM Agents. We focus on LLM agents modeled as programs $f : T_i \to T_o$ that, given user input $x \in T_i$, compute output $y = f(x)$. The program invokes parametric models $\mathcal{F}_p$ (e.g., LLMs) and tools $\mathcal{F}_c$ (e.g., SMT solvers); we write $\mathcal{F} = \mathcal{F}_p \cup \mathcal{F}_c$ for the invokable library functions. In the self-evolving paradigm, f is generated by a planner LLM from task-specific instructions. Given dataset $\mathrm{D} \subseteq T_i \times T_o$ and pointwise loss $\mathsf{L} : T_i \times T_o \times T_o \to \mathbb{R}^+$, the goal is to minimize the aggregated loss. Let CFG G define the syntactically valid programs $L(G) \subseteq \Sigma^*$, and let $\mathsf{S}(G, \mathcal{F})$ denote the search space, including all parametric values of the models in $\mathcal{F}_p$. Self-evolving frameworks aim to find:

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

Agent Synthesis with Formal Constraints. Eq. 1 provides no guarantee about f^* on unseen inputs outside D. In contrast, deductive program synthesis (Alur et al., 2013; Solar-Lezama, 2013) controls program generation just through

formal input $\Phi : T_i \to \{T, F\}$ and output specifications $\Psi : T_i \times T_o \to \{T, F\}$ with no performance incentive. The constrained learning problem in Eq. 2 captures both correctness specification and performance objective. The central question is whether Eq. 2 can be solved with correctness proofs while retaining the performance benefits from Eq. 1.

$$f^* = \arg\min_{f \in \mathsf{S}(G,\mathcal{F})} \overbrace{\tfrac{1}{|\mathrm{D}|}\textstyle\sum_{(x_i,_)\in \mathrm{D}} \mathsf{L}(x_i, _, f(x_i))}^{\text{soft objective}} \quad \text{s.t.} \overbrace{\forall x.\, \Phi(x) \implies \Psi(x, f(x))}^{\text{hard constraints}} \tag{2}$$

Examples of formal constraints (Φ, Ψ)**.** Hard constraints detect and mitigate errors during synthesis that would not be revealed by performance evaluation on a fixed dataset D. We highlight four representative tasks; full discussions are in Appendix B.2. ⓐ **Scientific Discovery:** constrained symbolic regression encodes prior knowledge about an unknown ground-truth formula f_{gt} as symbolic bounds or asymptotic behavior (Błądek & Krawiec, 2019; Haider & Kronberger, 2022; Li et al., 2019); constraints prevent overfitting under noise. ⓑ **Program Verification:** given a Dafny program with predefined specifications, the agent synthesizes annotations (loop invariants, ranking functions, assertions) (Loughridge et al., 2025; Sun et al., 2024; Mugnier et al., 2025). Banerjee et al. (2026) show that frontier LLMs cheat by altering the input program; AST-based diff checks serve as hard constraints to enforce semantic equivalence. ⓒ **Constrained LLM Generation:** formal grammars or static analyzers (Ugare et al., 2025b; Park et al., 2024; Mündler et al., 2025; Nagy et al., 2026) define Ψ for tasks like generating symbolic math expressions from natural language. ⓓ **Agentic Tool Use:** conversational agents must respect domain-specific policies (refund eligibility, booking-modification rules) (Barres et al., 2025). Agent-C (Kamath et al., 2025) provides a DSL for temporal properties translated into LTL and checked at each tool call.

3. The SEVerA Framework

Figure 1 gives a high-level overview of SEVerA for solving Eq. 2. This requires simultaneously optimizing task-specific performance while ensuring that the resulting program provably satisfies the behavioral constraints. This is difficult because the search space contains both discrete program structure and continuous model parameters, and because LLMs are inherently unreliable, making conformance with behavioral specifications hard to ensure.

To address this, SEVerA combines the best of both worlds: constraint-guided program search from deductive synthesis and unconstrained gradient-based parameter optimization. It employs a CEGIS-style loop in which a planner LLM proposes candidate parametric programs and a verifier checks their correctness against the behavioral specification. Once a parametric program is verified, SEVerA applies unconstrained gradient-based optimization to tune the underlying GM parameters and reduce L. The Formally Guarded Generative Model (FGGM) is the critical link between these two strategies: it binds each GM call to explicit local input–output contracts and a verified non-parametric fallback, ensuring the contracts hold regardless of the underlying GM parameters. The verifier leverages these local contracts to prove the correctness of the parametric program against the behavioral specification. Beyond correctness, the contracts also extract local learning objectives by characterizing expected outputs of each GM call, enabling gradient-based optimization to improve conformance.

Next, we present an example instantiation of SEVerA and provide an overview of its key components, algorithms, and theoretical results in Sections 3.2–3.4.

3.1. Example Instantiations

LLM-Assisted Verification. Given a Dafny program with pre-encoded specifications, the agent synthesizes annotations enabling verification (Sun et al., 2024; Loughridge et al., 2025). The agent converts any input program to an annotated program $(p : \mathit{str}) \to \mathit{str}$. On input-output pairs $y = f(x)$ the pointwise loss L assigns 1 to unverified outputs and 0 to verified ones. Hard constraints ensure that, for any valid input program, the output remains parsable and is not modified beyond annotations. Here, $\mathit{parse}, \mathit{noDiff}, \mathit{verify}$ are library functions

$$\begin{aligned} \mathsf{L}(\boldsymbol{x}, _, \boldsymbol{y}) &= 1 - \mathbb{I}\big(\mathit{parse}(\boldsymbol{y}) \wedge \mathit{noDiff}(\boldsymbol{x}, \boldsymbol{y}) \wedge \mathit{verify}(\boldsymbol{y})\big) \\ \Phi(\boldsymbol{x}) &: \ \mathit{parse}(\boldsymbol{x}) \\ \Psi(\boldsymbol{x}, f(\boldsymbol{x})) &: \ \mathit{parse}(f(\boldsymbol{x})) \wedge \mathit{noDiff}(\boldsymbol{x}, f(\boldsymbol{x})) \end{aligned} \tag{3}$$

Although parsing and diff checks could be folded into L, they must hold for *all* inputs, not just those in D.

Search Space and Library Functions. SEVerA uses a restricted subset of Dafny supporting four basic types $\{\mathit{int}, \mathit{bool}, \mathit{real}, \mathit{str}\}$, conditional blocks, while loops, and function calls. The library $\mathcal{F}$ partitions into non-parametric tools $\mathcal{F}_c$ (with user-provided contracts) and parametric models $\mathcal{F}_p$ whose contracts are synthesized via FGGMs. Axioms $\mathcal{A}$ encode known properties of library functions as universally quantified first-order sentences and assist verification. Full formalization is deferred to Appendix A.

3.2. Formally Guarded Generative Models

The FGGM setup binds each 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, these contracts are used to prove correctness against (Φ, Ψ). Our design ensures three essential properties: ❶ *Flexibility:* the planner LLM synthesizes custom first-order formulas over terms involving non-parametric functions and predicates from $\mathcal{F}_c$. ❷ *Parameter-Independent Correctness:* provided

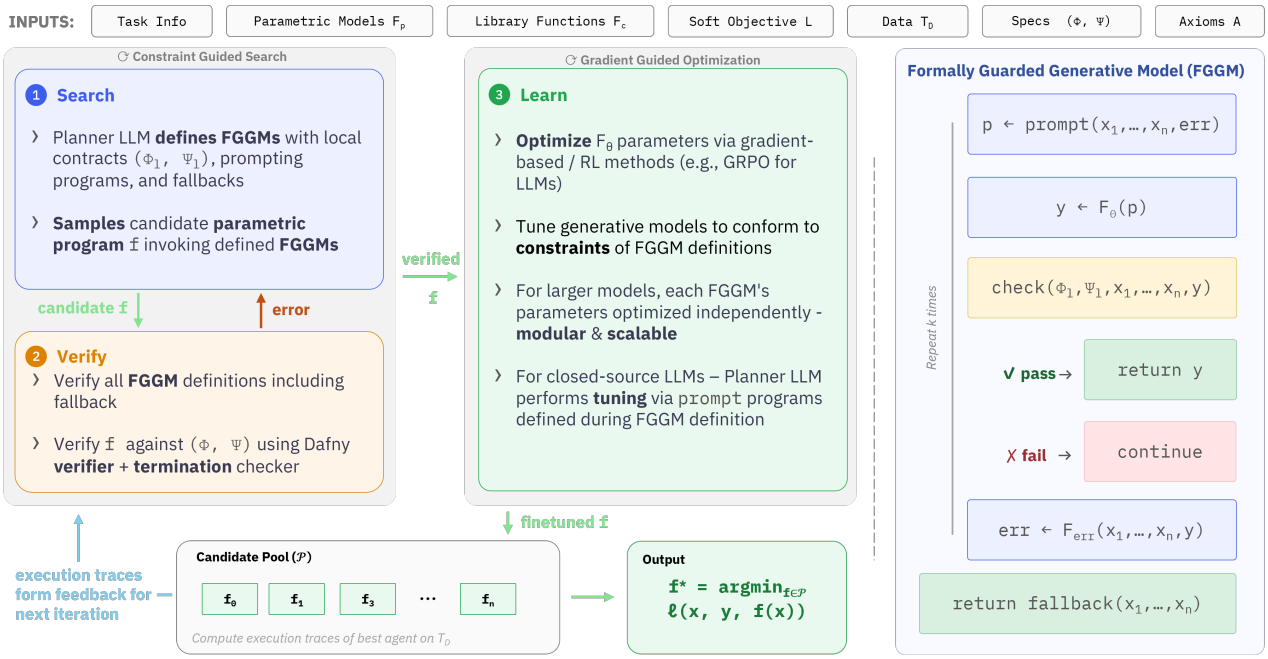

Figure 1. Overview of SEVerA, which operates in three steps. (1) **Search:** given task information, library functions $\mathcal{F}_c$, parametric generative models $\mathcal{F}_p$, and specifications (Φ, Ψ), the planner LLM outputs a parametric agentic program in which all formal output specifications of GM calls are defined using the FGGM setup. (2) **Verify:** SEVerA first verifies all FGGM definitions; once verified, it uses their local contracts to check the program against (Φ, Ψ). If all checks pass, the unconstrained learning step is invoked; otherwise, an error message is returned to the planner. (3) **Learn:** the parameters of the underlying GMs within each FGGM are tuned to improve conformance with local contracts and to reduce the task loss L over D. After tuning, SEVerA maintains a pool of verified fine-tuned agents and uses their execution traces on D to generate new candidates via step (1). The agent with lowest L on D is returned.

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

Figure 2. Auto-synthesized guarded GM with a stateful rejection sampler (1) and fallback (2).

the FGGM is well-formed, the contracts hold irrespective of the GM parameters, so gradient-based parameter optimization never breaks correctness. ❸ *Local Learning Objective:* the contracts (Φ_l, Ψ_l) characterize the expected output and provide a local objective for gradient-based optimization.

Intuition. For each FGGM with contracts (Φ_l, Ψ_l) and underlying parametric GM $\mathcal{L}_\Theta$, the planner synthesizes three non-parametric programs using only functions from $\mathcal{F}_c$. The first is a fallback f_d that satisfies (Φ_l, Ψ_l) and guards against edge cases where the GM hallucinates. Without a verified fallback, the rejection sampler could exhaust all samples; the fallback is what makes the contract hold unconditionally. Although f_d may lack the GM's task-solving capability, it prevents the agent from entering unsafe states. The second is a prompting program f_p that constructs the input to the GM (analogous to prompt-tuning techniques (Khattab et al., 2024; Beurer-Kellner et al., 2023)). The third, f_{err}, explains why a sample y is rejected, making the rejection sampler stateful and preventing it from repeating the same mistake.

Given the planner's $(\Phi_l, \Psi_l, f_p, f_d, f_{err})$, SEVerA automatically constructs a rejection sampler treating $\mathcal{L}_\Theta$ outputs as samples from π_p. The local specs define the support set of π_t, assigning zero probability to violating outputs. The sampler draws $K \geq 1$ samples and falls back to f_d if all are rejected (Fig. 2). f_p and optional conformance tuning improve acceptance, reducing reliance on f_d. The full FGGM syntax, well-formedness conditions, and computability of the output checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ are formalized in Appendix B.

For any valid FGGM, the synthesized rejection sampler satisfies the contract for all well-typed inputs and *all* parameters $\theta \in \Theta$, as stated in Theorem 3.1.

Theorem 3.1 (Valid Local Contract)**.** *For any valid FGGM*

G *with* (Φ_l, Ψ_l) *and parametric GM* f^{Θ}_{G}, $\varphi_{\mathcal{A}} \implies \big(\forall \theta \in \Theta, \forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\ \Phi_l(x_1, \ldots, x_n) \implies \Psi_l(x_1, \ldots, x_n, id_{\mathsf{G}}(x_1, \ldots, x_n))\big)$.

The proof and a completeness lemma for the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ on quantifier-free Ψ_l (Lemma B.3) are in Appendix W.

Example FGGM definitions. For LLM-assited verification the example FGGM defines $dafnyAnnotate$ with signature $(p : str) \to str$, an underlying LLM, $\Phi_l : parse(p)$, and Ψ_l in Eq. 4. The fallback safely returns the original program p, satisfying Ψ_l via reflexivity of $noDiff$.

$$\begin{aligned} &\Psi_l : parse(p) \wedge noDiff(p, f(p)) \\ &f_d : \textbf{function } f_d(\text{p, y}) \ \{\textbf{return } p;\} \end{aligned} \tag{4}$$

Design Considerations. We discuss why alternative approaches of providing guarantees on GM outputs fall short of FGGM. Constrained decoding (Ugare et al., 2025b; Park et al., 2024; Nagy et al., 2026; Banerjee et al., 2025) requires modifying decoding internals (limiting closed-source compatibility) and restricts constraints to syntactic forms. Post-hoc filtering without a fallback provides no guarantees. Fully verifying a parametric agentic program is intractable. FGGM addresses all three: it operates only on outputs, includes a verified fallback for unconditional correctness, and ensures guarantees are parameter-independent.

3.3. Search→Verify→Learn

SEVerA employs a three-step strategy:

(1) Search. The planner LLM defines FGGMs with prompting programs f_p, then samples candidate parametric programs that invoke these FGGMs. Direct calls to parametric GMs are disallowed: each must be wrapped within a FGGM.

(2) Verify. SEVerA checks all FGGM definitions for well-formedness and verifies the candidate program against (Φ, Ψ) using Dafny's built-in verifier. Failures produce error feedback (either in FGGM definition or the sampled program) that the planner uses to refine the candidate (CEGIS-style). Due to the FGGM setup, any verified program satisfies (Φ, Ψ) for *all* parameter values (Theorem C.1).

(3) Learn. Verified programs reduce the constrained problem to unconstrained gradient-based optimization. SEVerA optimizes the global task loss L jointly with local conformance losses $\mathsf{L}_{\Phi_l,\Psi_l}$ from each FGGM (formal definition in Appendix C.2, Eq. 11). The conformance loss measures the probability that a sampled GM output violates the local contract; optimizing it improves the rejection sampler's acceptance rate and reduces reliance on f_d. For accessible parameters (e.g., open-weight LLMs), this can be optimized with GRPO (Shao et al., 2024); for closed-source models, the learning step is skipped and improvements come solely from f_p.

Candidate pool and feedback. SEVerA maintains a pool of verified, fine-tuned agents (Fig. 1). The best-performing candidate's execution traces on D guide subsequent search iterations via the planner LLM. Once the candidate budget is exhausted, the agent in the pool with the lowest task loss is returned. SEVerA algorithms (Algorithms 1 and 2) and full formal details are in Appendix C.

SEVerA combines constraint-guided search to avoid bad agentic programs with gradient-based optimization to improve model parameters. Balancing these two strategies provides both provable reliability and performance.

3.4. Theoretical Results

SEVerA is sound: any returned f^* satisfies the behavioral specification.

Theorem 3.2 (Soundness)**.** *If* $f^* \neq \bot$, *then* $\varphi_{\mathcal{A}} \implies \forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, f^*(x_1, \ldots, x_n))$.

We also establish a sufficient condition under which a verified agent outperforms any unconstrained GM call in terms of performance. This follows from the fact that the fallback is triggered only when all samples from the GM are guaranteed to violate the constraints.

Theorem 3.3 (Sufficient Success Condition)**.** *If (i)* L *penalizes constraint violations:* $\forall x \in T_i, \forall y, y' \in T_o.\ (\Phi(x) \wedge \neg\Psi(x, y) \wedge \Psi(x, y')) \implies \mathsf{L}(x, _, y') < \mathsf{L}(x, _, y)$*; (ii)* Ψ *is quantifier-free; and (iii) there exists a non-parametric* $f_d \in \mathsf{S}(G, \mathcal{F}_c)$ *satisfying* (Φ, Ψ)*. Then for any type-correct GM* $f^{\theta}_n : T_i \to T_o$ *with initial parameters and any* f_p*, there exists* $f \in \mathsf{S}(G, \mathcal{F})$ *such that (1)* f *satisfies* (Φ, Ψ) *and (2)* $\mathsf{L}(f) \leq \mathsf{L}(f^{\theta}_n)$*, with strict improvement whenever* f^{θ}_n *violates* Ψ *on some* $(x, _) \in \mathrm{D}$.

The three assumptions are mild in practice: (i) holds for any reasonable loss in constrained settings, (ii) ensures completeness of the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ on concrete inputs, and (iii) is a natural existence requirement on the library. Theorem 3.3 guarantees that the search space contains agents that dominate unconstrained GM calls; finding them depends on the planner's search budget. Full proofs are in Appendix W.

4. Evaluation

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

4.1. Experimental Setup

Tasks. (1) *LLM-Assisted Program Verification (Dafny-Bench).* The agent receives a Dafny program with pre-encoded specs and synthesizes annotations enabling verification (Loughridge et al., 2025). The behavioral spec requires the output to remain parsable and equivalent to the input modulo annotations. Underlying LLM: Claude Sonnet 4.5 (Anthropic, 2025) (closed-source, no parameter tuning). (2) *Symbolic Math Synthesis (GSM-Symbolic)* (Mirzadeh et al., 2024). The behavioral spec enforces syntactic validity with respect to a formal grammar; performance measures whether the answer is equivalent to the ground truth. (3) *Agentic Tool Use (τ^2-bench)* (Barres et al., 2025), retail and airlines domains. The hard constraint is enforced per tool call (Agent-C (Kamath et al., 2025) checker for temporal policy compliance); pass rate is measured over multi-turn traces. (4) *Constrained Symbolic Regression (SymReg)*: 35 instances $\times$ 2 noise levels (5%, 10%) from (Haider & Kronberger, 2022; Błądek & Krawiec, 2019), with 600 noisy training samples and 400 test samples per instance, and behavioral specs encoding symbolic bounds.

Implementation. All experiments use NVIDIA A100 GPUs (40 GB). The pipeline: planner-guided search with budget $\Delta = 10$, FGGM rejection sampling with $K = 5$, deductive verification with bounded verifier and SMT timeouts, and hyperparameter tuning when parameters are accessible. The planner is given 3 in-context ground-truth examples (2 for τ^2-bench), excluded from train/test splits. Once a program verifies, SEVerA uses Dafny's built-in transpiler (Dafny Language Community, 2023) to translate to Python; tuning runs on the transpiled code. For GSM-Symbolic, the learning step uses GRPO with LoRA (Hu et al., 2022) on Qwen3-8B (Team, 2025). Splits, prompts, feedback, and hyperparameters are in Appendix T, V, U.

Baselines. For Dafny, τ^2-bench, and SymReg, we use SOTA hand-crafted agents (e.g., DafnyBench baseline, Agent-C, PySR (Cranmer, 2023), LLM-SR (Shojaee et al., 2025)). For GSM-Symbolic, we use the SOTA constrained decoder CRANE (Banerjee et al., 2025). We also compare to self-evolving frameworks without constraints, using the same library functions for fairness.

Verification success. For program verification, tool use, and GSM-Symbolic, SEVerA consistently finds a valid program within 10 attempts. For SymReg, it produces a verified program in 33/35 cases; baselines generate at least 11 cases that violate the spec on test inputs.

4.2. Main Results

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

Table 1. Dafny program verification results. *Verif. & NoDiff (ND)* requires both verification and the AST-based diff check; *Verif.* counts verification alone.

Dataset	Method	Ver. & ND (%)↑	Ver. ↑	Vio.↓	Time
HEval Dafny	Claude 4.5	73.7	76.8	8.1	9.8
	DafnyBench	86.9	87.9	4.0	16.1
	SEVerA w/o constr.	84.8	88.9	5.1	15.7
	SEVerA	97.0 (+10.1)	97.0	0.0	18.2
Dafny Bench	Claude 4.5	68.7	71.1	10.3	10.3
	DafnyBench	81.6	84.0	8.2	20.1
	SEVerA w/o constr.	79.2	84.8	7.9	18.4
	SEVerA	89.1 (+7.5)	89.1	0.0	25.6

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

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

LLM-Assisted Program Verification (Dafny).

Table 1 clearly illustrates the safety issue. The gap between *Verif.* and *Verif. & NoDiff* reveals how unconstrained agents inflate verification rates by modifying the input program: the LLM baseline reports 76.8% verification on HumanEvalDafny but only 73.7% with NoDiff, with 8.1% silent violations. The DafnyBench baseline and SEVerA without constraints reduce but do not eliminate this (4.0% and 5.1% violations). Only the full SEVerA pipeline, enforcing the NoDiff specification, achieves 0% violations on both benchmarks while attaining the highest *Verif. & NoDiff* rate (97.0% on HumanEvalDafny, 89.1% on DafnyBench). This is achieved without any parameter tuning: the gains come entirely from making constraints visible to the planner and enforcing them at synthesis time. The runtime overhead is modest ($\sim$1.9–2.5$\times$ vs. vanilla LLM generation), justified by the formal guarantees.

Agentic Tool Use (τ^2-bench).

Table 2 reports τ^2-bench. The Qwen3-8B baseline violates policies on 76.3% (retail) and 68.4% (airline) of interactions, with pass rates of just 11.3% and 13.2%. Agent-C and SEVerA both eliminate violations to 0%, confirming that formal policy enforcement is necessary. SEVerA without constraints raises pass rates to 49.4% and 44.7% but still incurs 10.3% and 25.5% violations. Enforcing constraints simultaneously eliminates all violations and improves pass rates further to 53.6% and 52.6%. By verifying every individual tool call, SEVerA explores a wider space of compliant strategies. Notably, SEVerA with the small open-weight

Table 3. GSM-Symbolic results with Qwen3-8B.

Method	Acc. (%)↑	Vio.↓	Time
Qwen3-8B	38.3	10.6	10.9
CRANE (Banerjee et al., 2025)	44.7	2.1	12.4
SEVerA (no tuning)	53.2 (+8.5)	0.0	18.8
SEVerA (with tuning)	66.0 (+21.3)	0.0	16.7

Table 4. Symbolic regression: % of instances violating the behavioral spec on test data.

Method	$\epsilon = 5\% \downarrow$	$\epsilon = 10\% \downarrow$
PySR	62.86%	62.86%
LLM-SR	31.43%	34.29%
SEVerA	0.00%	0.00%

Table 5. NMSE averaged over instances where the baseline does not violate the spec.

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

Qwen3-8B outperforms Agent-C with Claude Sonnet 4.5 on airline (52.6 vs. 47.3), and SEVerA is faster than Agent-C in both domains.

Symbolic Math Synthesis (GSM-Symbolic).

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

Table 3 compares against CRANE (Banerjee et al., 2025), a SOTA constrained-decoding method, on GSM-Symbolic with Qwen3-8B. The vanilla LLM achieves only 38.3% accuracy with 10.6% violations. CRANE improves to 44.7% with 2.1% violations.[1] SEVerA without parameter tuning achieves 53.2% accuracy and zero violations, an 8.5% gain over CRANE with strictly stronger guarantees. GRPO+LoRA fine-tuning under the same formal constraints provides a further 12.8% gain (66.0%) while preserving zero violations. The tuned model passes the FGGM checker more often on the first sample, reducing rejection iterations and even decreasing wall time (16.7s vs. 18.8s).

Constrained Symbolic Regression. We compare against the SOTA genetic-programming method PySR (Cranmer, 2023) and the self-evolving baseline LLM-SR (Shojaee et al., 2025). As shown in Table 4, PySR violates the spec on up to 62.86% of instances and LLM-SR on up to 34.29%. SEVerA finds a verified solution on 33/35 instances and produces 0% violations on the test set. Table 5 compares NMSE on the subset of instances where the baseline-generated programs satisfy the constraints; SEVerA achieves significantly lower NMSE.

[1] CRANE only ensures every decoding step produces a valid prefix of the target grammar.

Table 6. Constraint decomposition on GSM-Symbolic (Qwen3-8B). *Local* = FGGM contracts only; *Global* = behavioral spec only; *Full* = both.

Configuration	Acc. (%)↑	Vio.↓
Qwen3-8B	38.3	10.6
SEVerA (no tuning)	53.2 (+14.9)	0.0
SEVerA (local-only tuning)	55.3 (+17.0)	0.0
SEVerA (global-only tuning)	61.7 (+23.4)	0.0
SEVerA (full)	66.0 (+27.7)	0.0

Table 7. Fallback trigger rate and impact. *Trig.* = % of instances where the fallback produces the final output; *FB Acc.* = fallback as standalone baseline; *Gain* = gain over best baseline. *Gain** = gain over best baseline excluding fallback instances.

Benchmark	Trig.↓	FB Acc.	Gain	Gain*
HumanEvalDafny	5.1%	5.1%	+10.1	+8.1
DafnyBench	7.9%	29.3%	+7.5	+5.9
τ^2-Retail	15.6%	5.2%	+11.4	+10.6
τ^2-Airline	21.6%	5.4%	+13.2	+9.7
GSM-Symbolic	2.1%	0.0%	+21.3	+21.3

4.3. Constraint Decomposition

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

Table 6 decomposes the 12.8% gain from parameter tuning. Starting from SEVerA without tuning (53.2%), *local-only* tuning (FGGM conformance loss) yields a modest 2.1% gain to 55.3%; *global-only* tuning (task loss alone) yields a larger 8.5% gain to 61.7%; and the full pipeline combining both achieves 66.0%. The two training signals are complementary: global tuning improves answer correctness, while local tuning ensures per-call conformance.

4.4. Fallback Trigger Rate and Impact

> **RQ4 (Fallback).** *How often does the fallback fire, and how much of SEVerA's accuracy gain is attributable to it?*

Table 7 reports fallback statistics. The fallback alone achieves negligible accuracy (0.0–5.4%; DafnyBench is higher at 29.3% because returning the unmodified program trivially passes verification on already-correct inputs). Even after excluding fallback instances, SEVerA retains substantial gains over the best baseline (Gain*), confirming that the primary benefit of formal constraints is *search-space pruning* (making shortcuts infeasible forces synthesis toward higher-quality strategies) rather than the fallback itself. The fallback fires only when every sampled LLM output violates the specification, serving as a last-resort safety net.

4.5. Discussion

Across all four tasks, SEVerA achieves zero constraint violations on held-out test inputs while improving task performance over every baseline, confirming that formal behavioral specifications not only enforce safety but actively prune the candidate program space and steer synthesis toward higher-quality agents. SEVerA'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$\times$ slowdown vs. the LLM baseline. A current limitation is that the formulation is *resource-unaware*: behavioral specs constrain functional correctness but not LLM calls, token usage, or wall-clock budget. Extending SEVerA with resource-bound specifications (Knoth et al., 2019), e.g., encoding token or call budgets as additional FGGM-level hard constraints, is a promising direction.

Complete agent programs synthesized by SEVerA for Dafny-Bench, GSM-Symbolic, and τ^2-bench are included in Appendices N, O, and P.

5. Related Work

Self-Evolving Agents and Their Risks. Recent work explores automatic agent synthesis and self-improvement, from code-based action unification (Wang et al., 2024) and open-ended skill libraries (Wang et al., 2023) to architecture search (Hu et al., 2025; Zhuge et al., 2024; Gao et al., 2025), gradient-free symbolic updates (Zhou et al., 2024), co-evolution with world models (Fang et al., 2025), and reinforcement-learning-driven skill refinement (Wang et al., 2026; Tian et al., 2025). None of these provide formal behavioral guarantees, a gap made urgent by demonstrated risks of coding-agent exploits (Guo et al., 2024; Lee et al., 2025; Guardian, 2026), test-case exploitation (Zhong et al., 2025), and verification cheating (Banerjee et al., 2026). SEVerA addresses this by imposing and verifying hard formal specifications before deployment.

Runtime Monitoring and Shielding. Runtime approaches enforce safety on observed executions: shield synthesis from neural policies (Zhu et al., 2019) and temporal-logic runtime checking of tool-call sequences (Kamath et al., 2025). Both are limited to observed traces and cannot guarantee safety on unseen inputs. SEVerA verifies the synthesized program for *all* inputs and parameter values at synthesis time; monitoring checks can themselves be used to define the FGGM specification, as we do with (Kamath et al., 2025).

Neuro-Symbolic Program Synthesis. Neuro-symbolic methods, including neural program generation (Parisotto et al., 2016), type-directed differentiable programming (Valkov et al., 2018), wake-sleep library learning (Ellis et al., 2021), vision-language program composition (Surís et al., 2023), and LLM-guided symbolic regression (Shojaee et al., 2025), synthesize programs with neural components but lack formal correctness guarantees over the composed system. SEVerA 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 via grammar-based (Ugare et al., 2025b; Park et al., 2024; Ugare et al., 2025a), diffusion-based (Suresh et al., 2025), programming-abstraction (Beurer-Kellner et al., 2023; Khattab et al., 2024), reasoning-preserving (Banerjee et al., 2025), type-based (Mündler et al., 2025), and semantic (Nagy et al., 2026) techniques. These approaches require access to decoding internals (precluding closed-source models) and enforce per-token or per-output constraints, not program-level behavioral specifications. SEVerA's FGGM mechanism operates on 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 (syntax-guided (Alur et al., 2013), semantics-guided (Kim et al., 2021), counterexample-guided (Solar-Lezama, 2013), abstraction-guided (Guria et al., 2023), component-based (Feng et al., 2017)) provides strong correctness guarantees but targets deterministic, non-parametric programs. SEVerA bridges this gap with a CEGIS-style search–verify loop for program structure while using FGGMs to encapsulate parametric generative models within verified contracts.

6. Conclusion

We presented SEVerA, a framework for synthesizing self-evolving LLM agents with formal behavioral guarantees. By introducing Formally Guarded Generative Models (FGGMs), SEVerA binds each generative model call to a verified local contract backed by a rejection sampler with a provably correct fallback, ensuring specifications hold for all inputs and all parameter values. This 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. Across constrained symbolic regression, LLM-assisted Dafny verification, symbolic math synthesis, and policy-compliant agentic tool use, SEVerA achieves zero constraint violations while improving task performance over unconstrained and state-of-the-art baselines. 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.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. By introducing formal behavioral guarantees for self-evolving LLM agents, our work helps reduce the risks of deploying autonomous agents in safety-critical domains such as program verification, customer-service automation, and scientific discovery. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013. doi: 10.1109/FMCAD.2013.6679385.

Anthropic. System card: Claude sonnet 4.5. https://www-cdn.anthropic.com/963373e433e489a87a10c823c52a0a013e9172dd.pdf, 2025. Accessed: 2026-03-18.

Banerjee, D., Suresh, T., Ugare, S., Misailovic, S., and Singh, G. CRANE: Reasoning with constrained LLM generation. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=wKs9fHYxCV.

Banerjee, D., Bouissou, O., and Zetzsche, S. Dafnypro: Llm-assisted automated verification for dafny programs, 2026. URL https://arxiv.org/abs/2601.05385.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Beurer-Kellner, L., Fischer, M., and Vechev, M. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591300. URL https://doi.org/10.1145/3591300.

Błądek, I. and Krawiec, K. Solving symbolic regression problems with formal constraints. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pp. 977–984, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361118. doi: 10.1145/3321707.3321743. URL https://doi.org/10.1145/3321707.3321743.

Cranmer, M. Interpretable machine learning for science with pysr and symbolicregression.jl, 2023. URL https://arxiv.org/abs/2305.01582.

Dafny Language Community. Integrating dafny and python code. https://dafny.org/v3.10.0/DafnyRef/integration-py/IntegrationPython, 2023. Accessed: 2026-03-17.

Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. 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*, PLDI 2021, pp. 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Fang, T., Zhang, H., Zhang, Z., Ma, K., Yu, W., Mi, H., and Yu, D. Webevolver: Enhancing web agent self-improvement with coevolving world model. *arXiv preprint arXiv:2504.21024*, 2025.

Feng, Y., Martins, R., Van Geffen, J., Dillig, I., and Chaudhuri, S. 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*, PLDI 2017, pp. 422–436, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062351. URL https://doi.org/10.1145/3062341.3062351.

Gao, H., Liu, Y., He, Y., Dou, L., Du, C., Deng, Z., Hooi, B., Lin, M., and Pang, T. Flowreasoner: Reinforcing query-level meta-agents, 2025. URL https://arxiv.org/abs/2504.15257.

Guardian, T. Rogue ai agents published passwords and bypassed security protections. News investigation, 2026.

Guo, C., Liu, X., Xie, C., Zhou, A., Zeng, Y., Lin, Z., Song, D., and Li, B. Redcode: Risky code execution and generation benchmark for code agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=mAG68wdggA.

Guria, S. N., Foster, J. S., and Van Horn, D. Absynthe: Abstract interpretation-guided synthesis. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591285. URL https://doi.org/10.1145/3591285.

Haider, C. and Kronberger, G. 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, pp. 164–172, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-25311-9. doi: 10.1007/978-3-031-25312-6_19. URL https://doi.org/10.1007/978-3-031-25312-6_19.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Kamath, A., Zhang, S., Xu, C., Ugare, S., Singh, G., and Misailovic, S. Enforcing temporal constraints for llm agents, 2025. URL https://arxiv.org/abs/2512.23738.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., A, S. V., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

Kim, J., Hu, Q., D'Antoni, L., and Reps, T. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434311. URL https://doi.org/10.1145/3434311.

Knoth, T., Wang, D., Polikarpova, N., and Hoffmann, J. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 253–268, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314602. URL https://doi.org/10.1145/3314221.3314602.

Kronberger, G., de Franca, F. O., Burlacu, B., Haider, C., and Kommenda, M. Shape-constrained symbolic regression—improving extrapolation with prior knowledge. *Evolutionary Computation*, 30(1):75–98, 03 2022. ISSN 1063-6560. doi: 10.1162/evco_a_00294. URL https://doi.org/10.1162/evco_a_00294.

Lee, E., Kim, D., Kim, W., and Yun, I. Takedown: How it's done in modern coding agent exploits, 2025. URL https://arxiv.org/abs/2509.24240.

Leino, K. R. M. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pp. 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642175104.

Li, L., Fan, M., Singh, R., and Riley, P. Neural-guided symbolic regression with asymptotic constraints, 2019. URL https://arxiv.org/abs/1901.07714.

Li, W., Li, W., Sun, L., Wu, M., Yu, L., Liu, J., Li, Y., and Tian, S. Transformer-based model for symbolic regression via joint supervised learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ULzyv9M1j5.

Loughridge, C. R., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=yBgTVWccIx.

Mirzadeh, I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., and Farajtabar, M. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*, 2024.

Mugnier, E., Gonzalez, E. A., Polikarpova, N., Jhala, R., and Yuanyuan, Z. Laurel: Unblocking automated verification with large language models. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025. doi: 10.1145/3720499. URL https://doi.org/10.1145/3720499.

Mündler, N., He, J., Wang, H., Sen, K., Song, D., and Vechev, M. Type-constrained code generation with language models. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi: 10.1145/3729274. URL https://doi.org/10.1145/3729274.

Nagy, S., Zhou, T., Polikarpova, N., and D'Antoni, L. Chopchop: A programmable framework for semantically constraining the output of language models. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi: 10.1145/3776708. URL https://doi.org/10.1145/3776708.

Parisotto, E., rahman Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis, 2016. URL https://arxiv.org/abs/1611.01855.

Park, K., Wang, J., Berg-Kirkpatrick, T., Polikarpova, N., and D'Antoni, L. Grammar-aligned decoding. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=5G7ve8E1Lu.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Shojaee, P., Meidani, K., Gupta, S., Farimani, A. B., and Reddy, C. K. LLM-SR: Scientific equation discovery via programming with large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=m2nmp8P5in.

Solar-Lezama, A. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15 (5):475–495, 2013. ISSN 1433-2787. doi: 10.1007/s10009-012-0249-7. URL https://doi.org/10.1007/s10009-012-0249-7.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop verifiable code generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*, pp. 134–155, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-65111-3. doi: 10.1007/978-3-031-65112-0_7. URL https://doi.org/10.1007/978-3-031-65112-0_7.

Suresh, T., Banerjee, D., Ugare, S., Misailovic, S., and Singh, G. DINGO: Constrained inference for diffusion LLMs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=KaYMGsnZ4R.

Surís, D., Menon, S., and Vondrick, C. Vipergpt: Visual inference via python execution for reasoning. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 11854–11864, 2023. doi: 10.1109/ICCV51070.2023.01092.

Team, Q. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Tian, W., Zhang, S., Zhang, K., Chi, X., Fan, C., Lu, J., Luo, Y., Zhou, Q., Zhao, Y., Liu, N., Lin, S., Qin, Z., Ju, X., Zhang, S., and Tang, J. Seea-r1: Tree-structured reinforcement fine-tuning for self-evolving embodied agents, 2025. URL https://arxiv.org/abs/2506.21669.

Ugare, S., Gumaste, R., Suresh, T., Singh, G., and Misailovic, S. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=ac93gRzxxV.

Ugare, S., Suresh, T., Kang, H., Misailovic, S., and Singh, G. Syncode: LLM generation with grammar augmentation. *Transactions on Machine Learning Research*, 2025b. ISSN 2835-8856. URL https://openreview.net/forum?id=HiUZtgAPoH.

Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., and Chaudhuri, S. Houdini: lifelong learning as program synthesis. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 8701–8712, Red Hook, NY, USA, 2018. Curran Associates Inc.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Wang, J., Yan, Q., Wang, Y., Tian, Y., Mishra, S. S., Xu, Z., Gandhi, M., Xu, P., and Cheong, L. L. Reinforcement learning for self-improving agent with skill library, 2026. URL https://arxiv.org/abs/2512.17102.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better llm agents. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Zhong, Z., Raghunathan, A., and Carlini, N. Impossiblebench: Measuring llms' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*, 2025.

Zhou, W., Ou, Y., Ding, S., Li, L., Wu, J., Wang, T., Chen, J., Wang, S., Xu, X., Zhang, N., Chen, H., and Jiang, Y. E. Symbolic learning enables self-evolving agents, 2024. URL https://arxiv.org/abs/2406.18532.

Zhu, H., Xiong, Z., Magill, S., and Jagannathan, S. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 686–701, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314638. URL https://doi.org/10.1145/3314221.3314638.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=uTC9AFXIhg.

A. Technical Details: Search Space and Library Functions

We formalize the search space, library functions, and parametric models used by SEVerA.

A.1. Program Search Space

We use a restricted subset of the popular verification-aware language Dafny (Leino, 2010) 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 D. 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. SEVerA 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. This choice is motivated by several factors: (1) Dafny's built-in verification infrastructure directly supports the formal guarantees central to SEVerA, (2) its type system aligns with the basic types T used in our specification language, and (3) it supports annotations such as loop invariants and ranking functions that our planner LLM can synthesize to aid the verifier. While the framework is not language-specific, using a verification-aware host language avoids re-implementing a custom verifier and termination checker.

A.2. 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 $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n.\ {f_n}^{\Phi}(x_1, \ldots, x_n) \implies {f_n}^{\Psi}(x_1, \ldots, x_n, r)$. $\mathcal{A}$ contains all these formal contracts.

A.3. 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.

A.4. Inputs to SEVerA and Verifier Setup

The input $\mathcal{I} = (\mathcal{F}, \mathcal{A}, \mathsf{L}, \mathrm{D}, \Phi, \Psi, I)$ to SEVerA includes the library functions $\mathcal{F}$, axioms $\mathcal{A}$, point-wise 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.

B. Technical Details: FGGMs

B.1. Background: LLMs as Bounded String Generators

We treat LLMs as bounded probabilistic string generators. Fixed-length generation $\mathcal{L}_n : \Sigma^* \to \Sigma^{\leq n}$ iteratively composes two high-level steps, repeated at most n times. Each iteration samples a single character (or token) from Σ or the special end-of-sequence symbol $\langle eos \rangle \notin \Sigma$ that marks termination. 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 $\boldsymbol{x} \in \Sigma^*$ to a probability distribution over $\Sigma \cup \langle eos \rangle$. 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 $\boldsymbol{x}$ and the predicted distribution. If the decoding step produces $\langle eos \rangle$, generation terminates.

Definition B.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 an 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 $\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{y}_n$.

Definition B.2 (Support set)**.** 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 $S(\pi) = \{x \in \Omega \mid D_\pi(x) > 0\}$.

B.2. Examples of Formal Constraints

We provide complete encodings of the four representative tasks.

ⓐ **Scientific Discovery (Constrained Symbolic Regression).** The task is to recover an unknown mathematical formula f_{gt} from observed data. Formal constraints (Φ, Ψ) encode prior knowledge such as known symbolic bounds (Błądek & Krawiec, 2019; Haider & Kronberger, 2022; Li et al., 2023) or asymptotic behavior (Li et al., 2019). Any generated formula that violates the hard constraints is invalid, regardless of empirical fit. Under noise, constraints help prevent overfitting beyond what soft objectives can address (Kronberger et al., 2022).

ⓑ **Program Verification.** Given a program in a verification-aware language such as Dafny with a predefined specification, the agent synthesizes annotations (loop invariants, assertions, ranking functions) (Loughridge et al., 2025; Sun et al., 2024; Mugnier et al., 2025). Banerjee et al. (2026) show that agents using frontier models cheat by subtly modifying the input program (e.g., altering variable initializations) so the modified program with proposed annotations passes verification. 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 input and annotated output serve as hard constraints, ensuring the agent cannot cheat even on unseen inputs.

ⓒ **Constrained LLM Generation.** A class of problems requires LLMs to generate strings that follow a formal structure, defined by formal grammars (regular (Suresh et al., 2025) or context-free (Ugare et al., 2025b; Park et al., 2024)) or static analyzers such as type checkers (Mündler et al., 2025; Nagy et al., 2026). We consider generating symbolic math expressions from natural-language questions; hard constraints ensure the synthesized agent always produces a structurally valid expression.

ⓓ **Agentic Tool Use.** Conversational agents in customer-service settings must select and invoke API tools while respecting domain-specific policies (refund eligibility, booking-modification rules, authentication-before-access) (Barres et al., 2025). These temporal and logical constraints are expressed as formal specifications. Agent-C (Kamath et al., 2025) provides a DSL for temporal properties translated into LTL, enabling SMT-based checking of generated tool calls. The hard constraint is enforced at each individual tool call: given the trace of prior calls, the Agent-C checker verifies whether the next proposed call complies with the formal policy before execution. Task performance (utility), in contrast, is measured over the entire multi-turn trace as whether the agent successfully resolves the user request. In our formulation, Ψ uses the Agent-C checker to define a hard constraint per tool call, ensuring synthesized agents never violate domain policies regardless of input.

Overall, hard behavioral constraints capture the minimal requirements that synthesized agents must satisfy on all inputs, enabling pruning of untrusted candidates during synthesis.

B.3. FGGM Formal Definition

Eq. 5 specifies the formal syntax for defining an FGGM.

$$\langle FGGM\rangle ::= \langle id\rangle\ ";"\ \langle GMid\rangle ";"\ \overbrace{\langle typeSig\rangle\ ";"\ \langle \Phi_l\rangle ";"\ \langle \Psi_l\rangle\ ";"}^{\text{signature + formal contract}}$$
$$\underbrace{\langle f_p\rangle ";"}_{\text{prompt}}\ \underbrace{\langle f_d\rangle ";"}_{\text{fallback}}\ \underbrace{\langle f_{err}\rangle ";"}_{\text{error feedback}}\ \langle info\rangle \tag{5}$$

Where: (a) *id* is the unique identifier of the created FGGM; (b) *GMid* identifies the underlying GM used as proposal distribution; (c) *typeSig* defines the type signature, $(x_1 : T_1, \ldots, x_n : T_n) \to T_o$; (d) (Φ_l, Ψ_l) are first-order formulas whose terms can include non-parametric library functions from $\mathcal{F}_c$; (e) f_p constructs the GM input from the typed input variables and the task description; (f) f_d is a non-parametric fallback program satisfying (Φ_l, Ψ_l); (f) f_{err} is a non-parametric program explaining why a sample y is rejected;and (h) *info* is a textual description used by the planner during program generation. SEVerA parses these inputs, validates the definition, and synthesizes a rejection sampler (Fig. 2). For valid definitions, SEVerA synthesizes $check_{\mathcal{A},\Phi_l,\Psi_l}$, which checks whether a concrete tuple $(x_1, \ldots, x_n, y)$ satisfies (Φ_l, Ψ_l).

The BNF grammar for parsing FGGM definitions is in Appendix Q. Provided SEVerA validates the definition (including f_d), we have $\forall\theta \in \Theta.\forall x_1 \in T_1 \ldots \forall x_n \in T_n.\Phi_l(x_1, \ldots, x_n) \implies \Psi_l(x_1, \ldots, x_n, id(x_1, \ldots, x_n))$, ensuring that the formal contracts are preserved for all parameter values $\theta \in \Theta$.

B.4. FGGM Well-Formedness

Given an FGGM definition $\mathsf{G} = (id_{\mathsf{G}}, f_{\mathsf{G}}^{\Theta}, \tau_{\mathsf{G}}, \Phi_l, \Psi_l, f_p, f_d, info)$, well-formedness checks proceed as follows. Let the parametric GM identified by *GMid* be $\mathcal{L}_\Theta : T_{in} \to T_o$. SEVerA first checks the well-formedness of the type signature $\tau_{\mathsf{G}} = (T_1 \times \cdots \times T_n) \to T_o$ and of the local contracts (Φ_l, Ψ_l). We then verify that all terms in Φ_l and subsequently Ψ_l involving non-parametric functions from $\mathcal{F}_c$ are valid (details in Appendix R). The definition is considered valid provided both f_p and f_d are type-checked and terminating, and f_d satisfies $\mathcal{V}_{\mathsf{C}}[\Phi_l, \Psi_l](f_d)$.

After validity, SEVerA constructs $check_{\mathcal{A},\Phi_l,\Psi_l} : (T_1 \times \cdots \times T_n \times T_o) \to \{T, F\}$, which checks whether a concrete input–output tuple $(x_1, \ldots, x_n, y)$ satisfies Ψ_l (see Appendix B.5). The pseudocode of $check_{\mathcal{A},\Phi_l,\Psi_l}$ and a soundness proof showing $\varphi_{\mathcal{A}} \implies \big(\forall x_1, \ldots, x_n, y.\, \Phi_l(x_1, \ldots, x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \implies \Psi_l(x_1, \ldots, x_n, y))\big)$ are in Appendix W. For quantifier-free Ψ_l, the checker is complete (Lemma B.3).

Lemma B.3 (Completeness of Checker)**.** *For any valid FGGM* G *with quantifier-free* Ψ_l, $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n, \forall y \in T_o.\, \Phi_l(x_1, \ldots, x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi_l(x_1, \ldots, x_n, y))$.

Proof Sketch. By structural induction on the quantifier-free formula Ψ_l. *Base case:* an atomic predicate over computable library functions evaluates the same boolean value as Ψ_l under any concrete substitution. *Inductive step:* compound formulas $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$ are handled by $check_{\mathcal{A},\Phi_l,\Psi_l}$ recursively. Formal details in Appendix W. □

For any valid FGGM, the rejection sampler $id_{\mathsf{G}} : T_1 \times \cdots \times T_n \to T_o$ with fallback (Fig. 2) satisfies the contract (Φ_l, Ψ_l) for all well-typed inputs. Crucially, contract preservation holds over all parameters $\theta \in \Theta$. The function terminates on all inputs since K is finite and f_p, f_d, $check_{\mathcal{A},\Phi_l,\Psi_l}$ are all terminating.

Proof Sketch. [Theorem 3.1] Let $r = id_{\mathsf{G}}(x_1, \ldots, x_n)$. Based on the rejection sampler with checker $check_{\mathcal{A},\Phi_l,\Psi_l}$, the following holds (Eq. 6). Eq. 7 follows from validity of f_d, and Eq. 8 follows from soundness of $check_{\mathcal{A},\Phi_l,\Psi_l}$. We use $(\forall x_i \in T_i)$ to denote $(\forall x_1 \in T_1, \ldots, \forall x_n \in T_n)$ and y_f is the final rejected sample if f_d is used.

$$\begin{aligned}
&\varphi_{\mathcal{A}} \implies \forall\theta.(\forall x_i \in T_i).\Phi_l(x_1, \ldots, x_n) \\
&\quad \implies check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, r) \vee (r = f_d(x_1, \ldots, x_n, y_f)) && (6)\\
&\varphi_{\mathcal{A}} \implies \forall\theta.(\forall x_i \in T_i).\Phi_l(x_1, \ldots, x_n) \\
&\quad \wedge (r = f_d(x_1, \ldots, x_n, y_f)) \implies \Psi_l(x_1, \ldots, x_n, r) && (7)\\
&\varphi_{\mathcal{A}} \implies \forall\theta.(\forall x_i \in T_i).(\Phi_l(x_1, \ldots, x_n) \\
&\quad \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, r)) \implies \Psi_l(x_1, \ldots, x_n, r) && (8)
\end{aligned}$$

Combining yields $\varphi_{\mathcal{A}} \implies \forall\theta.(\forall x_i \in T_i).\Phi_l \implies \Psi_l$. Formal proof in Appendix W. □

B.5. Computability of $check_{\mathcal{A},\Phi_l,\Psi_l}$ on Concrete Values

The auto-synthesized rejection sampler requires checking whether a sample y from $\mathcal{L}_\Theta$, given a concrete input $(x_1, \ldots, x_n)$, satisfies the planner-defined Ψ_l. For any *quantifier-free* Ψ_l, this can be computed efficiently for all concrete values: since all library functions in $\mathcal{F}_c$ are computable, $check_{\mathcal{A},\Phi_l,\Psi_l}$ can evaluate every term on $(x_1, \ldots, x_n, y)$. Satisfiability is then decided in linear time in the number of terms in Ψ_l (Lemma B.3).

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 concrete values, encodes the axioms $\mathcal{A}$ and Φ_l similarly to Eq. 9, 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, so 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, false. This guarantees soundness; however, in the presence of quantifiers, $check_{\mathcal{A},\Phi_l,\Psi_l}$ may be incomplete and incorrectly reject valid samples.

B.6. Conformance Tuning

A FGGM that always uses the fallback is formally correct but practically no better than the non-parametric program f_d. Conformance tuning teaches the GM to satisfy the contract directly. The local contracts (Φ_l, Ψ_l) define a learning objective that assigns higher probability to outputs passing $check_{\mathcal{A},\Phi_l,\Psi_l}$. The objective in Eq. 11 minimizes the expected contract violation of GM samples over an input set $\mathbb{P}$. By optimizing $\theta \in \Theta$ to reduce this loss, the GM concentrates probability mass on the support defined by (Φ_l, Ψ_l). Overall, conformance tuning improves the rejection sampler's acceptance rate, reducing reliance on f_d.

B.7. Axiom Implication Schema

For any candidate program f to satisfy the behavioral specification, we require:

$$\varphi_{\mathcal{A}} \implies \forall x \in T_i.\, (\Phi(x) \implies \Psi(x, f(x))) \tag{9}$$

where $\varphi_{\mathcal{A}} = \bigwedge_{\varphi \in \mathcal{A}} \varphi$.

C. Technical Details: Algorithms and Learning

C.1. 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 $\mathcal{G} = \{\mathsf{G}_1, \ldots, \mathsf{G}_m\}$ following Eq. 5. Each FGGM wraps a parametric model with local contracts and a verified fallback. Using these FGGMs as callable functions, the planner samples a candidate parametric program $\mathsf{P}_{\{\Theta\}}$, where $\{\Theta\} = \{\Theta_1, \ldots, \Theta_k\}$ represents all optimizable parameters. FGGM calls at different program locations do not share parameters and add their own parameter to $\{\Theta\}$.

In the verification phase, SEVerA first checks the well-formedness of every $\mathsf{G}_i \in \mathcal{G}$. If any definition fails to verify, an error message is returned. If all definitions are valid, the verification context $\mathsf{C} = (G_D, \mathcal{F}, \mathcal{A})$ is extended with the synthesized FGGMs and their local contracts: $\mathsf{C}' = (G_D, \mathcal{F} \cup \mathcal{G}, \mathcal{A} \cup \mathcal{A}_{\mathcal{G}})$, where $\mathcal{A}_{\mathcal{G}}$ contains the local contracts $(\mathsf{G}_i^{\Phi_l}, \mathsf{G}_i^{\Psi_l})$ of each FGGM. The candidate program $\mathsf{P}_{\{\Theta\}}$ is then checked for syntactic validity, termination, and compliance with (Φ, Ψ). If verification succeeds, SEVerA accepts $\mathsf{P}_{\{\Theta\}}$, ensuring it satisfies (Φ, Ψ) over all parameters (Theorem C.1).

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 $\mathsf{P}_{\{\Theta\}}$ else $\perp$
$i \leftarrow 0$, $\mathsf{err} \leftarrow \{\}$
while $i < \delta$ **do**
  $\mathcal{G} \leftarrow defineFGGM(\mathcal{L}_p, I, \mathcal{F}, \mathcal{A}, I', \mathsf{err})$
  $\mathsf{err} \leftarrow validateFGGM(\mathcal{G}, \mathcal{F}_c, \mathcal{A})$
  if $\mathsf{err} \neq \{\}$ **then**
    $i \leftarrow i + 1$; **continue**
  end if
  $\mathsf{P}_{\{\Theta\}} \leftarrow sampleProgram(\mathcal{L}_p, I, \mathcal{F}, \mathcal{A}, I', \mathcal{G}, \mathsf{err})$
  $\mathsf{C} \leftarrow (G_D, \mathcal{F} \cup \mathcal{G}, \mathcal{A} \cup \mathcal{A}_{\mathcal{G}})$
  $\mathsf{err} \leftarrow validateProgram(\mathsf{P}_{\{\Theta\}}, \mathcal{P}_{\mathsf{C}}, \mathcal{T}_{\mathsf{C}}, \mathcal{V}_{\mathsf{C}}[\Phi, \Psi])$
  if $\mathsf{err} = \{\}$ **then**
    return $\mathsf{P}_{\{\Theta\}}$
  end if
  $i \leftarrow i + 1$
end while
return $\perp$

Algorithm 1 summarizes the loop. Given $\mathcal{I}$ and planner $\mathcal{L}_p$, it iteratively proposes candidate solutions within budget δ (Lines 1–4). In each iteration, the planner synthesizes FGGM definitions $\mathcal{G}$ (Line 6), checks them via $validateFGGM$ (Line 7), then samples $\mathsf{P}_{\{\Theta\}}$ (Line 11) and extends the verification context (Line 12). The candidate is validated for syntax, termination, and compliance via $validateProgram$ (Line 13). If verification succeeds, the program is returned; otherwise, the loop continues with error feedback until the budget is exhausted.

Theorem C.1. *If $\mathsf{P}_{\{\Theta\}} \neq \perp$ 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 3.1 with full details in Appendix W. □

C.2. Learn

Once a parametric program $\mathsf{P}_{\{\Theta\}}$ passes the search–verification loop (Algorithm 1), Theorem C.1 guarantees that $\mathsf{P}_{\{\Theta\}}$ satisfies (Φ, Ψ) for *all* parameter values. Hence the learning step can freely select parameters without violating (Φ, Ψ). Finding the optimal solution to Eq. 10 over $\{\Theta\}$ for a general L is intractable; instead, SEVerA employs scalable gradient-descent–based optimization.

$$({\theta_1}^*, \ldots, {\theta_k}^*) = \underset{(\theta_1,\ldots,\theta_k)\in\{\Theta\}}{\arg\min} \sum_{(x,y)\in \mathrm{D}} \mathsf{L}(x, y, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x)) \tag{10}$$

Conformance Loss. The 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, SEVerA increases the rejection sampler's success rate and avoids the static fallback. Since f_d cannot be tuned, reducing reliance on it allows gradient-based search to more effectively guide the parameters toward solutions that reduce L. Our experiments (§ 4.3) confirm this. For a single FGGM with (Φ_l, Ψ_l) and parameter θ, the conformance loss over an input set $\mathbb{P}$ is:

$$\begin{aligned} \mathsf{L}_{\Phi_l,\Psi_l}(\theta) &= \tfrac{1}{|\mathbb{P}|}\textstyle\sum_{p\in\mathbb{P}} \mathsf{L}^{\theta}_{\Phi_l,\Psi_l}(p), \\ \mathsf{L}^{\theta}_{\Phi_l,\Psi_l}(p) &= \mathbb{E}_{y\sim\mathcal{L}_\theta(p)} 1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y)) \end{aligned} \tag{11}$$

The augmented objective combines the task loss with local conformance losses:

$$\textstyle\sum_{(x,y)\in \mathrm{D}} \mathsf{L}(x, y, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x)) + \lambda \sum_{i,(x,y)} \mathsf{L}^{i}_{\Phi_l,\Psi_l}(\theta_i) \tag{12}$$

where $\lambda \in \mathbb{R}^+$ is a user-provided constant.

Decomposed Optimization. Optimizing all parameters jointly is feasible for smaller NNs but expensive for agents with multiple interdependent LLM calls. We leverage a structural property: the conformance loss $\mathsf{L}^{i}_{\Phi_l,\Psi_l}(\theta_i)$ for each FGGM depends only on local inputs and the underlying GM's outputs, not on the parameters of other FGGMs. This independence arises because each FGGM's rejection sampler and $check_{\mathcal{A},\Phi_l,\Psi_l}$ checker operate solely on the local contract (Φ^i_l, Ψ^i_l) and the corresponding GM's outputs. The conformance term decomposes naturally across FGGM calls. For larger models, we approximate the augmented loss by optimizing each FGGM independently:

$$\begin{aligned} \mathsf{L}_a(\theta_i) &= \tfrac{1}{|\mathbb{P}|}\textstyle\sum_{p\in\mathbb{P}} \mathsf{L}^{\theta_i}_a(p), \\ \mathsf{L}^{\theta_i}_a(p) &= \mathbb{E}_{y_l\sim id_{\theta_i}(p)} \mathsf{L}(x, _, y)\mathbb{I}(y = y_l) + \lambda \mathsf{L}^{\theta_i}_{\Phi_l,\Psi_l}(p) \end{aligned} \tag{13}$$

$$\begin{aligned} \mathcal{R}(p, y_l) = 1 - Sigmoid\big(&\mathsf{L}(x, _, y)\mathbb{I}(y = y_l) \\ &+ \lambda(1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l)))\big) \end{aligned} \tag{14}$$

The first term in $\mathsf{L}^{\theta_i}_a$ includes the task loss only when the FGGM's output y_l matches the agent's final output y. The decomposition enables efficient parallelized training across θ_i.

For generative models with accessible parameters, the objective in Eq. 13 can be optimized using RL-based fine-tuning methods such as GRPO (Shao et al., 2024) (details in Appendix S). Eq. 14 defines the reward for (p, y_l), learning an output distribution over y_l for inputs $p \in \mathbb{P}$ that maximizes total reward $\mathcal{R}(p, y_l)$. The reward combines two signals: (1) the task-specific loss L, and (2) the conformance indicator $\mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l))$. The sigmoid maps the combined penalty to $[0, 1]$, providing a smooth reward landscape. Outputs that violate the local contract receive near-zero reward due to $(1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l)))$, strongly discouraging rejected outputs. The reward need not be differentiable w.r.t. θ_i, as long as the outputs y_l remain differentiable (Shao et al., 2024). For closed-source models, the learning step is skipped and improvements rely on prompt tuning through f_p.

C.3. The SEVerA Algorithm

Algorithm 2 summarizes the overall workflow. SEVerA maintains a pool $\mathcal{P}$ of verified candidate agents and iteratively improves it through a search–verify–learn loop. Each iteration first invokes $searchVerify$ (Algorithm 1) to synthesize FGGM definitions and a verified candidate $\mathsf{P}_{\{\Theta\}}$. If verification succeeds, the program is passed to LEARN, which optimizes the underlying GMs' parameters using D and L. For closed-source models, this is skipped; improvements rely on prompt tuning via f_p. The tuned agent is added to the pool, and its execution traces on D are used to construct feedback I' (Appendix U) guiding the planner in subsequent iterations. After the budget Δ is exhausted, SEVerA returns the agent in the pool with the lowest task loss, or $\perp$ if the pool is empty.

Algorithm 2 SEVerA

Input: $\mathcal{I} = (\mathcal{F}, \mathcal{A}, \mathsf{L}, \mathrm{D}, \Phi, \Psi, I)$, 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**
    $({\theta_1}^*, \ldots, {\theta_k}^*) \leftarrow learn(\mathsf{P}_{\{\Theta\}}, \mathsf{L}, \mathrm{D})$
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}_{\{({\theta_1}^*, \ldots, {\theta_k}^*)/\Theta\}}\}$
  else
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}_{\{\Theta\}}\}$
  end if
  $I' \leftarrow collectFeedback(\mathsf{P}_{\{\Theta\}}, \mathrm{D})$
end while
$f^* \leftarrow \arg\min_{f \in \mathcal{P}} \sum_{(x,y) \in \mathrm{D}} \mathsf{L}(x, y, f(x))$
return f^*

C.4. Theoretical Results: Detailed Statements

C.4.1. Soundness

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

Proof Sketch. [Theorem 3.2] $(f^* \neq \perp) \implies (f^* \in \mathcal{P})$ and every $f \in \mathcal{P}$ satisfies $\varphi_{\mathcal{A}} \implies \forall x_1, \ldots, x_n. \Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$ by Theorem C.1. Full details in Appendix W. □

C.4.2. Sufficient Success Condition

We characterize a sufficient condition that ensures the existence of a candidate $f \in \mathsf{S}(G, \mathcal{F})$ satisfying (Φ, Ψ) and achieving no worse loss than any GM $f_n^\theta \in \mathcal{F}_p$ with any f_p and initial parameters θ_0. The high-level idea: if there exists a non-parametric $f_d \in \mathsf{S}(G, \mathcal{F}_c)$ satisfying (Φ, Ψ), we use it as the fallback together with any type-correct f_n^θ to construct an FGGM. A program that invokes this FGGM and returns its output satisfies (Φ, Ψ). Moreover, if $check_{\mathcal{A},\Phi,\Psi}$ is complete (which holds for quantifier-free Ψ by Lemma B.3), the FGGM always returns valid samples from f_n^θ.

Proof Sketch. [Theorem 3.3] Define $\mathsf{G} = (id, f_n^\theta, \tau, \Phi_l, \Psi_l, f_p, f_d, info)$ with $(\Phi_l, \Psi_l) := (\Phi, \Psi)$. Since f_d satisfies (Φ, Ψ), the FGGM is valid. Define f:

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

For inputs satisfying Φ, by completeness of $check_{\mathcal{A},\Phi_l,\Psi_l}$ (Lemma B.3), id returns $f_n^\theta(x)$ whenever $\Psi(x, f_n^\theta(x))$ holds, else $f_d(x)$ which satisfies Ψ. By assumption (i), substituting fallback outputs cannot increase loss, so $\mathsf{L}(f) \leq \mathsf{L}(f_n^\theta)$. If f_n^θ violates Ψ on some $(x, _) \in \mathrm{D}$, the improvement is strict. Full details in Appendix W. □

Theorem 3.3 provides a constructive argument: the FGGM mechanism can always improve upon a bare GM call by filtering out constraint-violating outputs and substituting the fallback. The three assumptions are mild in practice: (i) requires only that constraint-satisfying outputs are preferred over violating ones, which holds for any reasonable loss function in constrained settings; (ii) ensures completeness of the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ (Lemma B.3) so that no valid GM outputs are incorrectly rejected. In our experiments, the symbolic regression task uses quantifier-free specifications, while the program verification task uses library-function predicates (e.g., $parse$, $noDiff$) that are computable on concrete inputs, effectively behaving as quantifier-free checks at runtime; (iii) the existence of a valid fallback is a natural requirement: if no non-parametric program can satisfy the specification, then the specification may be too strong for the given library. The theorem does not guarantee that SEVerA will *find* the optimal program, as the planner LLM may not propose the right candidate within the search budget. Rather, it establishes that the search space $\mathsf{S}(G, \mathcal{F})$ is rich enough to contain solutions that dominate unconstrained GM calls.

D. 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\ \texttt{(}\langle params\rangle\texttt{)}\ \langle specs\rangle\ \texttt{\{}\langle stmt_list\rangle\texttt{\}} \\
\langle function_decl\rangle & ::= & \texttt{function}\ \langle ident\rangle\ \texttt{(}\langle params\rangle\texttt{)}\ \texttt{:}\ \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\ \texttt{(}\langle args\rangle\texttt{)}\ \texttt{;} \\
\langle args\rangle & ::= & \epsilon \mid \langle expr\rangle \mid \langle expr\rangle, \langle args\rangle \\
\langle if_stmt\rangle & ::= & \texttt{if}\ \texttt{(}\langle formula\rangle\texttt{)}\ \texttt{\{}\langle stmt_list\rangle\texttt{\}} \\
 & & \mid \texttt{if}\ \texttt{(}\langle formula\rangle\texttt{)}\ \texttt{\{}\langle stmt_list\rangle\texttt{\}}\ \texttt{else}\ \texttt{\{}\langle stmt_list\rangle\texttt{\}} \\
\langle while_stmt\rangle & ::= & \texttt{while}\ \texttt{(}\langle formula\rangle\texttt{)}\ \langle loop_annots\rangle\ \texttt{\{}\langle stmt_list\rangle\texttt{\}} \\
\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 \texttt{1} \mid \cdots \mid \texttt{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}
$$

E. 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 $M = 1$.

F. Library Functions for Symbolic Regression

Library Function Set $\mathcal{F}$

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

G. Additional Axioms for Symbolic Regression

G.1. Axiom Syntax

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

Axiom Set $\mathcal{A}$

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

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

H. Library Functions for Dafny Program Verification

Library Function Set $\mathcal{F}$

- $claude(prompt : str) \rightarrow r : str$
- $noDiff(base_code : str, annotated_code : str) \rightarrow r : bool$
- $noDiffWithErrorMsg(base_code : str, annotated_code : str) \rightarrow r : str$
- $dafnyVerifier(dafny_code : str) \rightarrow r : bool$
- $dafnyVerifierWithErrorMsg(dafny_code : str) \rightarrow r : str$
- $extractDafnyCode(text : str) \rightarrow r : str$
- $isSubstring(sub : str, full : str) \rightarrow r : bool$

I. Additional Axioms for Dafny Program Verification

Axiom Set $\mathcal{A}$

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

J. Library Functions for τ^2-bench

Library Function Set $\mathcal{F}$

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

K. 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” $\implies agentCCheck(r, t, d)$.

L. Library Functions for GSM-Symbolic

Library Function Set $\mathcal{F}$

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

M. Additional Axioms for GSM-Symbolic

Axiom Set $\mathcal{A}$

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

N. 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. 3), $diffErrorFGGM$ (Fig. 4), and $verifierErrorFGGM$ (Fig. 5)—each with local contract $\Psi_l : noDiff(base_program, \cdot)$ 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 and error function f_{err}: 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. Each FGGM includes a stateful f_{err} that explains why a sample was rejected, enabling the rejection sampler (Fig. 2) to accumulate error context across iterations. The agent program (Fig. 6) dispatches to the appropriate FGGM on each loop iteration. On HumanEvalDafny, the fallback fires on only 5.1% of instances (Table 7), confirming that the rejection sampler with f_{err} feedback rarely exhausts all K samples.

```
FGGM Definition:
  id := initialFGGM
  GMid := claude
  typeSig := (base_program : str) → str
  Φ_l := true
  Ψ_l := noDiff(base_program, f(base_program))
  f_p := 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("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("<", base_program) ∧ isSubstring(">", 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("set<", base_program) ∨ isSubstring("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("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_err := function f_err(base_program : str, y : str) : (str) {
      var diff : str := noDiffErrorMsg(base_program, y);
      return "Output modified the base program. Diff: " + diff
        + "\nOnly add annotations without changing signatures or logic."; }
  f_d := function f_d(base_program : str, y : str) : (str)
      ensures noDiff(base_program, f_d(base_program, y))
  {
      return base_program;
  }
```

Figure 3. FGGM definition for $initialFGGM$. The prompting function builds context-specific guidance by checking for lemmas, generics, sets, and multisets in the base program. The f_{err} function explains why a sample was rejected (diff violation), making the rejection sampler stateful. The fallback returns the original program, satisfying Ψ_l by the reflexivity axiom. Fallback trigger rate: 5.1% on HumanEvalDafny (Table 7).

```
FGGM Definition:
  id := diffErrorFGGM
  GMid := claude
  typeSig := (base_program : str, diff_error : str, prev_attempt : str) → str
  Φ_l := true
  Ψ_l := noDiff(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"
       + "noDiff 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_err := function f_err(base_program : str, diff_error : str, prev_attempt : str, y : str) : (str) {
      var diff : str := noDiffErrorMsg(base_program, y);
      return "Rejection: output still modifies base program. Diff: " + diff
        + "\nPrevious diff error was: " + diff_error; }
  f_d := function f_d(base_program : str, diff_error : str, prev_attempt : str, y : str) : (str)
      ensures noDiff(base_program, f_d(base_program, diff_error, prev_attempt, y))
  {
      return base_program;
  }
```

Figure 4. 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. The f_{err} function provides diagnostic feedback on why the new sample was also rejected, preventing repeated mistakes. Fallback trigger rate: 5.1% on HumanEvalDafny (Table 7).

```
FGGM Definition:
  id := verifierErrorFGGM
  GMid := claude
  typeSig := (base_program : str, verify_error : str, prev_attempt : str) → str
  Φ_l := true
  Ψ_l := noDiff(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("timeout", verify_error) ∨ isSubstring("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("lemma", verify_error) ∨ isSubstring("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("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("type", verify_error) ∨ isSubstring("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_err := function f_err(base_program : str, verify_error : str, prev_attempt : str, y : str) : (str) {
      var diff : str := noDiffErrorMsg(base_program, y);
      return "Rejection: output modifies base program. Diff: " + diff
         + "\nThe task is to fix verification errors, not modify program structure."; }

  f_d := function f_d(base_program : str, verify_error : str, prev_attempt : str, y : str) : (str)
      ensures noDiff(base_program, f_d(base_program, verify_error, prev_attempt, y))
  {
      return base_program;
  }
```

Figure 5. 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). The f_{err} function explains why each rejected sample violates the contract, enabling the stateful rejection sampler to avoid repeated failures. Fallback trigger rate: 5.1% on HumanEvalDafny (Table 7).

```
method agent(base_program : str) returns (r : str)
  ensures noDiff(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 noDiff(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 := noDiffWithErrorMsg(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;
}
```

Figure 6. Verified agent program for Dafny annotation synthesis. On the first iteration, $initialFGGM$ (Fig. 3) generates an initial annotation. On subsequent iterations, the agent dispatches to $diffErrorFGGM$ (Fig. 4) or $verifierErrorFGGM$ (Fig. 5) depending on the previous error type. Each FGGM guarantees $noDiff(base_program, y)$ via a rejection sampler with $K = 5$ and stateful f_{err} feedback, maintaining the loop invariant. The fallback fires on only 5.1% of instances (HumanEvalDafny) and 7.9% (DafnyBench), confirming that the f_{err}-guided sampler produces conforming outputs in most cases. Upon exhausting all attempts, the agent returns $best$, which satisfies the postcondition by the loop invariant.

O. Example Agent for GSM-Symbolic

The following is an example verified agent program for the GSM-Symbolic math synthesis task. The agent defines and invokes four FGGMs—$initialFGGM$ (Fig. 7), $parseErrorFGGM1$ (Fig. 8), $parseErrorFGGM2$ (Fig. 9), and $parseErrorFGGM3$ (Fig. 10)—each with local contract $\Psi_l : gsmParser(\cdot)$ enforcing syntactic validity of the output expression and a fallback that returns "0" (a trivially parsable expression). All four share $GMid := qwen$ and differ in the prompting function f_p and error function f_{err}: the first generates an initial math expression, while the remaining three provide increasingly focused repair strategies for parse errors. Each FGGM includes a stateful f_{err} that explains why a sample was rejected (parse failure details), enabling the rejection sampler (Fig. 2) to avoid repeating the same syntactic mistakes. The agent program (Fig. 11) first attempts generation via $initialFGGM$, then dispatches to the appropriate parse-error FGGM on subsequent iterations. On GSM-Symbolic, the fallback fires on only 2.1% of instances (Table 7), confirming that the f_{err}-guided sampler produces syntactically valid expressions in nearly all cases.

```
FGGM Definition:
  id := initialFGGM
  GMid := qwen
  typeSig := (question : str, variable_types : str) → str
  Φ_l := true
  Ψ_l := gsmParser(f(question, variable_types))
  f_p := function f_p(question : str, variable_types : str) : (str) {
    var p : str :=
      "You are a math expert solving grade-school word problems.\n\n"
       + "CRITICAL RULES:\n"
       + "1. Use ONLY the variable names provided in 'Variable types'\n"
       + "2. ONLY use operators: +, -, *, /, //, %, **, int(), parentheses, numbers\n"
       + "3. NO functions like round(), abs(), pow(), etc.\n"
       + "4. Use // for integer division, / for regular division\n"
       + "5. Return ONLY the mathematical expression\n\n";

    p := p + "Question: " + question + "\n\n"
       + "Variable types: " + variable_types + "\n\n"
       + "Provide your answer as a single expression wrapped in << >>.";
    return p; }

  f_err := function f_err(question : str, variable_types : str, y : str) : (str) {
      var pe : str := gsmParserErrorMsg(y);
      return "Expression failed to parse. Error: " + pe
         + "\nAllowed: +, -, *, /, //, %, **, int(), vars from: " + variable_types; }

  f_d := function f_d(question : str, variable_types : str, y : str) : (str)
      ensures gsmParser(f_d(question, variable_types, y))
  {
      return "0";
  }
```

Figure 7. FGGM definition for $initialFGGM$ (GSM-Symbolic). The prompting function provides rules and examples for generating valid math expressions. The f_{err} function reports the parse error, enabling the rejection sampler to avoid repeating syntactic mistakes. The fallback returns "0", a trivially valid expression. Fallback trigger rate: 2.1% (Table 7).

```
FGGM Definition:
  id := parseErrorFGGM1
  GMid := qwen
  typeSig := (question : str, variable_types : str, parse_error : str, prev_expr : str) → str
  Φ_l := true
  Ψ_l := gsmParser(f(question, variable_types, parse_error, prev_expr))
  f_p := function f_p(question : str, variable_types : str,
          parse_error : str, prev_expr : str) : (str) {
    var p : str :=
      "Fix this expression. Use ONLY variable names from the list.\n\n"
       + "Failed expression: " + prev_expr + "\n"
       + "Error: " + parse_error + "\n\n"
       + "Variable types: " + variable_types + "\n"
       + "Question: " + question + "\n\n"
       + "Rules:\n"
       + "- Use exact variable names from Variable types (case-sensitive)\n"
       + "- Only operators: +, -, *, /, //, %, **, int(), parentheses\n"
       + "- NO round(), abs(), pow() or other functions\n\n"
       + "Return corrected expression in << >>.";
    return p; }
  f_err := function f_err(question : str, variable_types : str,
          parse_error : str, prev_expr : str, y : str) : (str) {
      var pe : str := gsmParserErrorMsg(y);
      return "Still invalid. Previous error: " + parse_error
         + "\nNew error: " + pe + "\nFailed expr: " + prev_expr; }
  f_d := function f_d(question : str, variable_types : str,
          parse_error : str, prev_expr : str, y : str) : (str)
      ensures gsmParser(f_d(question, variable_types, parse_error, prev_expr, y))
  {
      return "0";
  }
```

Figure 8. FGGM definition for $parseErrorFGGM1$ (GSM-Symbolic). Invoked on the first parse-error recovery attempt. The prompt provides the failed expression and error details. The f_{err} function accumulates both previous and new parse errors.

```
FGGM Definition:
  id := parseErrorFGGM2
  GMid := qwen
  typeSig := (question : str, variable_types : str, parse_error : str, prev_expr : str) → str
  Φ_l := true
  Ψ_l := gsmParser(f(question, variable_types, parse_error, prev_expr))
  f_p := function f_p(question : str, variable_types : str,
        parse_error : str, prev_expr : str) : (str) {
    var p : str :=
      "The expression has errors. Fix it carefully.\n\n"
       + "Failed: " + prev_expr + "\n"
       + "Error: " + parse_error + "\n"
       + "Variables: " + variable_types + "\n"
       + "Question: " + question + "\n\n"
       + "Check:\n"
       + "1. Are all variable names exactly as listed?\n"
       + "2. Are you using only: +, -, *, /, //, %, **, int(), ()?\n"
       + "3. No round(), abs(), or other functions?\n\n"
       + "Corrected expression in << >>:";
    return p; }

  f_err := function f_err(question : str, variable_types : str,
        parse_error : str, prev_expr : str, y : str) : (str) {
      var pe : str := gsmParserErrorMsg(y);
      return "Rejection: expression still invalid. Error: " + pe
         + "\nCheck variable names and operators carefully."; }

  f_d := function f_d(question : str, variable_types : str,
        parse_error : str, prev_expr : str, y : str) : (str)
      ensures gsmParser(f_d(question, variable_types, parse_error, prev_expr, y))
  {
      return "0";
  }
```

Figure 9. FGGM definition for $parseErrorFGGM2$ (GSM-Symbolic). Invoked on the second parse-error recovery attempt with a checklist-style prompt. The f_{err} function reports rejection with specific parse diagnostics.

```
FGGM Definition:
  id := parseErrorFGGM3
  GMid := qwen
  typeSig := (question : str, variable_types : str, parse_error : str, prev_expr : str) → str
  Φ_l := true
  Ψ_l := gsmParser(f(question, variable_types, parse_error, prev_expr))
  f_p := function f_p(question : str, variable_types : str,
        parse_error : str, prev_expr : str) : (str) {
    return "Fix to use only: +, -, *, /, //, %, **, int(), parentheses "
       + "and variables from: " + variable_types + "\n\n"
       + "Expression: " + prev_expr + "\n"
       + "Error: " + parse_error + "\n\n"
       + "Corrected in << >>:"; }

  f_err := function f_err(question : str, variable_types : str,
        parse_error : str, prev_expr : str, y : str) : (str) {
      var pe : str := gsmParserErrorMsg(y);
      return "Final attempt rejected. Error: " + pe; }

  f_d := function f_d(question : str, variable_types : str,
        parse_error : str, prev_expr : str, y : str) : (str)
      ensures gsmParser(f_d(question, variable_types, parse_error, prev_expr, y))
  {
      return "0";
  }
```

Figure 10. FGGM definition for $parseErrorFGGM3$ (GSM-Symbolic). A minimal last-resort prompt used on the third and subsequent parse-error attempts.

```
method agent(question : str, variable_types : str) returns (r : str)
  ensures gsmParser(r)
{
  var max_attempts : int := 7;
  var attempt : int := 0;
  var result : str := "0";

  // — initial generation —
  var response : str := initialFGGM(question, variable_types);
  var extracted : str := extractExpression(response);

  if (extracted ≠ "") { result := extracted; }
  else { result := response; }

  // — repair loop —
  while (attempt < max_attempts)
    invariant gsmParser("0")
    decreases max_attempts − attempt
  {
    if (result = "") { result := "0"; break; }

    var parse_error : str := gsmParserWithErrorMsg(result);

    if (parse_error = "") {
      r := result; return;
    }

    // — FGGM dispatch based on attempt —
    attempt := attempt + 1;
    var fix_response : str;

    if (attempt = 1) {
      fix_response := parseErrorFGGM1(question, variable_types, parse_error, result);
    } else if (attempt = 2) {
      fix_response := parseErrorFGGM2(question, variable_types, parse_error, result);
    } else {
      fix_response := parseErrorFGGM3(question, variable_types, parse_error, result);
    }

    var new_result : str := extractExpression(fix_response);
    if (new_result = "") { new_result := fix_response; }

    if (new_result ≠ "" ∧ new_result ≠ result) {
      result := new_result;
    } else {
      if (attempt ≥ max_attempts − 1) { break; }
    }
  }

  // — final validation —
  if (result ≠ "" ∧ gsmParser(result)) {
    r := result; return;
  }
  r := "0"; return;
}
```

Figure 11. Verified agent program for GSM-Symbolic. The agent first generates an expression via $initialFGGM$, then enters a repair loop dispatching to $parseErrorFGGM1$/$parseErrorFGGM2$/$parseErrorFGGM3$ based on the attempt number. Each FGGM guarantees $gsmParser(\cdot)$ via a rejection sampler with $K = 5$ and stateful f_{err} feedback. The fallback fires on only 2.1% of instances (Table 7).

P. Example Agent for τ^2-bench

The following is an example verified agent program for the τ^2-bench multi-turn tool-use task. The agent defines and invokes three FGGMs—$initialFGGM$ (Fig. 12), $duplicateFGGM$ (Fig. 13), and $agentCErrorFGGM$ (Fig. 14)—each with local contract $\Psi_l : (\neg isToolCall(\cdot)) \vee agentCCheck(parseToolCall(\cdot),\ trace)$, ensuring that any tool call satisfies the Agent-C temporal policy checker. The fallback transfers to a human agent, which trivially satisfies the policy. All three share $GMid := qwen$ and differ in prompting function f_p and error function f_{err}: the first generates an initial response, the second handles duplicate tool-call detection, and the third recovers from Agent-C constraint violations with targeted guidance. Each FGGM includes a stateful f_{err} that explains why a sample was rejected (policy violation details), enabling the rejection sampler (Fig. 2) to avoid repeating the same policy-violating tool calls. The agent program (Fig. 15) dispatches to the appropriate FGGM based on duplicate detection and constraint-check results. On τ^2-bench, the fallback fires on 15.6% (retail) and 21.6% (airline) of instances (Table 7).

```
FGGM Definition:
  id := initialFGGM
  GMid := qwen
  typeSig := (conversation : str, policy : str, tools : str) → str
  Φ_l := true
  Ψ_l := (¬isToolCall(f(...))) ∨ agentCCheck(parseToolCall(f(...)), trace)
  f_p := function f_p(conversation : str, policy : str, tools : str) : (str) {
    var sys : str := policy
      + "\n\nCRITICAL INSTRUCTIONS:\n"
      + "1. When user confirms an action, EXECUTE it immediately with the correct tool call\n"
      + "2. For reservation updates: call update_reservation_flights with NEW flights\n"
      + "3. For basic economy changes: cancel_reservation then book_reservation\n"
      + "4. Always communicate specific numbers: prices, refund amounts, IDs\n"
      + "5. When no direct flights exist, try search_onestop_flight\n"
      + "6. Only transfer to human after genuinely attempting required actions\n";
    return "[system: " + sys + "] " + conversation; }

  f_err := function f_err(conversation : str, policy : str, tools : str, y : str) : (str) {
      var tc : str := parseToolCall(y);
      var err : str := agentCCheckErrorMsg(tc, trace);
      return "Tool call rejected by policy checker. Error: " + err
        + "\nEither fix the tool call or respond with text instead."; }

  f_d := function f_d(conversation : str, policy : str, tools : str, y : str) : (str)
      ensures (¬isToolCall(f_d(...))) ∨ agentCCheck(parseToolCall(f_d(...)), trace)
  {
      return transferToHuman("Unable to process request");
  }
```

Figure 12. FGGM definition for $initialFGGM$ (τ^2-bench). The prompting function provides domain-specific instructions. The f_{err} function reports the Agent-C policy violation, enabling the rejection sampler to avoid repeating non-compliant tool calls. The fallback transfers to a human agent. Fallback trigger rate: 15.6% retail, 21.6% airline (Table 7).

```
FGGM Definition:
  id := duplicateFGGM
  GMid := qwen
  typeSig := (messages : str, prev_response : str, tool_name : str, tools : str) → str
  Φ_l := true
  Ψ_l := (¬isToolCall(f(...))) ∨ agentCCheck(parseToolCall(f(...)), trace)
  f_p := function f_p(messages : str, prev_response : str,
          tool_name : str, tools : str) : (str) {
    var feedback : str :=
      "You are repeating the same tool call. Try a DIFFERENT approach:\n"
       + "1. If you have user confirmation, EXECUTE the action tool NOW.\n"
       + "2. If missing prerequisites: call get_user_details or get_reservation_details ONCE.\n"
       + "3. If no direct flights found, try search_onestop_flight.\n"
       + "4. For basic economy: cancel_reservation + book_reservation.\n"
       + "5. Stop gathering information and START EXECUTING the confirmed action.";
    return messages + " + [assistant: " + prev_response
      + ", user: " + feedback + "]"; }
  f_err := function f_err(messages : str, prev_response : str,
          tool_name : str, tools : str, y : str) : (str) {
      var tc : str := parseToolCall(y);
      var err : str := agentCCheckErrorMsg(tc, trace);
      return "Rejected: policy violation on tool " + tool_name
        + "\nError: " + err + "\nTry a different tool or respond with text."; }
  f_d := function f_d(messages : str, prev_response : str,
          tool_name : str, tools : str, y : str) : (str)
      ensures (¬isToolCall(f_d(...))) ∨ agentCCheck(parseToolCall(f_d(...)), trace)
  {
      return transferToHuman("Unable to satisfy prerequisites. Tool: " + tool_name);
  }
```

Figure 13. FGGM definition for $duplicateFGGM$ (τ^2-bench). Invoked when a duplicate tool call is detected. The prompt instructs the model to try a different approach. The f_{err} function reports the specific policy violation to prevent repeated non-compliant calls.

```
FGGM Definition:
  id := agentCErrorFGGM
  GMid := qwen
  typeSig := (messages : str, prev_response : str, agentc_error : str,
        tool_name : str, tools : str) → str
  Φ_l := true
  Ψ_l := (¬isToolCall(f(...))) ∨ agentCCheck(parseToolCall(f(...)), trace)
  f_p := function f_p(messages : str, prev_response : str, agentc_error : str,
        tool_name : str, tools : str) : (str) {
    var guidance : str := "Tool call '" + tool_name
      + "' failed agentC constraint: " + agentc_error + "\n\n";
    if (isSubstring("update_reservation", tool_name)) {
      guidance := guidance
        + "To update a reservation you need:\n"
        + "1. Call get_reservation_details first to get current state.\n"
        + "2. Check if cabin is basic_economy – if yes, cancel then book.\n"
        + "3. For flight changes: search for new flights first.\n"
        + "Try calling the missing prerequisite tool now.\n"; }
    else if (isSubstring("cancel_reservation", tool_name)) {
      guidance := guidance
        + "To cancel: call get_reservation_details first to verify eligibility.\n"
        + "Verify cancellation rules are satisfied.\n"; }
    else if (isSubstring("book_reservation", tool_name)) {
      guidance := guidance
        + "To book: call get_user_details for payment methods,\n"
        + "search for flights, ensure all passenger details.\n"; }
    else {
      guidance := guidance
        + "Call prerequisite tools first: get_user_details, get_reservation_details, or search flights.\n"; }
    return messages + " + [assistant: " + prev_response
      + ", user: " + guidance + "]"; }
  f_err := function f_err(messages : str, prev_response : str, agentc_error : str,
        tool_name : str, tools : str, y : str) : (str) {
      var tc : str := parseToolCall(y);
      var err : str := agentCCheckErrorMsg(tc, trace);
      return "Rejection: still violates policy. Original error: " + agentc_error
        + "\nNew error: " + err
        + "\nCall prerequisites before the action tool."; }
  f_d := function f_d(messages : str, prev_response : str, agentc_error : str,
        tool_name : str, tools : str, y : str) : (str)
      ensures (¬isToolCall(f_d(...))) ∨ agentCCheck(parseToolCall(f_d(...)), trace)
  {
      return transferToHuman("Unable to satisfy prerequisites. Error: " + agentc_error);
  }
```

Figure 14. FGGM definition for $agentCErrorFGGM$ (τ^2-bench). Invoked when a tool call fails the Agent-C policy check. The prompting function analyzes the tool name to provide targeted recovery guidance. The f_{err} function accumulates both the original and new policy errors, enabling the stateful rejection sampler to progressively narrow toward compliant actions.

```
method agent(conversation : str, tool_trace : str, available_tools : str,
    policy : str, domain : str, tools : str) returns (r : str)
  ensures isTextResponse(r) ∨ agentCCheck(r, tool_trace)
{
  var max_retries : int := 5;
  var retry_count : int := 0;
  var last_tool_name : str := "";
  var last_arguments : str := "";
  var prev_response : str := "";

  // — initial FGGM call —
  var response : str := initialFGGM(conversation, policy, tools);

  // — main loop —
  while (retry_count < max_retries)
    decreases max_retries − retry_count
  {
    var parsed : str := parseToolCall(response);

    if (¬hasName(parsed)) {
      r := makeTextResponse(response); return;
    }

    var tool_name : str := getName(parsed);
    var arguments : str := getArguments(parsed);

    // — duplicate detection —
    if (tool_name = last_tool_name ∧ arguments = last_arguments) {
      retry_count := retry_count + 1;
      if (retry_count ≥ max_retries) {
        r := transferToHuman("Unable to proceed. Tool: " + tool_name); return;
      }
      response := duplicateFGGM(conversation, prev_response, tool_name, tools);
      continue;
    }

    last_tool_name := tool_name;
    last_arguments := arguments;
    prev_response := response;

    // — agentC constraint check —
    var satisfies : bool, error_msg : str :=
      agentCCheckToolCall(tool_name, arguments, tool_trace, domain);

    if (satisfies ∧ tool_name ≠ "transfer_to_human_agents") {
      r := makeToolCallResponse(tool_name, arguments); return;
    } else {
      retry_count := retry_count + 1;
      if (retry_count ≥ max_retries) {
        r := transferToHuman("Constraint failures. Error: " + error_msg); return;
      }
      response := agentCErrorFGGM(conversation, prev_response, error_msg,
        tool_name, tools);
      continue;
    }
  }

  // — exhausted retries —
  r := transferToHuman("Unable to process request after multiple attempts"); return;
}
```

Figure 15. Verified agent program for τ^2-bench. The agent generates an initial response via $initialFGGM$, then enters a loop that detects duplicate tool calls (dispatching to $duplicateFGGM$) and checks Agent-C policy compliance (dispatching to $agentCErrorFGGM$ on failure). Each FGGM guarantees policy compliance via a rejection sampler with $K = 5$ and stateful f_{err} feedback. The fallback fires on 15.6% (retail) and 21.6% (airline) of instances (Table 7).

Q. 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 ($\langle$ident$\rangle$, $\langle$type$\rangle$, $\langle$spec$\rangle$, $\langle$program$\rangle$, $\langle$string_lit$\rangle$, $\langle$formula$\rangle$) from Appendix D.

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

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

S. 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:

$$A^{(j)} = \frac{r^{(j)} - \mu_p}{\sigma_p}, \quad \mu_p = \frac{1}{G}\sum_{j=1}^{G} r^{(j)}, \quad \sigma_p = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(r^{(j)} - \mu_p)^2 + \epsilon} \tag{15}$$

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

$$\max_{\theta} \ \mathbb{E}_{p\sim\mathbb{P}}\left[\frac{1}{G}\sum_{j=1}^{G}\min\left(\rho^{(j)}A^{(j)},\ \text{clip}(\rho^{(j)}, 1-\epsilon, 1+\epsilon)A^{(j)}\right)\right] \tag{16}$$

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. 14:

$$\mathcal{R}(p, y_l) = 1 - Sigmoid\big(\mathsf{L}(x, _, y) \times \mathbb{I}(y = y_l) + \lambda \times (1 - \mathbb{I}(check_{\mathcal{A},\Phi_l,\Psi_l}(p, y_l)))\big) \tag{17}$$

T. 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.

U. Planner Feedback

After each search–verify–learn iteration, SEVerA 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

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

previous candidate failed, SEVerA 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.

V. 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 U).

```
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}
```

Figure 16. 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.

W. Proof Details

Lemma B.3 (Completeness of Checker)**.** *For any valid FGGM* G *with quantifier-free* Ψ_l, $\forall x_1 \in T_1, \ldots, \forall x_n \in T_n, \forall y \in T_o.\, \Phi_l(x_1, \ldots, x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1, \ldots, x_n, y) \iff \Psi_l(x_1, \ldots, x_n, y))$.

Proof of Lemma B.3. 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 W.1 (Checker Soundness). $\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1,\ldots,x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \implies \Psi_l(x_1,\ldots,x_n,y))\big)$.

Proof. For quantifier-free Ψ_l soundness follows from Lemma B.3. 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 $(x_1,\ldots,x_n,y)$

$$\begin{aligned}
check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) &\implies (\varphi_{\mathcal{A}} \wedge \Phi_l(x_1,\ldots,x_n)) \wedge \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) &\implies \Phi_l(x_1,\ldots,x_n) \wedge \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \wedge \Phi_l(x_1,\ldots,x_n)) &\implies \Psi_l(x_1,\ldots,x_n,y)\\
(\varphi_{\mathcal{A}} \wedge \Phi_l(x_1,\ldots,x_n)) &\implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) \implies \Psi_l(x_1,\ldots,x_n,y))\\
\varphi_{\mathcal{A}} &\implies \big(\Phi_l(x_1,\ldots,x_n)) \implies ((check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y)) \implies \Psi_l(x_1,\ldots,x_n,y))\big) \qquad (18)
\end{aligned}$$

From Eq 18 it follows , $\varphi_{\mathcal{A}} \implies \big(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n, \forall y \in T_o.\ \Phi_l(x_1,\ldots,x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,y) \implies \Psi_l(x_1,\ldots,x_n,y))\big)$. □

Theorem 3.1 (Valid Local Contract). *For any valid FGGM* G *with* (Φ_l,Ψ_l) *and parametric GM* f^{Θ}_{G}, $\varphi_{\mathcal{A}} \implies \big(\forall\theta \in \Theta, \forall x_1 \in T_1,\ldots,\forall x_n \in T_n.\ \Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n, id_{\mathsf{G}}(x_1,\ldots,x_n))\big)$.

Proof. Let $r = id_{\mathsf{G}}(x_1,\ldots,x_n)$. 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 where y_f is the final rejected sample.

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies (check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r) \vee (r = f_d(x_1,\ldots,x_n,y_f))) \qquad (19)$$

Eq. 20 follows from the validity of the fallback f_d, and Eq. 8 follows from the soundness of the checker $check_{\mathcal{A},\Phi_l,\Psi_l}$ (lemma W.1). To simplify the notation, we used $(\forall x_i \in T_i)$ to denote $(\forall x_1 \in T_1,\ldots,\forall x_n \in T_n)$.

$$\varphi_{\mathcal{A}} \implies \forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \wedge (r = f_d(x_1,\ldots,x_n,y_f)) \implies \Psi_l(x_1,\ldots,x_n,r) \qquad (20)$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).(\Phi_l(x_1,\ldots,x_n) \wedge check_{\mathcal{A},\Phi_l,\Psi_l}(x_1,\ldots,x_n,r)) \implies \Psi_l(x_1,\ldots,x_n,r)) \qquad (21)$$

$$\varphi_{\mathcal{A}} \implies (\forall\theta \in \Theta.(\forall x_i \in T_i).\Phi_l(x_1,\ldots,x_n) \implies \Psi_l(x_1,\ldots,x_n,r)) \quad \text{Using Eq. (19, 20, 21)}$$

□

Theorem C.1. *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.

$$\begin{aligned}
&\mathcal{A}_{\mathcal{G}} = \{\varphi_{\mathsf{G}_1}, \ldots, \varphi_{\mathsf{G}_k}\} \text{ where } \varphi_{\mathsf{G}_i} \text{ defined below} \\
&\varphi_{\mathsf{G}_i} : \forall x_1 \in T_1, \ldots, x_n \in T_n.\mathsf{G}_i^{\Phi_l}(x_1, \ldots, x_n) \implies \mathsf{G}_i^{\Psi_l}(x_1, \ldots, x_n, \mathsf{G}_i(x_1, \ldots, x_n)) \\
&(\varphi_{\mathcal{A}} \implies (\forall \theta_1 \in \Theta_1.\varphi_{\mathsf{G}_1})) \wedge \cdots \wedge (\varphi_{\mathcal{A}} \implies (\forall \theta_k \in \Theta_k.\varphi_{\mathsf{G}_k})) \quad \text{from Theorem 3.1} \\
&\big(\varphi_{\mathcal{A}} \implies \wedge_{i=1}^{k}(\forall \theta_i \in \Theta_i.\varphi_{\mathsf{G}_i})\big)
\end{aligned} \tag{22}$$

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

$$\begin{aligned}
(\mathsf{P}_{\{\Theta\}} \neq \bot) \implies ((\varphi_{\mathcal{A}} \wedge (\wedge_{i=1}^{k} \forall \theta_i \in \Theta_i.\varphi_{\mathsf{G}_i})) \implies \\
\forall x_1 \in T_1 \ldots \forall x_n \in T_n.\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1, \ldots, x_n)) \\
(\mathsf{P}_{\{\Theta\}} \neq \bot) \implies ((\varphi_{\mathcal{A}} \implies \\
\forall \theta_1 \in \Theta_1, \ldots, \forall \theta_k \in \Theta_k, \forall x_1 \in T_1 \ldots \forall x_n \in T_n. \\
\Phi(x_1, \ldots, x_n) \implies \Psi(x_1, \ldots, x_n, \mathsf{P}_{\{(\theta_1,\ldots,\theta_k)/\Theta\}}(x_1, \ldots, x_n)) \text{ Using Eq. 22 and Eq. 23}
\end{aligned} \tag{23}$$

□

Theorem 3.2 (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. $f^* \leftarrow \arg\min_{f \in \mathcal{P}} \sum_{(x,y) \in \mathrm{D}} \mathsf{L}(x, y, f(x))$. If $f^* \neq \bot$, then by definition $(\mathcal{P} \neq \{\}) \wedge (f^* \in \mathcal{P})$. By Theorem C.1, 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 3.3 (Sufficient Success Condition)**.** *If (i)* L *penalizes constraint violations:* $\forall x \in T_i, \forall y, y' \in T_o.(\Phi(x) \wedge \neg\Psi(x, y) \wedge \Psi(x, y')) \implies \mathsf{L}(x, _, y') < \mathsf{L}(x, _, y)$*; (ii)* Ψ *is quantifier-free; and (iii) there exists a non-parametric* $f_d \in \mathsf{S}(G, \mathcal{F}_c)$ *satisfying* (Φ, Ψ)*. Then for any type-correct GM* $f_n^\theta : T_i \to T_o$ *with initial parameters and any* f_p*, there exists* $f \in \mathsf{S}(G, \mathcal{F})$ *such that (1)* f *satisfies* (Φ, Ψ) *and (2)* $\mathsf{L}(f) \leq \mathsf{L}(f_n^\theta)$*, with strict improvement whenever* f_n^θ *violates* Ψ *on some* $(x, _) \in \mathrm{D}$.

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 B.3).

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 3.1,

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

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,

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

which implies

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

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. □