

THE GRAINGER COLLEGE OF ENGINEERING
SIEBEL SCHOOL OF COMPUTING AND DATA SCIENCE

Lean4Agent

Formal Modeling and Verification for Agent Workflow and Trajectory

Ruida Wang, Jun 18th 2026

ILLINOIS

Introduction to Lean4

Lean4 is a tool that makes math formal and verifiable:

- It is a dependent-type-based programming language, which is currently mainly used to formalize mathematical theorem proving
- A traditional Lean application is to transform every theorem statement into a condition-goal state space to reason over it.
- But it is fundamentally a type system, where the “proof” is actually a conversion argument.
- However, the potential for such language is way beyond math proofs. It can be used to prove the behavior of computer programs.
- Furthermore, our current research aims to extend such proof to LLM behaviors.

Lean4 in formal Math reasoning

```
theorem amc12_2001_p9
  (f : ℝ → ℝ)
  (h₀ : ∀ x > 0, ∀ y > 0, f (x * y) = f x / y)
  (h₁ : f 500 = 3) :
    f 600 = 5 / 2 := by
  -- We are given that f satisfies the functional equation
  -- f(xy) = f(x) / y for all positive x and y.
  -- We also know that f(500) = 3.

  -- To find f(600), we can let x = 500 and y = 6/5.
  specialize h₀ 500 _ (6 / 5) _
  · linarith -- x = 500 is positive
  · linarith -- y = 6/5 is positive

  -- Now, calculate using the functional equation:
  calc
    f 600 = f (500 * (6 / 5)) := by
      congr
      norm_num
    -- f(600) = f(500 * (6/5)) by substitution
    _ = f 500 / (6 / 5) := by
      rw [h₀]
    -- = f(500) / (6/5) by the functional equation
    _ = 3 / (6 / 5) := by
      rw [h₁]
    -- = 3 / (6/5) since f(500) = 3
    _ = 5 / 2 := by
      norm_num
    -- = 5/2 after simplifying the division
```

```
1 goal
f : ℝ → ℝ
h₀ : ∀ x > 0, ∀ y > 0, f (x * y) = f x / y
h₁ : f 500 = 3
⊢ f 600 = 5 / 2
```

Applying specific number to h_0

```
1 goal
f : ℝ → ℝ
h₁ : f 500 = 3
h₀ : f (500 * (6 / 5)) = f 500 / (6 / 5)
⊢ f 600 = 5 / 2
```

All goals solved, theorem proved!

```
No goals
```

Lean4 in Verifying Programs

```
-- Description of the coding problem in natural language
-- Remove an element from a given array of integers at a specified index. The resulting array should
-- contain all the original elements except for the one at the given index. Elements before the
-- removed element remain unchanged, and elements after it are shifted one position to the left.
-- Code implementation
def removeElement (s : Array Int) (k : Nat) (h_precond : removeElement_pre s k) : Array Int :=
  s.eraseIdx! k
-- Pre-condition
def removeElement_pre (s : Array Int) (k : Nat) : Prop :=
  k < s.size -- the index must be smalle
-- Post-condition
def removeElement_post (s : Array Int) (k : Nat) (res: Array Int) (h_precond : removeElement_pre s k)
    : Prop :=
  res.size = s.size - 1 ∧ -- Only one element is removed
  (∀ i, i < k → res[i]! = s[i]!) ∧ -- The elements before index k remain unchanged
  -- The elements after index k are shifted by one position
  (∀ i, i < res.size → i ≥ k → res[i]! = s[i + 1]!)
-- Proof (proof body omitted for brevity)
theorem removeElement_spec (s: Array Int) (k: Nat) (h_precond : removeElement_pre s k) :
  removeElement_post s k (removeElement s k h_precond) h_precond := by sorry
```

The definition of a program

The precondition of the program

The postcond of the program

Theorem indicate that the program is correct

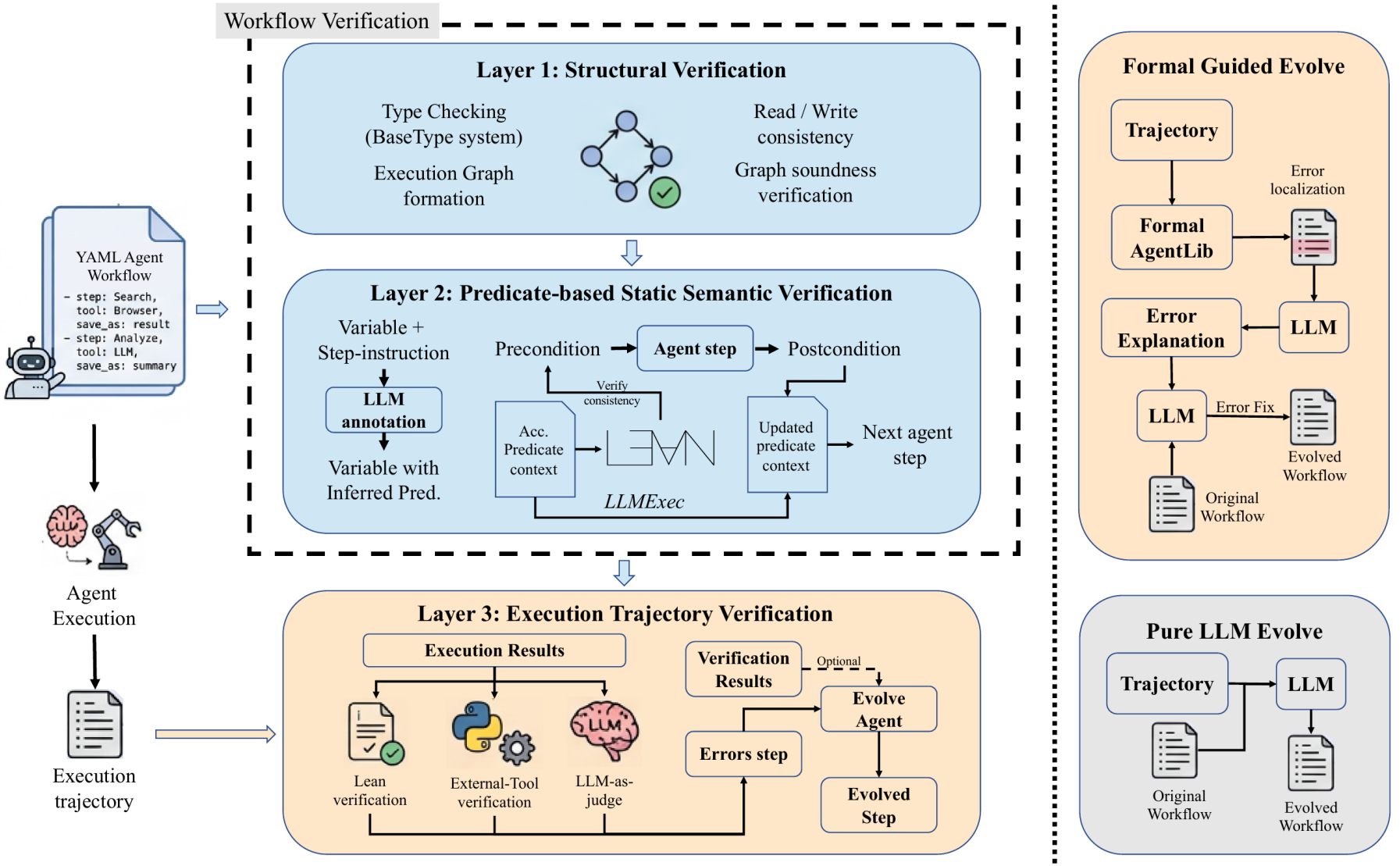
Why do we need to verify the agent

- **The reliability gap:**
  - Agents are used in high-stakes tasks.
  - They are less human-controlled.
  - Most systems lack formal guarantees.
- **Existing verifications are fragmented**:
  - Current approaches can only address one slice of the problem.
  - They are limited by the expressiveness of their formalism.
- **Natural languages are inherently ambiguous:**
  - Just like NL proofs, NL-specified agent workflows carry hidden semantic ambiguity
- **What we did in math:**
  - Formal theorem proving
- **We extend it to agent systems**

Problem formulation

- Two stages for agent systems:
  - Human-defined workflow
  - LLM-motivated execution
- The problem we want to resolve:
  - How to dig the hidden requirements in natural language prompts.
  - How to make sure the human-defined workflows are self-consistent.
  - How to judge whether the real LLM-execution meets our requirements
- So our problem is: Given an LLM workflow, can we uniformly model and verify the workflow and its execution?

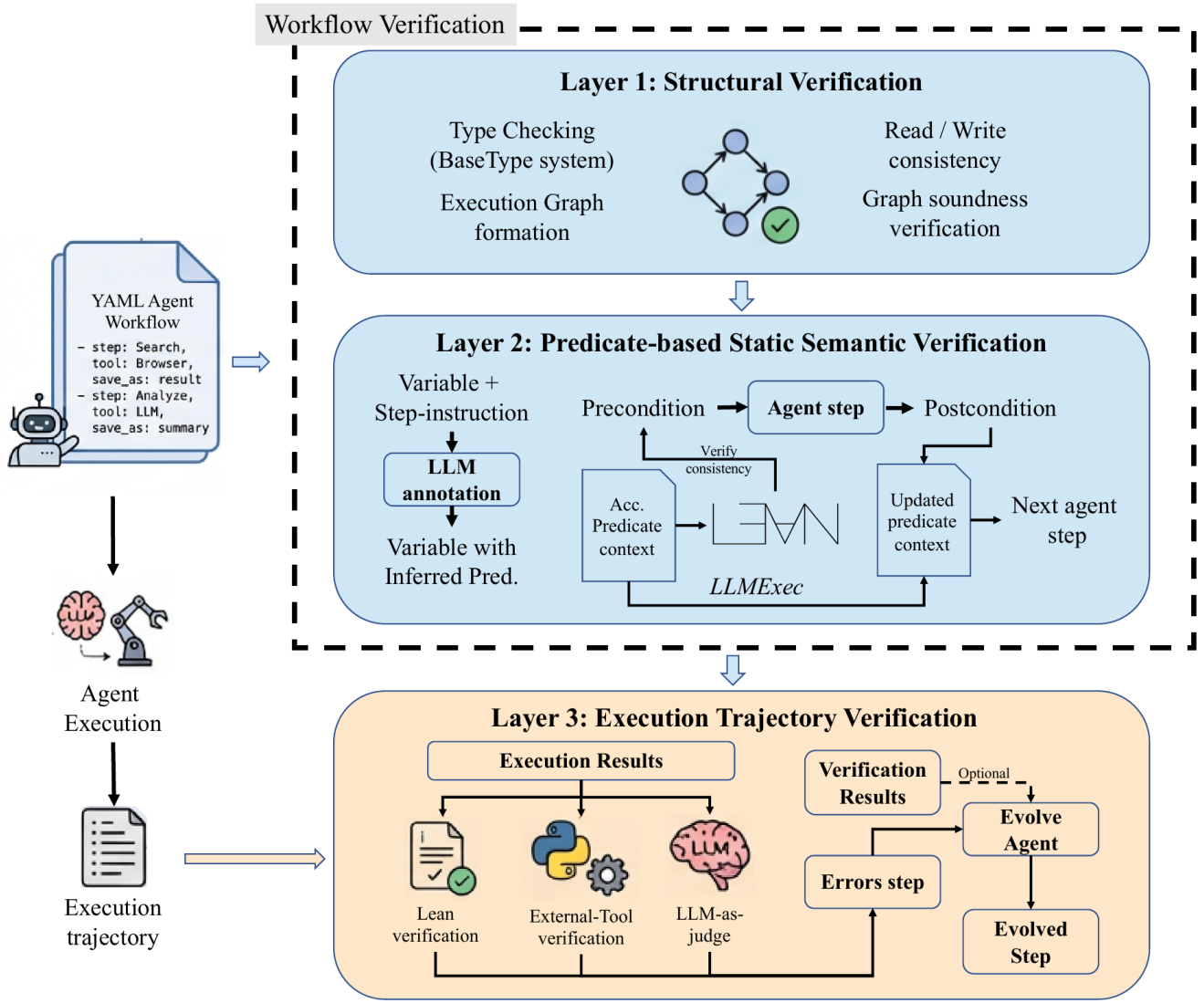
Overview of Lean4Agent



(a) FormalAgentLib (b) LeanEvolve

FormalAgentLib: Uniform verification of workflow and trace

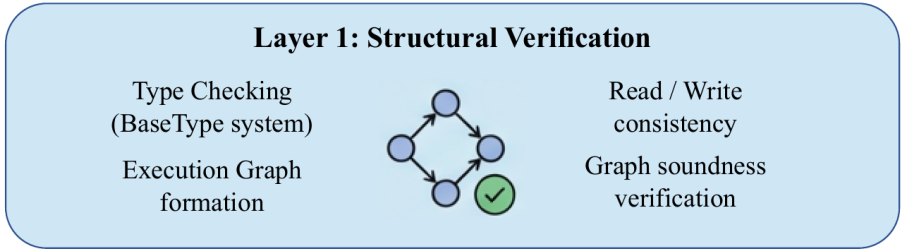
- **The three-layer structure:**
  - Static structural layer
  - Static Semantic layer
  - Dynamic layer
- To formally model the ambiguous NL, we implement the predicate system.
- Given a YAML agent workflow, we verify its self-consistency through the first two layers
- After its execution, we can use the dynamic layer to check whether the conditions are satisfied.

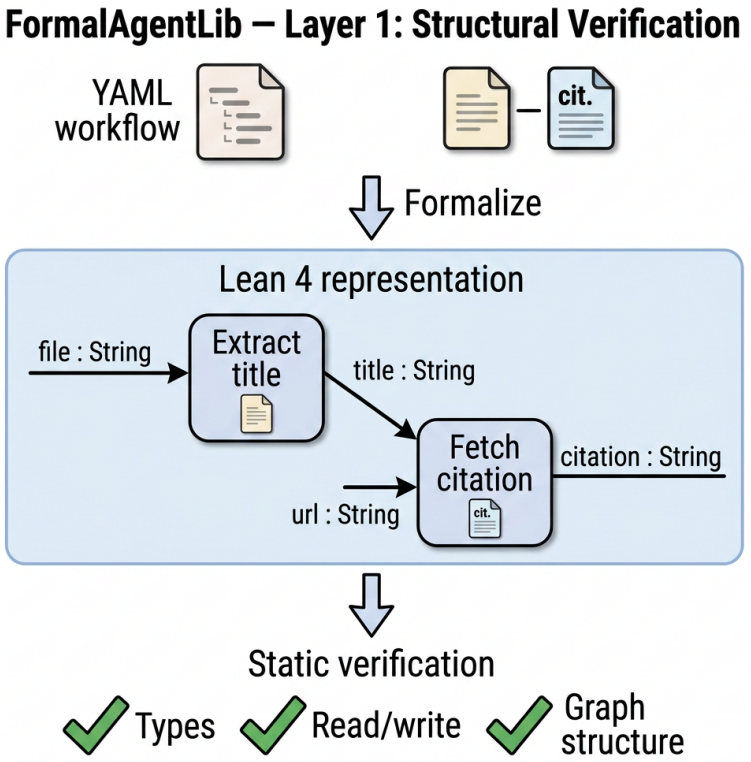


(a) FormalAgentLib

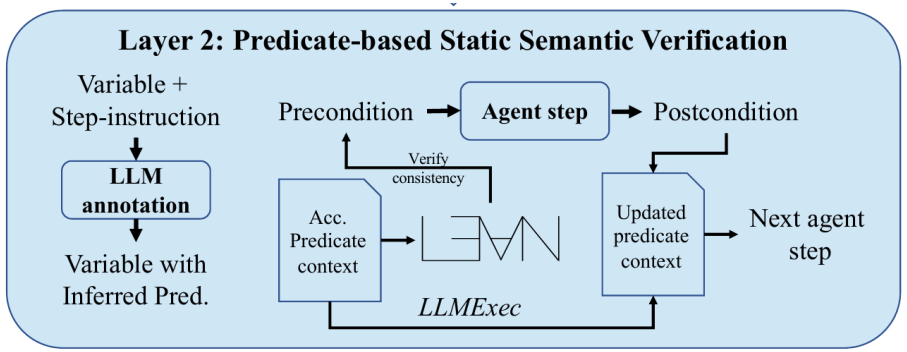
FormalAgentLib-Layer 1: Structural Verification

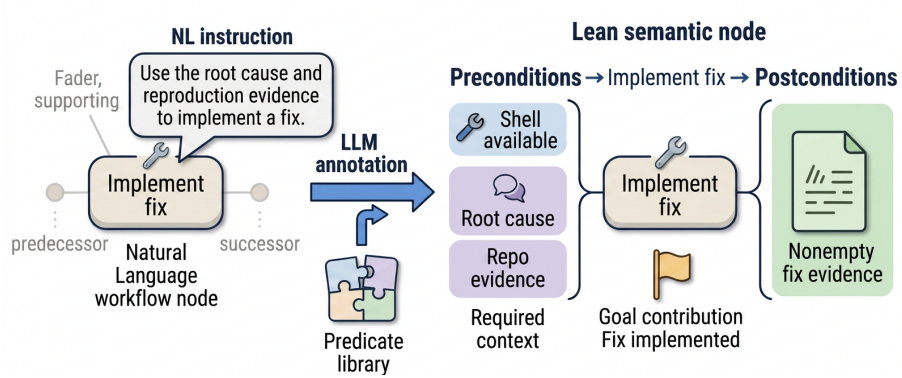
- In this layer, we perform static analysis on the YAML agent plan.
- This results in a traditional Workflow Graph in Lean.
- From the graph, we extract the variables needed at each execution step and determine their static types.
- With the graph and variable information, we are able to analyze the consistency of read and write
- Additionally, the graph structure is used to aggregate information for future layers.
- This layer settle foundations for our entire framework.





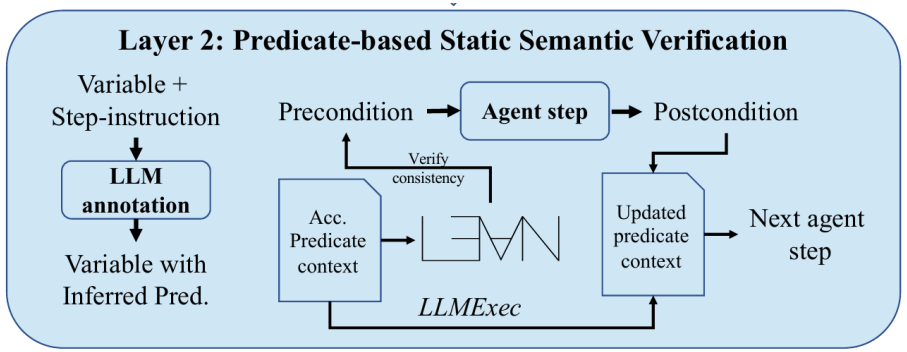
FormalAgentLib-Layer 2: Predicate-based Static Semantic Verification

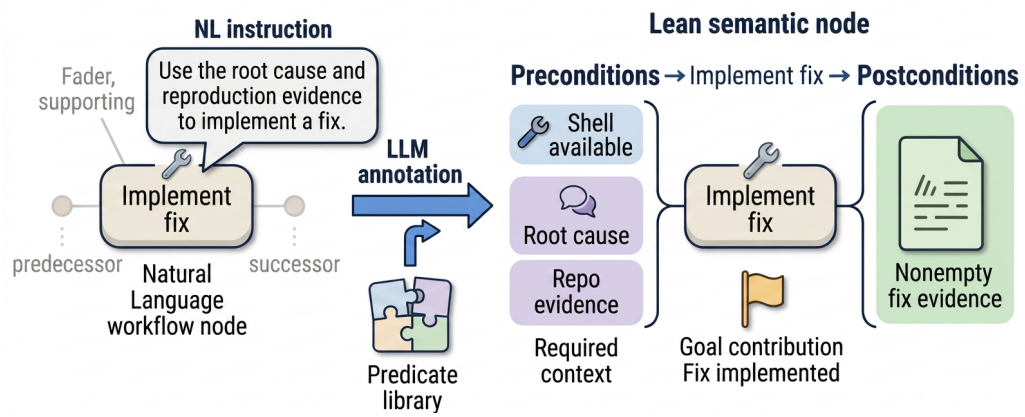




- This layer is the core of our verification. It aims to address the vital question of how to model ambiguous NL instruction in a formal context.
- In our framework, we model the agent's execution using pre- and post-condition states. We assume that the LLM can move the state from pre to post.
- To model the state, we design a highly extensible predicate system to represent the pre- and post-conditions.
- The predicate declares the property of the variable or execution step.
- We have a set of base predicate definitions, an extensible system, and a logical operation system for predicates.

FormalAgentLib-Layer 2: Predicate-based Static Semantic Verification

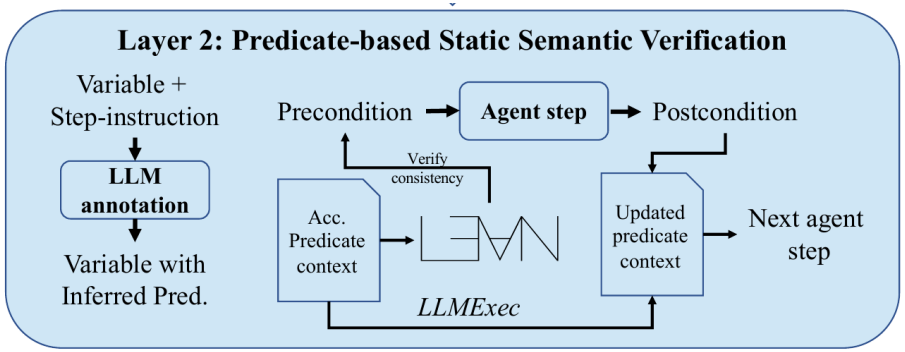


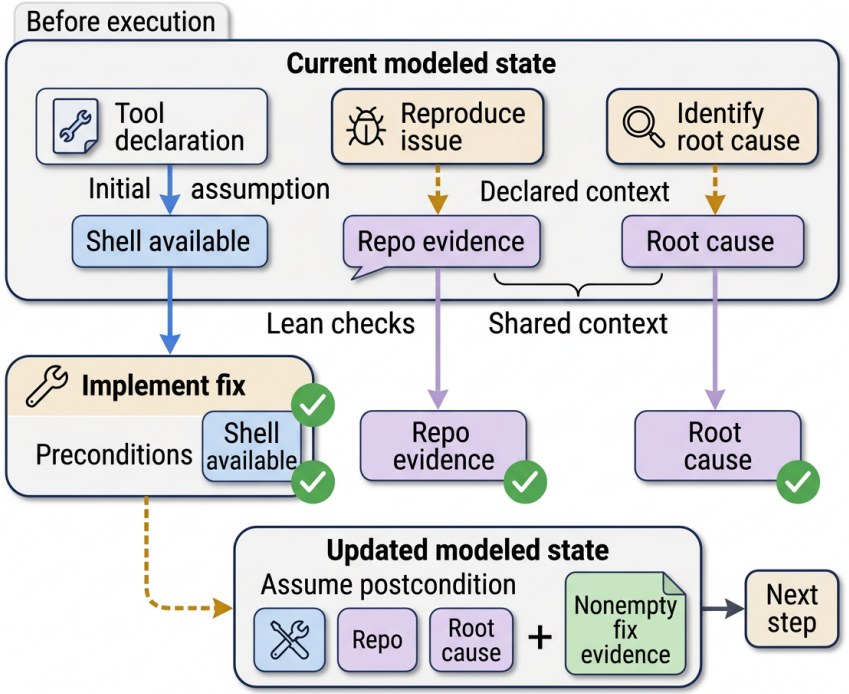


- In this layer, the predicates can be divided into three components: Variable-level, information-flow-level, and graph-level.
- Variable-level predicate: It models the basic properties of a variable. The basic building block for our system.
- Information flow predicates: Similar to the variable level, but this serves more as the implicit information that a step requires or carries.
- Graph-level predicates: It models the higher-level prosperity of the entire workflow. It is defined as the pre- and post-condition for the workflow.

FormalAgentLib-Layer 2: Predicate-based Static Semantic Verification

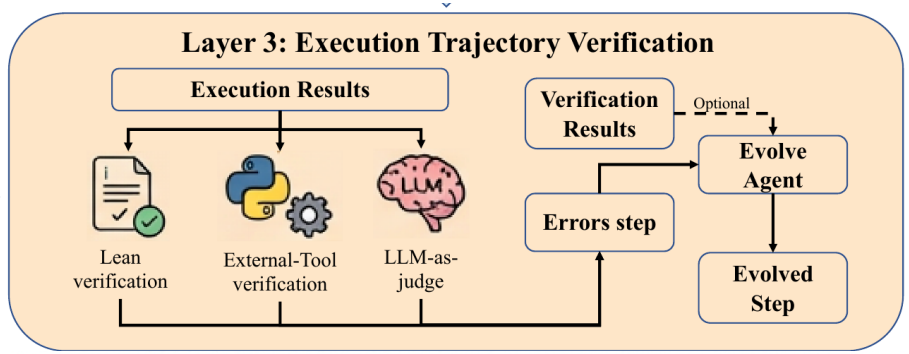
- With the predicate system, we can verify the static semantic correctness of a workflow.
- In particular, we define such correctness as the self-consistency of the workflow.
- In this step, we first verify whether the precondition is satisfied by the current state space.
- After we verify that the precondition holds, we add the postcondition to the state space for further verification.
- This system also supports abstracting submodules into pre- and post-condition regularization.
- This layer enables us to measure the “quality” of a workflow before any execution really happens.



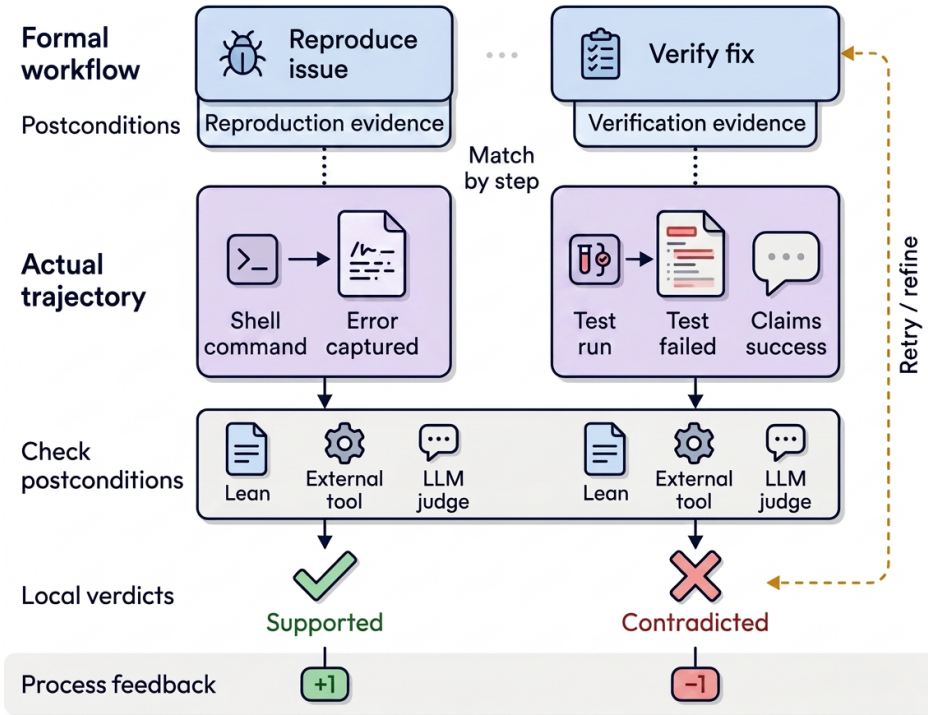


FormalAgentLib-Layer 3: Execution Trajectory Verification

- Based on the predicate schema, with real execution trajectory, we are able to discharge our assumption in layer-2.
- Specifically, we verify whether the postcondition is satisfied through:
  - Lean verification
  - External tool verification
  - LLM-as-judge check
- In this layer, unsatisfied postconditions indicate a flaw in the execution and can be used for immediate retry or workflow refinement.

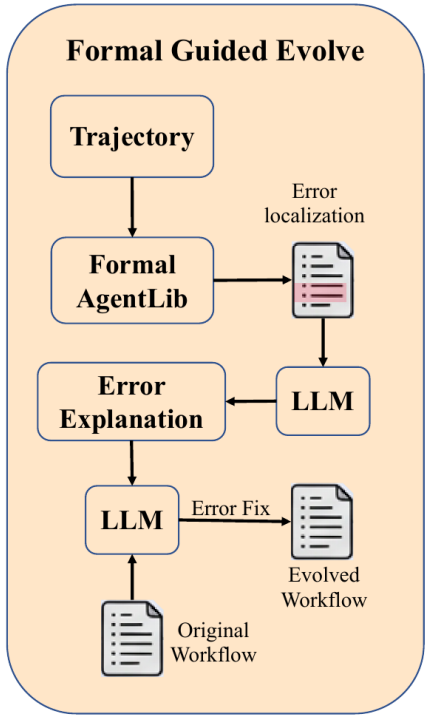


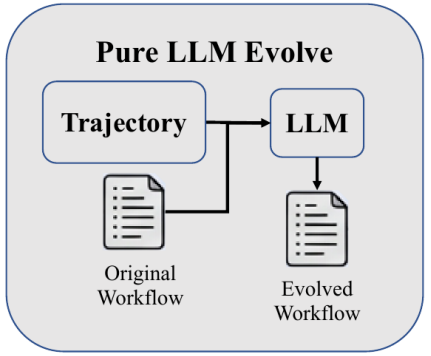
Lean4Agent verifies a real execution trajectory at formalitzed Layer 3



LeanEvolve: Refining agent workflow

- Our formal verification can not only make the agent execution more reliable and secure. Through LeanEvolve design, we are able to enhance the capability of an agentic system.
- It applies the FormalAgentLib's error-localization capability.
- When a trajectory has an error, we can identify exactly which steps are problematic. Then we can use LLM to further refine that step.
- This method can be combined with the pure LLM evolve to further enhance its capability.





Experiments: Design and Setup

- **Experiment Goal:**
  - Does formal verification identify better agent workflows?
  - Can verification feedback improve failed executions?
- **FormalAgentLib verification:** Generate multiple candidate workflows -> automatically annotate the Layer-2 semantic specifications -> compare results of verification-passing to failing workflows
- **LeanEvolve refinement:** Run the verified workflows and inspect failed trajectories -> use Layer-3 diagnostics to localize failure steps -> combine formal guidance with optional pure LLM-based evolve
- **Benchmarks to test:** SWE-Bench and ELAIP-Bench
- **Cross model evaluation:**
  - 5 leading LLMs, both open- and closed-source.
  - Same workflow pool and annotation methods.

Experiment Result: FormalAgentLib-Layer 2

Table 1: Main experimental results of **FormalAgentLib**-Layer 2 verification.

Model	SWE-Bench-Verified			ELAIP-Bench		
	Avg. Passed	Avg. Failed	Diff.	Avg. Passed	Avg. Failed	Diff.
Large Models						
GPT-5.2	62.67%	50.00%	12.67%	32.00%	20.33%	11.67%
GLM-5	58.00%	50.67%	7.33%	37.33%	25.00%	10.83%
Kimi-K2.5	61.33%	46.00%	15.33%	36.00%	28.33%	7.67%
Small Models						
Gemma-4-31B	60.00%	32.67%	27.33%	35.67%	27.33%	8.34%
Qwen-3.5-27B	49.33%	38.00%	11.33%	42.00%	36.67%	5.33%
Average	58.27%	43.47%	**14.80%**	36.60%	27.53%	**9.07%**

Experiment Result: FormalAgentLib-Layer 2

Table 5: SWE-Bench Jimenez et al. [2024] subset's results with 95% CI

Model	Avg. Passed	Avg. Failed	Diff.	Lower CI	Upper CI
Large Models					
GPT-5.2	62.67%	50.00%	12.67%	5.33%	20.67%
GLM-5	58.00%	50.67%	7.33%	0.67%	14.00%
Kimi-K2.5	61.33%	46.00%	15.33%	6.00%	24.67%
Small Models					
Gemma-4-31B	60.00%	32.67%	27.33%	18.67%	36.00%
Qwen-3.5-27B	49.33%	38.00%	11.33%	2.67%	20.00%
Average	58.27%	43.47%	**14.80%**	10.00%	19.60%

Table 6: ELAIP-Bench-Verified Dai et al. [2025] subset's results with 95% CI

Model	Avg. Passed	Avg. Failed	Diff.	Lower CI	Upper CI
Large Models					
GPT-5.2	32.00%	20.33%	11.67%	5.00%	18.67%
GLM-5	37.33%	25.00%	10.83%	6.33%	18.33%
Kimi-K2.5	36.00%	28.33%	7.67%	1.33%	14.00%
Small Models					
Gemma-4-31B	35.67%	27.33%	8.34%	2.33%	14.33%
Qwen-3.5-27B	42.00%	36.67%	5.33%	-0.33%	10.67%
Average	36.60%	27.53%	**9.07%**	5.66%	13.07%

Experiment Result: LeanEvolve

Table 2: **LeanEvolve** enhanced performance of SWE-Bench hard problem subset.

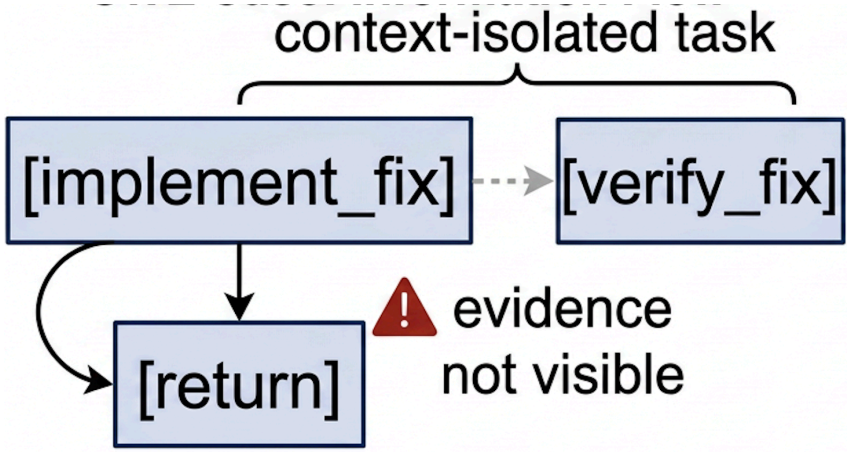
Model	Add. Solved	Overall
GPT-5.2	8.00%	70.67%
GLM-5	4.67%	62.67%
Kimi-K2.5	8.67%	70.00%
Gemma-4-31B	5.33%	65.33%
Qwen-3.5-27B	10.67%	60.00%

Table 3: Formal-guided evolve compare with pure-LLM evolve in ELAIP-Bench subset.

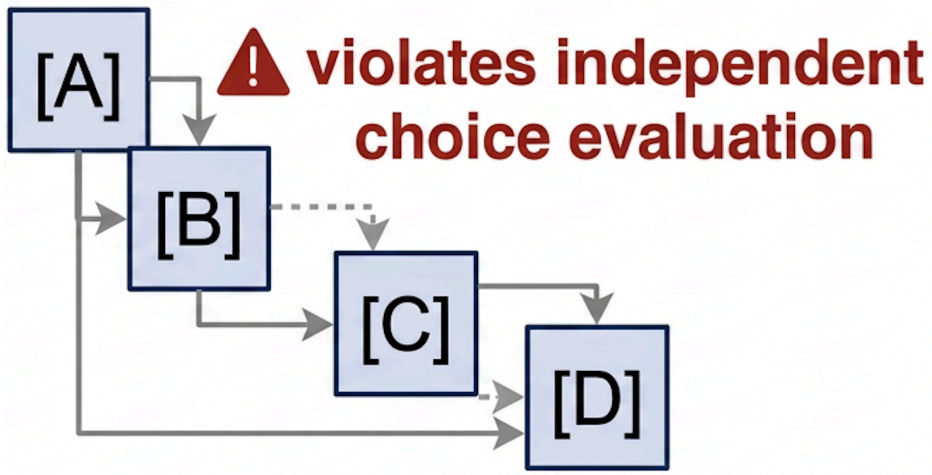
Model	Pure-LLM	Formal-guided
GPT-5.2	18.33%	25.00%
GLM-5	11.67%	22.33%
Kimi-K2.5	18.33%	21.67%
Gemma-4-31B	21.67%	28.33%
Qwen-3.5-27B	6.67%	13.33%

Experiments: Case Study

- **Workflow Errors identified:**
  - Detects hidden workflow errors before rollout.
  - SWE Case: context management error
  - ELAIP Case: The independent evaluation of choices affects the accuracy.

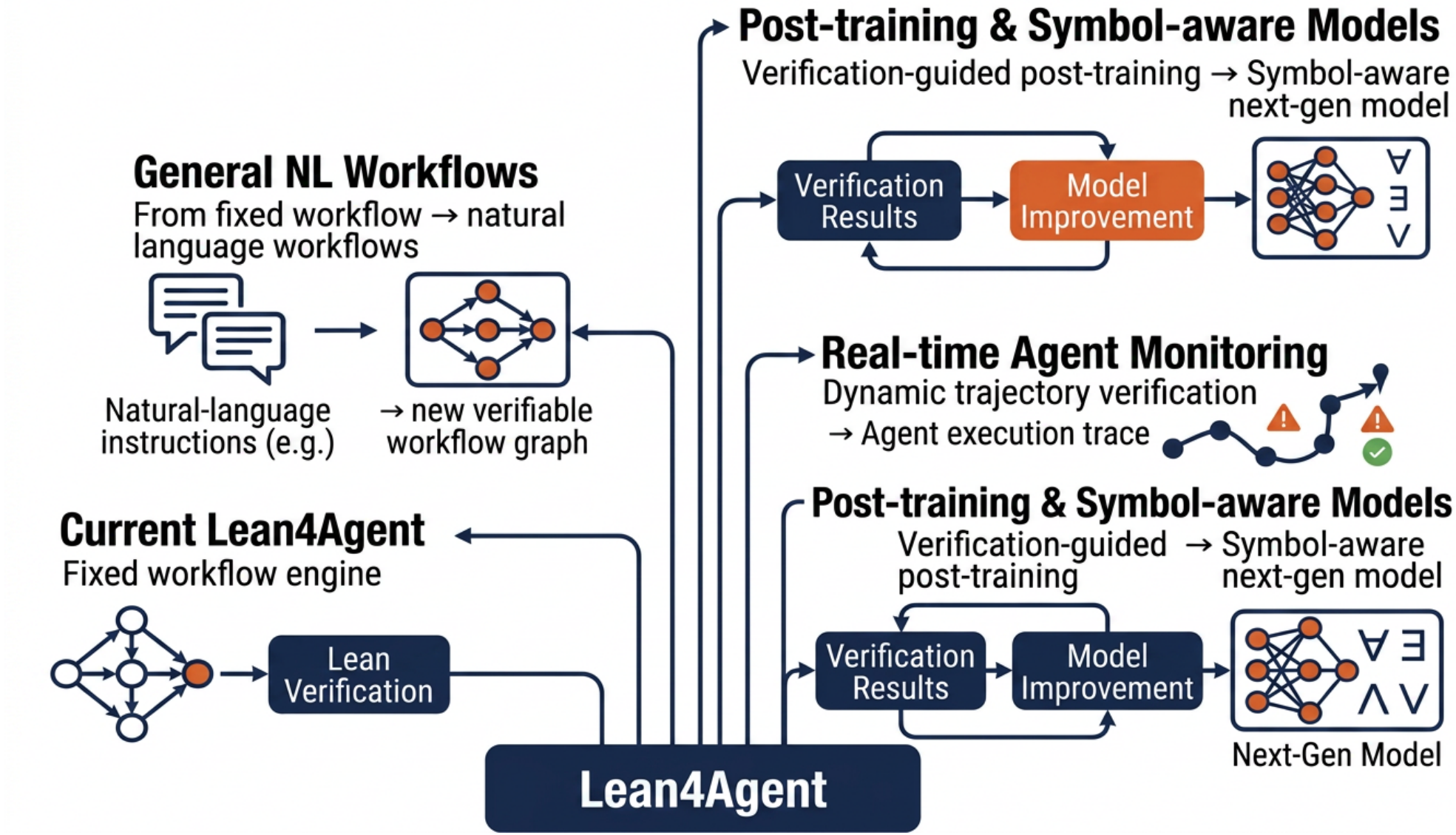


(a) SWE Workflow Error



(b) AI paper understanding WF error

Ongoing works and future directions



Philosophical bird-view of the entire project

- The philosophical ideas underlying this work are mainly two: a symbolic-probabilistic view of the world and a recursive assumption.
- Symbolic-probabilistic view of the world:
  - The world is dominated by a sequence of underlying rules that follow some rigorous logical or symbolic rules. This is the embedded space.
  - Our current physical world, which we can touch and sense, is based on probabilistic sampling from the embedded world.
- Recursive assumption:
  - In terms of task execution, we should adopt a recursive view.
  - Each layer has its own role, and most of the time, we should care only about the input and output, ignoring the intermediate process.
  - This leads us to design a system that uses predicates to regulate submodule behavior.



The Grainger College of Engineering

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN