Neurosymbolic Natural English Contract to Verified Implementation Generation for SysML v2 Models

Amer Tahat, David Hardin, Isaac Amundson, and Darren Cofer
Collins Aerospace, An RTX Business
DARPA PROVERS Program
{amer.tahat, david.hardin, isaac.amundson, darren.cofer}@collins.com

***Abstract*—The development of high-confidence avionics systems demands rigorous traceability from requirements to implementation, yet manual allocation often severs the link between high-level assurance cases and low-level code. While Generative AI (GenAI) offers a potential solution for automation, its deployment in safety-critical workflows is severely constrained by hallucinations, prohibitive token costs, and two critical barriers: the scarcity of open-source domain data and the restriction on fine-tuning state-of-the-art proprietary models (e.g., GPT-5 class) for open use.**

In this paper, we introduce a novel neuro-symbolic pipeline designed to bridge this gap. Our approach integrates state-of-the-art Codex agents into the DARPA PROVERS INSPECTA toolchain, a workflow that orchestrates the automated translation of SysML v2 system models into high-assurance code. By leveraging the HAMR code generator and Logika verifier, the pipeline ensures that properties established in the architectural model are mathematically preserved in the final implementation.

To circumvent fine-tuning limitations, we introduce a novel *Meta-Rule* methodology that functions as a verifiable surrogate for black-box weight adaptation. Unlike transient prompting strategies, this framework *crystallizes* ephemeral In-Context Learning (ICL) into persistent, expert-curated abstraction patterns. This allows the system to iteratively "adapt" its formal translation logic and drive a self-healing generate–verify–repair loop (translating English to GUMBO contracts and HAMR code), where HAMR/Logika feedback triggers largely automated, Meta-Rule guided revisions with minimal user intervention until system model integration and code-level verification succeed, effectively bypassing the data scarcity and proprietary restrictions that currently prohibit fine-tuning state-of-the-art models.

We evaluate this methodology on the Isolette Infant Incubator, utilized here as a standard proxy for safety-critical subsystems. From natural language requirements, our operational copilot achieved 100% formal code-level verification for the application logic in 25 minutes using $\sim$4 million tokens. In contrast, the baseline failed to produce verifiable artifacts despite consuming nearly 19 million tokens while processing extensive system documentation ($>$37 pages). The Meta-Rule–guided approach demonstrated a 58$\times$ increase in task throughput, reconciling the efficiency of GenAI with the strict rigor required for certifiable critical system software. We conclude by discussing current limitations and outlining future directions for scaling this architecture to broader cyber-physical applications.

***Index Terms*—SWE agents, Neuro-symbolic AI, Formal Methods, SysML v2, MBSE, Supervised Self-Healing, Supervised Self-Adaptation.**

I. Introduction

The development of high-confidence digital avionics systems faces a fundamental tension between increasing complexity and the rigid demands of certification. A primary challenge in this domain is ensuring rigorous traceability from high-level natural language requirements to low-level implementation. In traditional Model-Based Systems Engineering (MBSE) workflows, the manual allocation of requirements often severs the traceability link between the architectural assurance case and the software code. Without strong model-to-implementation synchronization, critical design flaws are often discovered late in integration or after fielding, driving up costs and jeopardizing safety certification [1].

Generative AI (GenAI), and especially Large Language Models (LLMs), appear well-suited to automate parts of this transition by translating requirements into structured specifications, contracts, and implementation scaffolding. However, directly deploying LLMs in safety-critical workflows remains problematic. Beyond stochastic hallucinations and the practical burden of token-heavy prompting, two structural barriers limit adoption in aerospace environments: (i) scarce open-source, domain-representative datasets that would enable robust specialization, and (ii) restrictions on fine-tuning proprietary, state-of-the-art models (e.g., GPT-5.2 Max at the time of writing this paper) [**?**]. As a result, organizations are left with an uncomfortable choice between unadapted general-purpose models and costly, fragile prompt engineering or fine-tuning earlier models that are generally less capable—neither of which naturally produces the traceability and evidence expected in certification-oriented development.

To address these barriers, we introduce a novel neuro-symbolic **Spec-Code-Proof (SCP)** pipeline designed to bridge the gap between natural language requirements and verified implementation. Our approach integrates state-of-the-art Codex software engineering agents into the DARPA PROVERS **INSPECTA** (Industrial-Scale Proof Engineering for Critical Trustworthy Applications) toolchain [1] as a user-friendly Spec-Code- Proof (SCP) SysMl v2 Codex Copilot. INSPECTA provides the necessary formal rigor by leveraging **SysML v2** [**?**] for architectural definition and the **GUMBO** contract language [**?**] for behavioral specification. Crucially, the pipeline automates the translation of these models into executable, memory-safe code using the **HAMR** (High Assurance Modeling and Rapid engineering) framework [**?**], [**?**] and mathematically proves correctness using the **Logika** verifier [**?**], which utilizes SMT solvers such as Z3 [**?**] and CVC5 [**?**].

A. Theoretical Motivation: Meta-Rules as Persistent ICL

A key innovation of our approach is the method by which we adapt the LLM to this complex toolchain without fine-tuning. We rely on a mechanism grounded in recent interpretability research. Recent theoretical work by von Oswald et al. mechanistically demonstrates that In-Context Learning (ICL) functions as a form of "implicit fine-tuning," where the model updates its internal representations via gradient descent dynamics during the forward pass [**?**]. However, this learned capability is ephemeral; once the context window is reset, the knowledge is lost.

The SCP Codex Copilot bridges this gap by *crystallizing* high-quality ICL interactions—derived from golden examples and human feedback—into "Meta-Rules." We define Meta-Rules as reusable assets that transfer ephemeral ICL knowledge into persistent, verified formalization logic. This rules-based framework functions as a verifiable surrogate for black-box weight adaptation, effectively capturing the results of implicit optimization. This approach eliminates the need for repetitive, token-heavy context prompting while avoiding the proprietary and computational constraints of traditional parameter fine-tuning.

B. Related Work

Recently, generative AI, and particularly LLMs, have shown promising potential to improve explainability and guide automated formal verification. Early efforts include OpenAI's GPT-f, which achieved notable success in Metamath theorem proving [**?**], [**?**]. Other initiatives have applied LLMs successfully to proof repair in Isabelle/HOL [**?**], theorem diagnosis in Coq [**?**], and discovering program invariants [**?**], [**?**]. Stanford and VMware's Clover project represents another significant step forward, focusing on verifiable code generation with generative assistance [**?**].

In the domain of conversational formal reasoning, Tahat et al. demonstrated high success rates using multi-turn conversational LLMs for proof repair in Coq, underscoring conversational learning's value in formal reasoning domains [**?**]. However, purely neural approaches face limits; Apple's GSM-Symbolic [**?**] highlighted fundamental limitations of LLMs in symbolic reasoning tasks. Similarly, Amazon's recent SMT-backed hallucination prevention framework [**?**], while innovative, remains closed-source, available exclusively as a web service, and has yet to integrate within aerospace-specific MBSE pipelines. Closer to our architectural focus, Tahat et al. introduced *AGREE-Dog* [**?**], a neuro-symbolic copilot for the OSATE AADL environment that automates explainable compositional reasoning and repair, laying the groundwork for the SysML v2 approach presented in this paper.

C. Contributions and Organization

The main contributions of this paper are:

1) **Trustworthy English-to-Implementation Pipeline:** A multi-agent framework integrated with the INSPECTA toolchain (HAMR/Logika) that generates verified contracts and code from requirements.

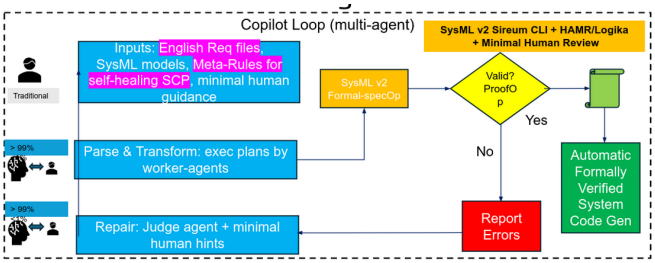

Fig. 1. SCP copilot self-healing neuro-symbolic code generation loop. It parses English requirements, applies meta-rules, and utilizes the INSPECTA toolchain for verification.

2) **Meta-Rules Methodology:** A novel neuro-symbolic framework that functions as a verifiable surrogate for fine-tuning. By crystallizing ephemeral ICL into locally stored symbolic assets, the system iteratively adapts via formal verification feedback to drive a self-healing loop that repairs contracts and skeletal code until strict code-level verification succeeds.
3) **Empirical Evaluation:** An evaluation on the Isolette benchmark [2] (a proxy for avionics subsystems), demonstrating a 58× throughput increase and 100% verification success compared to a failing baseline.

The remainder of this paper is organized as follows. Section II details the System Architecture and the integration with INSPECTA. Section III presents the Meta-Rules methodology. Section IV describes the Isolette case study and experimental results. Section V discusses limitations and future work, and Section VI concludes the paper.

```
# TODO: to add background on sysmlv2/Gumbo contracts
    , related work sec, and update bib for ref
```

Listing 1. background and related work

II. System Architecture

The SCP Codex Copilot introduces a multi-agent "Copilot Loop" that integrates Large Language Models (LLMs) with the INSPECTA toolchain.

A. Neuro-Symbolic Workflow

The architecture operates as a coordinated multi-agent loop. As shown in Fig. 1, the system parses English requirements, applies meta-rules for formalization, and generates model contracts. It then validates these contracts using the *SysML v2 Sireum CLI* and *HAMR/Logika* tools. If verification fails, a "Judge Agent" uses minimal human hints to repair the specification before generating the final verified system code [3].

B. Meta-Rules Integration

To overcome the limitations of pre-trained models, the platform uses "Meta-Rules"—transferable formalization rules learned through a supervised process.

Instead of relying on messy fine-tuning of a huge model, the SCP platform uses a smart "teacher" agent to create these clear, reusable rules. As detailed in Algorithm 1, this

supervised self-adaptation loop watches the main AI, spots patterns where it struggles (e.g., ”hold previous value”), writes a generic mini-template (the meta-rule), and tests it before adding it to the rulebook.

Listing 2 illustrates a specific resulting rule for the ”State” section. This rule directs the agent to introduce persistent variables when requirements imply memory, such as hysteresis or timeouts.

```
/* * 2) WHEN TO USE EACH SECTION
 * 2.1 state
 * Add if the English requires memory:
 * - Hysteresis "shall not be changed" / "hold
     previous"
 * (e.g., Heat within desired band; Alarm no-change
     band).
 * - Timeouts (duration in mode > 1.0 s) if not
     directly
 * available from the platform as a primitive.
 */

// Pattern examples provided to the Agent:
state
  lastHeat: Isolette_Data_Model::On_Off;
  initElapsed: Base_Types::S64; // time units
```

Listing 2. Extract from Gumbo FSE Agent Plan: Meta-Rules for State Formalization.

C. Learning Meta-Rules from Examples

It is crucial to distinguish between the two distinct stages of the Meta-Rules approach. For the end-user, Stage 1—applying existing meta-rules—is straightforward and transparent; the copilot simply uses the established plan. However, Stage 2—learning *new* rules—is a semi-automatic process that requires a supervised feedback loop and careful tuning of learning parameters by developers.

We have developed a technique to automate the creation of these rules, visualized in Fig. 2. It works by utilizing pairs of English requirements and their corresponding correct GUMBO formalization. The system analyzes these pairs to find common patterns and then generates a generalized ’meta-rule’. This rule can then be applied to new, unseen requirements.

The procedural steps for this semi-automatic learning process are outlined in Algorithm 1.

III. Demo Key Moments

The demonstration utilized the ”Isolette System” to benchmark the platform’s capabilities against a direct formalization baseline.

A. Moment 1: The Baseline Failure

In the initial evaluation, a ”direct formalization baseline” using Codex GPT-5.1 Max was tasked with translating requirements into GUMBO contracts. As detailed in the failure log (Listing 3), the model failed to encode FAA requirements related to hysteresis and state variables.

The listing above illustrates the core failure mode. The model is running out of memory and context because it tried every syntactic combination it knew and failed. It is attempting to base its formalization on AADL concepts (like ‘@pre‘), but it does not know how to translate those into the target SysML v2 GUMBO syntax. It lacks the self-confidence and knowledge to formalize the requirements correctly on its own.

```
- Baseline - Codex worked for 33m with virtually no
    success

I started encoding the remaining FAA requirements (
    hysteresis, "no change", etc.) directly in the
    SysML GUMBO clauses. To do so I tried to
    introduce state variables such as last_alarm_cmd
     and last_heat_cmd so that we could talk about
    previous actuator commands (which is necessary
    for REQ-MA-3 and REQ-MHS-4). However the SysML
    parser rejects every variant of the state ...
    syntax I've tried, and it also treats @pre as an
     unsupported "classification test" operator. In
    other words, the current toolchain doesn't
    appear to accept any of the GUMBO constructs
    that refer to component state or pre-state
    values, which makes it impossible to capture the
     remaining requirements faithfully.

Could you share an example (or enable access to the
    original AADL GUMBO subclause) that shows the
    specific syntax this environment expects for
    declaring state variables and referencing their
    pre/post values? Once I know the accepted form I
     can finish adding the REQ-MA-3/5 and REQ-MHS-4
    cases.

Token usage: total=1,067,603 input=981,770 (+
    18,673,920 cached) output=85,833 (reasoning
    54,080)
To continue this session, run codex resume 019ab4a9-
    d677-7631-9b23-aeda551e9513
```

Listing 3. Baseline Failure Log: Transcript from the failed session showing parser rejection and massive token usage.

Therefore, the solution is not vastly more compute, but smarter teaching. We must add reusable, transferable teaching artifacts. This approach is what we call ”Meta-Rules.”

B. Moment 2: Success via Meta-Rules

Upon applying the Meta-Rules (as shown in Listing 2), the SCP Copilot successfully formalized all 40+ requirements. The listing below (Listing 4) displays the copilot’s summary report after this successful run.

Listing 4 demonstrates that the issues identified in the failure log have been completely resolved. The summary confirms that the copilot has now ”remember[ed] the previous actuation for hysteresis” and correctly encoded the ”Manage Regulator Mode contract with state,” proving that the Meta-Rules successfully taught the agent the correct syntax for state and memory. Consequently, the verification tools (HAMR/Logika) ran successfully, achieving 100% proof coverage for both the Monitor and Regulate threads [3].

C. Algorithm

Algorithm 1 presents Meta-Rule learning, application, and bounded repair with human supervision.

1) Algorithm Description: Neuro-Symbolic Meta-Rule Learning and Application: Algorithm 1 formalizes the SCP Codex workflow as a dual-phase neuro-symbolic process designed to ensure the generation of trustworthy, formally

```
Codex optimized with meta-Rules. Worked for ~25
    minutes with complete success

- Regulate.sysml:5-9 now imports Base_Types so the
    new GUMBO blocks can reference typed integer
    literals...
- Regulate.sysml:150-241 embeds the Manage Regulator
     Interface contract...
- Regulate.sysml:281-353 adds the Manage Regulator
    Mode contract with state for the last mode and
    INIT timer plus helper predicates that encode
    the Regulator Status definition from Table A-10;
     cases REQ_MRM_1-REQ_MRM_4 reproduce the
    diagrammed transitions and latch semantics.
- Regulate.sysml:394-476 introduces the Manage Heat
    Source contract: Integration assumes the desired
    -range/temperature bounds, initialization forces
     the heat off, state remembers the previous
    actuation for hysteresis, and compute cases
    cover REQ_MHS_1-REQ_MHS_5 plus the explicit 6s
    latency references...

Verification
- ./sireum hamr sysml logika --sourcepath isolette/
    sysml
- isolette/hamr/slang/bin/run-logika.cmd
Both steps completed successfully (only the
    customary "Ignoring ... connection" notices);
    Logika proved every initialise and timeTriggered
     method for Monitor and Regulate threads.

Token usage: total=369,318 input=317,485 (+
    4,302,976 cached) output=51,833 (reasoning
    36,032)
```

Listing 4. Success Log: Transcript showing successful formalization and verification after applying Meta-Rules.

verified system code. The algorithm distinguishes between the acquisition of transferable knowledge (Phase I) and its application to production tasks (Phase II), integrating formal verification tools directly into the learning loop to guarantee correctness.

Phase I: Meta-Rule Extraction and Validation. The first phase focuses on the supervised acquisition of "Meta-Rules"—abstracted templates that map natural language patterns to formal GUMBO constructs. The process iterates over a set of "golden examples" (G), which consist of paired English requirements and their ground-truth GUMBO formalizations.

- *Extraction:* For each golden pair, the system derives a candidate meta-rule (m) via the EXTRACTRULE function.
- *Generalization Check:* To prevent overfitting, the candidate rule is tested against a sample subset of the target requirements (R). The system generates a candidate formalization ($\hat{g}$) and first applies a cosine distance check (using threshold ϵ) to ensure structural conformity.
- *Verification-Guided Repair:* The candidate formalization is submitted to the verifier (V), utilizing the HAMR/Logika toolchain. If verification fails, the system enters a bounded repair loop constrained by budget (B).
- *Acceptance:* Only rules that successfully verify across the sample set—meeting the acceptance threshold (τ)—are committed to the persistent Meta-Rules library (M).

Phase II: Rule-Guided Formalization. The second phase

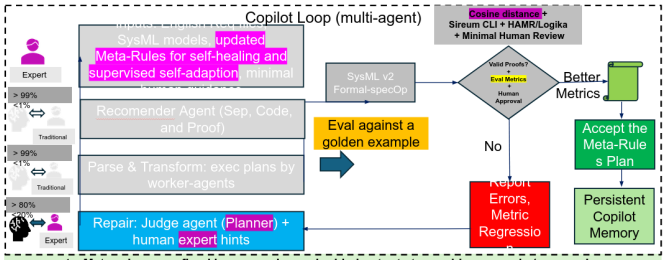

Fig. 2. The semi-automatic supervised learning loop used to generate transferable Meta-Rules from requirement/GUMBO pairs.

represents the production deployment. Instead of relying on the stochastic generation of a raw LLM, the system utilizes the validated library (M) to synthesize formal contracts ($\hat{g}$).

- *Synthesis:* The SYNTHESIZEWITHRULES function applies deterministic patterns, reducing hallucination.
- *Self-Healing:* A similar verification-guided repair loop is employed to resolve semantic errors.
- *Outcome:* Specifications that pass verification are added to set (C), while stubborn failures are isolated in set (F) for expert human review.

2) Discussion: Human-in-the-Loop Refinement: While the Meta-Rule extraction process is highly automated, the "Human Expert" remains essential to ensuring the robust transferability of knowledge. The learning process begins with high-quality inputs; beyond standard requirement text, human experts provide domain-specific documentation—such as the Steve Miller illustrations []—which serve as rich, semantic anchors.

A critical aspect of the framework is the **Generalization-Specialization Trade-off**, managed through a hybrid of mechanical metrics and human review:

- **Semantic Proximity:** The system employs a cosine distance metric to evaluate the semantic similarity between the generated contract and the "Golden" reference. Counter-intuitively, a distance of zero may signal overfitting (the rule is too specific). In such cases, the expert generalizes the rule structure to capture intent rather than instance.
- **Human-Guided Refinement:** If a candidate rule is too general, it risks hallucinations on corner cases; the expert provides specific examples to constrain it. Conversely, if a rule is overly rigid, the expert modifies the template to broaden applicability.

IV. EVALUATION AND METRICS

The quantitative impact of the Meta-Rules is visualized in Fig. 3.

```
# TODO: to add metrics definitions subsection.
```

Listing 5. Meta-Rules

A. Performance Comparison

As illustrated in Fig. 3, the orange bars represent the failing baseline, while the blue bars represent the Fine-Tuned Meta-Rules approach.

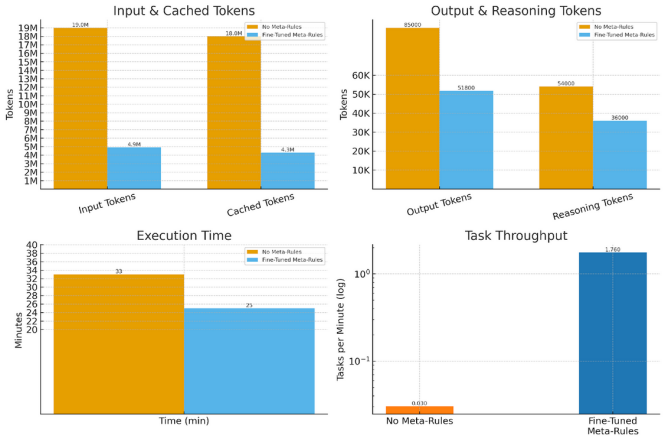

Fig. 3. Performance comparison: Before (Orange) vs. After Meta-Rules (Blue). The charts demonstrate a massive reduction in input/output tokens and execution time, while Task Throughput increases by orders of magnitude.

- **Token Reduction:** The baseline (Orange) consumed ≈19M input tokens. After applying Meta-Rules (Blue), input tokens dropped to ≈4.9M (cached) and only ≈200k uncashed.
- **Reasoning Efficiency:** Output and reasoning tokens saw significant reductions, dropping from $>$ 80k to ≈51k output tokens.
- **Execution Time:** The total time was cut from 33 minutes to ≈25 minutes, transforming a failing workflow into a fully verified one.
- **Correctness:** The approach achieved a $> 58\times$ improvement in correctness, moving from near-zero to 100% verification success [3].

```
# TODO: Limitations and future work section.
```

Listing 6. Meta-Rules

V. Conclusion

The SCP Codex Open Platform demonstrates that using Meta-Rules to extract and reuse formalization knowledge significantly outperforms direct model querying. This neuro-symbolic approach enables the reliable generation of trustworthy, formally verified system code with orders-of-magnitude improvements in efficiency and cost.

References

[1] D. Hardin, I. Amundson, J. Babar, D. Cofer, S. Hasan, K. Hoech, J. Belt, J. Hatcliff, Robby, and S. Hallerstede, "Automated sysml v2 system model to memory-safe language code generation for avionics applications," in *Proceedings of the IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2025, author version / preprint (uploaded as hardin2025dasc.pdf).

[2] J. Hatcliff *et al.*, "The isolette system: Illustrating end-to-end artifacts for rigorous model-based engineering," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2024, author version (uploaded as Hatcliff-etal-ISOLA2024-Isolette-Overview.pdf).

[3] A. Tahat *et al.*, "Neurosymbolic automated contract and code generation for sysml v2 models," in *Collins Aerospace*, 2026, author version / preprint (uploaded as slides.pdf).

Appendix

```
# TODO: Paste complete rules file here.
```

Listing 7. Meta-Rules

Insert a link to the complete Meta-Rules file here.

Algorithm 1: SCP Meta-Rules Learning, Application, and Verification-Guided Repair

Input: Requirements set R; golden examples G (English + normalized GUMBO); verifier V (HAMR/Logika); budgets B (time/tokens/repair cycles); similarity threshold ϵ; acceptance threshold τ.

Output: Verified contracts C; retained Meta-Rules library M; flagged failures F.

Initialization:

$M \leftarrow \emptyset$ `// persistent rule memory`

$C \leftarrow \emptyset$; $F \leftarrow \emptyset$

Phase I: Meta-Rule extraction and validation

foreach $(r_g, g_g) \in G$ **do**
- $m \leftarrow \textsc{ExtractRule}(r_g, g_g)$
- $okCount \leftarrow 0$
- **foreach** $r \in \textsc{Sample}(R)$ **do**
  - $\hat{g} \leftarrow \textsc{ApplyRule}(m, r)$
  - **if** $\textsc{Distance}(\hat{g}, g_g) > \epsilon$ **then**
    - **continue**
  - $(ok, fb) \leftarrow V(\hat{g}, B)$
  - **while** not ok ***and*** $\textsc{RepairBudgetRemaining}(B)$ **do**
    - $\hat{g} \leftarrow \textsc{Repair}(\hat{g}, fb, B)$
    - $(ok, fb) \leftarrow V(\hat{g}, B)$
  - **if** ok **then**
    - $okCount \leftarrow okCount + 1$
- **if** $okCount \geq \tau$ **then**
  - $M \leftarrow M \cup \{m\}$ `// retain only if it verifies and generalizes`
- **else**
  - $\textsc{DiscardOrRefineWithHuman}(m)$

Phase II: Rule-guided formalization with bounded repair

foreach $r \in R$ **do**
- $\hat{g} \leftarrow \textsc{SynthesizeWithRules}(r, M)$
- $(ok, fb) \leftarrow V(\hat{g}, B)$
- **while** not ok ***and*** $\textsc{RepairBudgetRemaining}(B)$ **do**
  - $\hat{g} \leftarrow \textsc{Repair}(\hat{g}, fb, B)$
  - $(ok, fb) \leftarrow V(\hat{g}, B)$
- **if** ok **then**
  - $C \leftarrow C \cup \{(r, \hat{g})\}$
- **else**
  - $F \leftarrow F \cup \{(r, \hat{g}, fb)\}$

Quality assessment and logging:

Log timestamps, token usage, latency, and repair count; store M for reuse.