Bridging Natural Language to Verified Implementation: A Neuro-Symbolic High-Assurance MBSE Pipeline in SysML v2

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

***Abstract*—The development of high-confidence avionics systems demands rigorous allocation and end-to-end traceability from requirements to implementation, yet current Model-Based Systems Engineering (MBSE) environments provide little support for automated, mathematically verified allocation and traceability. While Generative AI (GenAI) offers a potential automation boon, its deployment in safety-critical workflows is severely constrained by hallucinations, prohibitive token costs, scarcity of open-source domain data, and restrictions on fine-tuning state-of-the-art proprietary models such as GPT-5.**

To address these barriers, we introduce a neuro-symbolic Spec–Code–Proof (SCP) copilot that integrates Codex-class agents into the DARPA PROVERS INSPECTA pipeline. The pipeline translates natural-language requirements into SysML v2 models with GUMBO contracts, generates HAMR code, and discharges Logika proofs, ensuring properties are mathematically preserved in the implementation. To enable this without fine-tuning, we introduce a *Meta-Rule* methodology that functions as a verifiable surrogate for weight adaptation. By crystallizing ephemeral in-context learning into persistent, expert-curated abstraction patterns, Meta-Rules drive a self-healing generate–verify–repair loop until verification constraints are formally satisfied.

We frame our work as a specialized paradigm called Formal Specification-Driven Development (FSDD). FSDD elevates formal specifications—contracts, verification conditions, and machine-checkable proofs—to first-class artifacts that drive generation, verification, and repair. Consequently, FSDD establishes formal proofs, rather than tests, as the primary acceptance gate.

We evaluate our methodology on safety-critical systems using an example sourced from an FAA requirements guidebook (>37 pages) relevant to avionics and cyber-physical domains. Our copilot achieved 100% code-level verification and a 58× throughput increase compared to a baseline that, lacking Meta-Rules, failed to produce verifiable artifacts. This demonstrates a practical path to certification-grade automation under strict data and model governance constraints, reconciling the efficiency of Generative AI with the strict rigor required for certifiable critical system software.

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

I. Introduction

High-assurance avionics software development faces a persistent tension: system complexity continues to increase, while certification demands strong evidence that implementation faithfully realizes requirements and architectural intent. In practice, maintaining end-to-end traceability from natural-language requirements to architectural models and down to code remains costly and fragile. In many Model-Based Systems Engineering (MBSE) workflows, requirements allocation and refinement still involve manual, tool-fragmented steps that weaken (or sever) the link between the model-level assurance case and the software implementation. The resulting drift is often detected late—during integration or verification—driving rework, schedule risk, and certification friction [1].

Large Language Models (LLMs) appear well-suited to assist with this transition by translating requirements into structured specifications, contracts, and implementation scaffolding [2]. However, directly deploying LLMs in safety-critical workflows faces structural barriers. Beyond hallucinations and token-heavy prompting, aerospace environments face the *fine-tuning bottleneck*: domain-representative data is often scarce or proprietary, and fine-tuning frontier proprietary models (e.g., GPT-5-class systems [3]) is typically restricted by vendor policy or cost [2], [4].

As a result, these restrictions leave three capabilities—highly desirable in high-assurance certification-oriented development pipelines—unmet: (i) reducing token-heavy, brittle prompting; (ii) filtering or rejecting hallucinated behaviors with machine-checkable acceptance gates; and (iii) enabling non-formalists to work from English while still producing reusable, auditable evidence that persists across runs.

To address these barriers, we introduce a neuro-symbolic **Spec–Code–Proof (SCP)** *copilot*—a generate–verify–repair pipeline that turns natural-language requirements into traceable formal artifacts and verified implementations. SCP targets the three gaps above by (i) shifting effort from repeated token-heavy prompting into reusable intermediate artifacts; (ii) using formal verification as the primary acceptance gate to catch hallucinated or underspecified behavior; and (iii) providing a low-entry English interface while accumulating governed, version-controlled specialization knowledge. Our approach integrates Codex-class software engineering agents into the DARPA PROVERS **INSPECTA** (Industrial-Scale Proof Engineering for Critical Trustworthy Applications) toolchain [1]. INSPECTA provides the formal backbone: **SysML v2** [5], [6] captures architecture, **GUMBO** [7], [8] expresses behavioral assume/guarantee contracts, **HAMR** [9], [10] generates executable memory-safe code, and **Logika** [10] discharges proof obligations using SMT solvers (e.g., Z3 [11] and CVC5 [12]). The intended outcome is an auditable workflow where the

toolchain mechanically preserves properties established at the model level in the generated code. However, while this infrastructure ensures correctness *after* specification, the initial creation of formal models and contracts remains a manual, expertise-heavy bottleneck. Ideally, an agentic copilot would assist engineers in synthesizing these rigorous specifications from natural language, yet realizing this capability faces a critical barrier: effectively *specializing* an LLM to such a complex, verification-centric toolchain under base-model fine-tuning restrictions.

Recent mechanistic interpretability results suggest that In-Context Learning (ICL) can act like an "implicit fine-tuning" process during inference [13], but the effect is transient: once the context window is cleared, the learned behavior is lost. In high-assurance settings, this ephemerality is a practical obstacle: it forces repeated token-heavy prompting and makes it difficult to govern, audit, or reuse what the model "learned" across runs.

To stabilize this transient learning into a permanent asset, we introduce *Meta-Rules*: persistent, human-readable formalization abstractions that *crystallize* successful ICL interactions (derived from golden examples, tool feedback, and expert edits) into a reusable rule library. Meta-Rules serve as a verifiable surrogate for black-box weight adaptation: instead of modifying parameters, the system externalizes adaptation into version-controlled artifacts that are (i) inspectable by humans, (ii) exercised by the toolchain, and (iii) constrained by formal verification. In developer mode, the system iteratively proposes and refines Meta-Rules using semantic similarity metrics and HAMR/Logika feedback; only candidates that satisfy quantitative improvement criteria, pass verification, and receive human approval are retained. In user mode, the copilot injects this curated library into the context, effectively triggering the underlying implicit fine-tuning mechanism [13] to behave as a domain-adapted agent. This allows us to bypass the aforementioned fine-tuning restrictions while achieving comparable specialization, driving a generate–verify–repair loop until model integration and code-level verification succeed.

In addition to consistently producing practically formally verifiable output at low cost (Section VI)—effectively lowering the entry barrier for users—this method systematically accumulates a dataset of verified repair trajectories, alleviating current data scarcity and paving the way for future fine-tuning as model accessibility improves.

By anchoring the development workflow in these governed, externalized artifacts rather than opaque model behaviors, our approach aligns with the industry's emerging *Specification-Driven Development (SDD)* practices. Frameworks such as GitHub's Spec Kit and AWS Kiro [14] aim to curb "vibe coding" by structuring AI-assisted development around natural-language specifications and acceptance criteria rather than unconstrained prompt-driven generation. We extend this paradigm to *Formal Specification-Driven Development (FSDD)* by making *formal verification* the primary acceptance gate. In our workflow, specifications are refined into SysML v2 models and GUMBO contracts, then discharged through HAMR/Logika verification. This yields machine-checkable evidence alongside generated code, enabling a low-entry workflow that remains aligned with certification demands for traceability, repeatability, and auditable assurance artifacts.

In summary, the main contributions of this paper are:

1) **Trustworthy English-to-Implementation Pipeline:** A multi-agent SCP generate–verify–repair loop workflow integrated with INSPECTA (SysML v2 + GUMBO + HAMR/Logika) that generates verified contracts and code from requirements.
2) **Supervised Meta-Rules Adaptation:** A verifiable surrogate for fine-tuning that crystallizes ephemeral ICL behaviors into persistent Meta-Rules, enabling a governed generate–verify–repair loop driven by formal verification feedback and human approval.
3) **Empirical Evaluation:** An evaluation on the Isolette benchmark [15] (a proxy for avionics subsystems), demonstrating a substantial throughput increase and full verification success compared to a failing baseline.

The remainder of this paper is organized as follows. Section II summarizes SysML v2 GUMBO contracts and the HAMR/Logika verification workflow. Section III presents the SCP neuro-symbolic architecture and the Meta-Rules methodology. Section IV compares Meta-Rules to fine-tuning and transient prompting, and discusses trade-offs. Section V presents the Isolette running example. Section VI details the experimental evaluation and metrics. Section VII discusses related work. Section VIII discusses current limitations with future research directions, and Section IX concludes.

To foster community collaboration and accelerate future research, all toolchain source code, along with our curated dataset of successful repair trajectories, is released as open source.[1]

II. GUMBO Contracts in SysML v2 and HAMR/Logika codegen and verification

TODO: Reserved for a brief SysML v2 GUMBO tutorial.

GUMBO contracts express assumptions and guarantees for SysML v2 components, supporting compositional reasoning at the model level and refinement to implementation-level specifications. A typical GUMBO block includes optional state variables, initialization guarantees, assumptions constraining inputs, global guarantees, and `compute_cases` clauses aligning conditional requirements with case-specific guarantees.

Listing 1 provides an illustrative skeleton.

```
language "GUMBO" /*{
  state lastCmd: On_Off;
  initialize
    guarantee initLastCmd: lastCmd == Off;

  compute
    assume lowerLEUpper: lower <= upper;
    guarantee lastCmdTracksOutput: lastCmd ==
        heat_control;
```

[1] https://github.com/loonwerks/INSPECTA-Spec-Code-Proof-Copilot

```
  compute_cases
    case REQ_1:
      assume mode == INIT;
      guarantee heat_control == Off;
    case REQ_2:
      assume mode == NORMAL && temp < lower;
      guarantee heat_control == On;
} */
```

Listing 1. Illustrative skeleton of a SysML v2 component with a GUMBO contract.

III. Neuro-Symbolic Contract Generation with Meta-Rules

A. SCP Copilot Overview

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

B. SCP Copilot 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 [16].

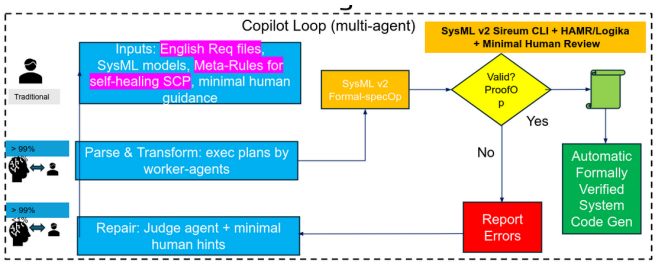

Fig. 1. SCP copilot neuro-symbolic generate–verify–repair loop. It parses English requirements, applies Meta-Rules, and utilizes the INSPECTA toolchain for verification.

C. Meta-Rules: Persistent, Auditable Formalization Abstractions

Meta-Rules are persistent, human-readable abstractions that constrain how the copilot translates natural-language requirements and architectural context into GUMBO contracts. Each Meta-Rule captures a reusable mapping from recurring requirement patterns (e.g., hysteresis, "no change" semantics, timeouts, latched modes) to concrete, tool-compatible GUMBO constructs (e.g., state variables, initialization guarantees, and structured `compute_cases`). In contrast to transient prompt-only conditioning, Meta-Rules are stored locally as version-controlled artifacts (a "rule book") that can be reviewed, curated, and audited by domain experts.

Operationally, Meta-Rules function as a verifiable surrogate for weight adaptation: the copilot's behavior is shaped by persistent symbolic assets rather than by modifying model parameters. This provides (i) inspectability for human stakeholders, (ii) repeatability across runs, and (iii) compatibility with certification-oriented governance practices.

1) Two Modes of Use: User Mode vs. Developer Mode: We distinguish two operational modes:

User Mode (application). End users trigger the copilot with a low entry-point prompt that applies an existing, curated Meta-Rule library to translate requirements into candidate GUMBO contracts. The workflow then executes HAMR/Logika verification; failures trigger bounded, tool-guided repair until integration-level and code-level verification succeed.

Developer Mode (adaptation). Developers iteratively refine the Meta-Rule library itself. Adaptation is supervised and gated: candidate Meta-Rule modifications are treated as hypotheses and retained only if they improve measurable criteria, pass verification, and receive explicit human approval. This yields cumulative improvement without fine-tuning.

2) Example Meta-Rule: Listing 2 shows an excerpt from the rule book used in our evaluation. The rule instructs the agent to introduce explicit state when the requirements imply memory (e.g., hysteresis bands, "hold previous" semantics, time-dependent predicates). Such rules were motivated by baseline failures in which the agent attempted incompatible idioms (e.g., AADL pre-state operators) and could not consistently realize SysML v2 GUMBO state syntax.

```
/* * 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.

D. Supervised Meta-Rule Adaptation and Acceptance Criteria

Meta-Rule adaptation is driven by three complementary gates:

(1) Semantic improvement. When golden reference formalizations are available (e.g., from validated contract blocks), we compute embedding-based cosine distance between the candidate contract text and the golden reference. Candidate Meta-Rule changes must demonstrate monotonic improvement against this semantic metric over a validation subset.

(2) Formal verification. All retained candidates must pass the INSPECTA verification workflow: SysML integration-level Logika verification followed by code-level Logika verification of HAMR-generated artifacts.

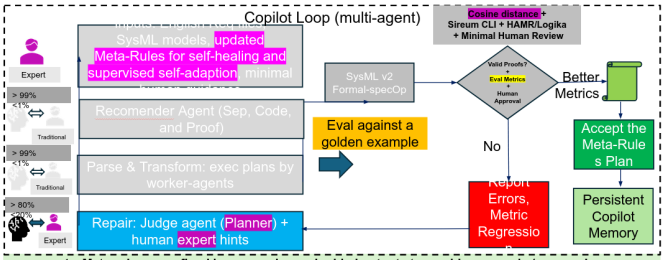

Fig. 2. Semi-automatic supervised loop used to learn and refine transferable Meta-Rules from requirement/GUMBO pairs under formal verification and human governance.

(3) Human governance. A human reviewer approves any Meta-Rule change before it is persisted into the rule book, ensuring the retained rule remains interpretable and transferable.

E. Practical Considerations

Meta-Rules are stored as local, version-controlled artifacts (a rule book) and are supplied to the agent at execution time together with the verification plan. This design keeps adaptation auditable and allows changes to be reviewed, reproduced, and rolled back. We defer discussion of limitations and trade-offs to Section VIII.

F. Algorithm: Meta-Rule Learning, Application, and Verification-Guided Repair

Algorithm 1 formalizes the SCP workflow as a dual-phase neuro-symbolic process.

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).

- *Extraction:* For each golden pair, the system derives a candidate Meta-Rule (m) via EXTRACTRULE.
- *Generalization Check:* To reduce over-specialization, the candidate rule is tested against a sample subset of target requirements (R). A candidate formalization ($\hat{g}$) is generated and evaluated with a similarity check using threshold ϵ.
- *Verification-Guided Repair:* The candidate formalization is submitted to the verifier (V).
- *Acceptance:* Only rules that verify across the validation subset—meeting acceptance threshold (τ) and human approval—are committed to the persistent Meta-Rule library (M).

Phase II: Rule-Guided Formalization. The second phase represents production deployment. Instead of relying on unconstrained generation, the system utilizes the validated library (M) to synthesize formal contracts ($\hat{g}$) and iteratively repairs them until verification succeeds or a budget is exhausted.

G. Discussion: Human-in-the-Loop Refinement

While Meta-Rule extraction and validation are highly automated, human oversight remains essential to ensure transferability and prevent drift. A critical aspect is the

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**

  $m \leftarrow \text{EXTRACTRULE}(r_g, g_g)$

  $okCount \leftarrow 0$

  foreach $r \in \text{SAMPLE}(R)$ **do**

    $\hat{g} \leftarrow \text{APPLYRULE}(m, r)$

    if $\text{DISTANCE}(\hat{g}, g_g) > \epsilon$ **then**

      continue

    $(ok, fb) \leftarrow V(\hat{g}, B)$

    while not ok ***and*** $\text{REPAIRBUDGETREMAINING}(B)$ **do**

      $\hat{g} \leftarrow \text{REPAIR}(\hat{g}, fb, B)$

      $(ok, fb) \leftarrow V(\hat{g}, B)$

    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

    $\text{DISCARDORREFINEWITHHUMAN}(m)$

Phase II: Rule-guided formalization with bounded repair

foreach $r \in R$ **do**

  $\hat{g} \leftarrow \text{SYNTHESIZEWITHRULES}(r, M)$

  $(ok, fb) \leftarrow V(\hat{g}, B)$

  while not ok ***and*** $\text{REPAIRBUDGETREMAINING}(B)$ **do**

    $\hat{g} \leftarrow \text{REPAIR}(\hat{g}, fb, B)$

    $(ok, fb) \leftarrow V(\hat{g}, B)$

  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.

generalization–specialization trade-off, managed through semantic proximity checks, verification gates, and human gov-

ernance.

IV. Comparison to Fine-Tuning and Transient Prompting

This section situates Supervised Meta-Rules Adaptation relative to two common approaches for specializing LLMs: parameter fine-tuning and transient prompting / in-context learning (ICL). The comparison emphasizes suitability for high-assurance, certification-oriented workflows where auditability, governance, and reproducibility are first-class requirements.

A. Parameter Fine-Tuning

Fine-tuning modifies model weights to internalize domain-specific behavior. While it can yield strong task performance, it imposes significant practical barriers in aerospace settings: training data are often sensitive or scarce, compute costs can be prohibitive, and distributing tuned checkpoints may be restricted for proprietary frontier models. From an assurance perspective, fine-tuning produces opaque behavioral changes that are difficult to inspect, justify, or certify at the granularity expected in safety-critical development.

B. Transient Prompting and In-Context Learning

ICL adapts behavior via examples and instructions supplied in the prompt window. Mechanistic interpretability work suggests ICL can behave like "implicit fine-tuning" during inference [13], but its effects are ephemeral: the learned behavior disappears when context is reset. In practice, this ephemerality drives repeated long-context prompting, higher token cost, and limited governance over what was learned across sessions.

C. Supervised Meta-Rules Adaptation

Meta-Rules externalize learned formalization behavior into persistent, version-controlled artifacts. Rather than adapting weights, the system crystallizes successful ICL interactions into reusable abstraction patterns that are (i) inspectable by humans, (ii) constrained by formal verification gates (HAMR/Logika), and (iii) retained only under explicit human approval. This transforms adaptation into an auditable engineering process and provides a practical surrogate for fine-tuning under data, cost, and governance constraints.

Table I summarizes the adaptation mechanisms discussed in this section.

V. A Running Example: The Isolette System

We use the Isolette Infant Incubator benchmark as a proxy for safety-critical subsystems [15]. The goal is to translate natural-language requirements into SysML v2 GUMBO contracts and then discharge both integration-level and code-level verification obligations using the SCP Copilot Self-healing mechanism with minimal user intervention.

A. Experimental Inputs and Outputs

Inputs. The experiment uses (i) natural-language requirements provided in a realistic FAA-style source document (37+ pages) containing plain text, tables, and figures (e.g., `Steve_Miller_FAA_docAR-08-32.pdf`), and (ii) SysML v2 models and shared libraries representing the Isolette architecture (e.g., `Monitor.sysml`, `Regulate.sysml`, plus SysML↔AADL shared libraries). These artifacts intentionally include realistic friction: domain terminology, cross-references, and requirements that imply memory (hysteresis/latched state) that must be encoded using SysML v2 GUMBO constructs accepted by the toolchain.

Outputs. The pipeline produces (i) SysML v2 models augmented with GUMBO contracts (including state/initialize/compute_cases blocks), (ii) HAMR-generated implementation artifacts, and (iii) verification logs demonstrating successful discharge of Logika proof obligations for both model integration and code-level execution.

B. Conditions: Baseline vs. Meta-Rule–Guided SCP

We compare two conditions:

Baseline (prompt-only ICL). A Codex-class agent is tasked to read the requirement document and directly generate correct SysML v2 GUMBO contracts without an external Meta-Rule library. This relies on transient in-context learning and ad-hoc prompt conditioning.

Meta-Rule–guided SCP. The same overall task is executed using a curated Meta-Rule library (Section III). The copilot is triggered with a low entry-point prompt, applies the rule book to guide formalization, and then executes a bounded generate–verify–repair loop driven by HAMR/Logika feedback. Importantly, the Meta-Rules used in this condition are treated as persistent engineering artifacts: they are created/refined in developer mode and retained only if they satisfy the acceptance gates described in Section III (semantic improvement where golden references exist, verification success, and human approval).

C. Moment 1: Baseline Failure (Token Explosion and Tool Mismatch)

In the initial evaluation, the baseline using Codex GPT-5.1 Max failed to encode requirements that implied memory (hysteresis, "no change") in SysML v2 GUMBO. Listing 3 shows the failure transcript: the agent repeatedly attempts incompatible pre-state idioms (e.g., `@pre`) and cannot converge on tool-accepted SysML v2 state syntax. This failure manifests as rapid token growth with no verifiable outcome.

Connection to the comparison. This is a concrete instance of the limitations discussed in Section IV: prompt-only ICL is ephemeral and does not reliably internalize niche toolchain syntax, leading to long-context retries and token explosion. A standard remedy would be domain fine-tuning; however, fine-tuning frontier proprietary models was not available in our setting (vendor restrictions). This motivates a persistent, auditable specialization mechanism that does not require weight updates.

```
- 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.

D. Moment 2: Success via Meta-Rules (Generate–Verify–Repair Under Gates)

Upon applying the curated Meta-Rules (e.g., Listing 2), the SCP copilot successfully formalized the requirements, inserted tool-accepted SysML v2 GUMBO blocks (including state for hysteresis), and completed both integration-level and code-level verification. Listing 4 summarizes the successful run.

Connection to the acceptance gates. The rules used here are not ad-hoc prompts; they are persistent assets curated under the developer-mode gates in Section III. In particular, Meta-Rule updates are retained only when they (i) improve semantic alignment where golden references exist (cosine-distance guidance), (ii) pass integration and code-level verification (hard gate), and (iii) receive human approval prior to being committed to the rule book. The user-mode run then applies these retained rules to reduce search and prevent token-heavy trial-and-error.

```
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.

E. Cross-Model Robustness (GPT-5.1 → GPT-5.2)

In follow-on runs, the same Meta-Rule library authored and validated under Codex GPT-5.1 Max executed unchanged on Codex GPT-5.2 Max while preserving verification outcomes. We observed reduced reasoning-token consumption for comparable tasks, indicating that the approach benefits directly from improvements in newer models without requiring fine-tuning or re-authoring rules. This supports the claim that Meta-Rules provide a forward-compatible, locally governed specialization layer that can track model evolution while keeping adaptation auditable and low-risk.

VI. Evaluation and Metrics

A. Experimental Settings

We evaluated SCP on the Isolette benchmark under two conditions: (i) a baseline prompt-only Codex agent and (ii) a

TABLE I
QUALITATIVE COMPARISON OF ADAPTATION MECHANISMS FOR HIGH-ASSURANCE PIPELINES.

	Fine-tuning	Prompting/ICL	Meta-Rules
Persistent across runs	Yes	No	Yes (external)
Requires weight updates	Yes	No	No
Auditability / review	Low	Low–Med	High
Governance / rollback	Low	Low	High
Runtime context overhead	Low	High	Med
Data/IP constraints	High	Low	Low
Verification integration	Indirect	Indirect	Direct

Meta-Rule–guided SCP copilot. In both conditions, the agent was provided a realistic natural-language requirement source document (37+ pages, containing figures, tables, and dense prose) along with the SysML v2 architecture models and shared libraries. The Meta-Rule–guided condition additionally supplied the curated Meta-Rule library and verification plan artifacts described in Section III. Success was defined as completion of integration-level and code-level verification using the INSPECTA workflow, culminating in a verified set of GUMBO-augmented SysML v2 models and verified HAMR-generated code.

B. Metrics Definitions

We report performance using token accounting and wall-clock time. Each run produces a structured usage record of the form: *total tokens, input tokens (uncached), cached input tokens, output tokens, reasoning tokens, and wall-clock runtime.* We define:

- **Wall-clock time (T_{wall}):** elapsed runtime from the start of the copilot run until termination (success or failure), measured in minutes.
- **Uncached input tokens (N_{in}):** tokens in the prompt/-context that are processed as new input for the run.
- **Cached input tokens (N_{cache}):** tokens reused from prior context caching (reported separately because cached tokens contribute to throughput and cost differently than uncached input).
- **Output tokens (N_{out}):** total tokens generated by the model in the response stream.
- **Reasoning tokens (N_{reason}):** internal reasoning tokens reported by the model (a subset of model computation correlated with cost and latency). We report these explicitly because reductions in N_{reason} often translate directly to lower runtime and lower cost in practice. They are typically priced higher than the other tokens.
- **Total tokens (N_{total}):** aggregate token volume for the run:
$$N_{\text{total}} = (N_{\text{in}} + N_{\text{cache}}) + N_{\text{out}}.$$
- **Task throughput (Θ):** verified task completion rate. In this study, we report throughput as "successful verified run per minute" (binary success over T_{wall}), and we also interpret qualitative throughput improvements using token efficiency (verified success achieved with lower N_{total}).

C. Performance Comparison

The quantitative impact of Meta-Rules is visualized in Fig. 3. The orange bars represent the failing baseline, while the blue bars represent the Meta-Rule–guided approach.

As illustrated in Fig. 3:

- **Token efficiency:** The baseline consumed ≈19M cached input tokens and still failed. With Meta-Rules, cached input tokens dropped to ≈4.3M and uncached input dropped to ≈317k, while verification succeeded.
- **Reasoning efficiency:** Reasoning tokens decreased (e.g., from ≈54k to ≈36k in the representative logs), consistent with reduced search and fewer failed retries.

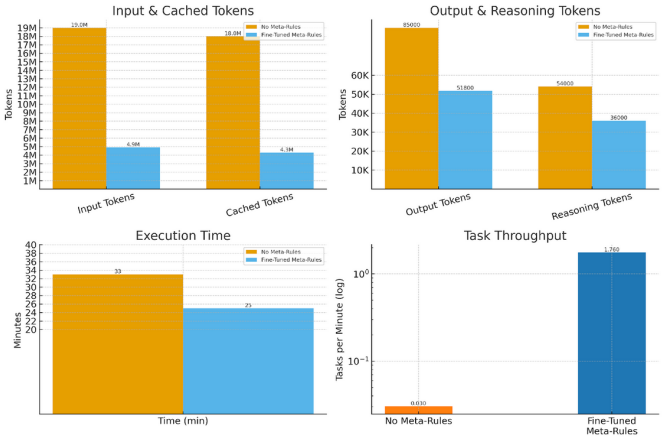

Fig. 3. Performance comparison: Before (Orange) vs. After Meta-Rules (Blue). The charts show a reduction in token usage and wall-clock time, converting a failing run into a fully verified run.

- **Wall-clock time:** Total runtime decreased from 33 minutes (failure) to ≈25 minutes (success).
- **Correctness:** The Meta-Rule–guided approach achieved full integration-level and code-level verification where the baseline failed to produce verifiable artifacts [16].

VII. Related Work

Recently, generative AI and particularly LLMs have shown promising potential to improve explainability and guide automated formal verification. Early efforts in AI-assisted theorem proving demonstrated the viability of integrating neural models with rigorous proof assistants. Notable examples include OpenAI's GPT-f and their subsequent fine-tuning (FT) experiments, which achieved early evidence of success in Metamath theorem proving [17]. Other initiatives have successfully applied LLMs to proof repair, automated tactic generation, and theorem diagnosis across a variety of foundational formal systems, such as Isabelle/HOL [18] and Coq [19]. In the specific context of the Lean theorem prover, recent AI advancements by commercial AI start-up models such as Harmonic Arsitotole and Axiom's Prover have showcased the ability of neuro-symbolic models to solve some complex, open mathematical problems [20] and reach gold medal performance in mathOlympics benchmarks [21].

Beyond pure mathematics, efforts in code verification have also increasingly adopted neuro-symbolic approaches. Microsoft has extensively explored AI-assisted code verification leveraging their F* framework [22]. Similarly, AWS recently introduced an SMT-backed hallucination prevention tool for generative AI [23], offered as a web service, and Apple released GSM-Symbolic to evaluate the limits of LLMs in symbolic reasoning tasks [24].

Within the domain of systems engineering, formal methods are currently being applied to modern MBSE standards. The OMG SysML v2 [25] was designed to support formal semantics, and reasoning platforms like Imandra [26] have developed transpilation tools to verify SysML v2 constraints, supporting an AI-assisted framework at the model level provided as a

commercialized closed-source web service [26]. Additionally, models such as the PA-1 Archie model [27] pave the way to bridge formal methods and physical-world modeling. However, these MBSE applications are fundamentally limited to the model level; they analyze physical world properties and state machines but currently lack mechanisms for end-to-end formally verified code generation. Closer to our architectural focus, Tahat et al. introduced *AGREE-Dog* [28], a neuro-symbolic copilot for the OSATE AADL [29] environment that automates explainable compositional reasoning and repair using the AGREE tool.

Our work proposes a significantly deeper and more comprehensive framework. Rather than stopping at model-level analysis or functioning solely as an isolated proof assistant or specification language, the integration of the INSPECTA pipeline in our SCP copilot provides an end-to-end formal verification toolchain. It bridges the entire gap from English natural-language requirements down to GUMBO-annotated SysML v2 models, and ultimately synthesizes verified application code. By targeting high-assurance deployment platforms like the seL4 microkernel [30] and leveraging verified compilation microkit [31], our approach ensures that the strict mathematical rigor established at the architectural level is mechanically preserved all the way down to the executable implementation.

VIII. Limitations and Future Work

In this section we identify three key directions for future research to address our current limitations and scale the architecture:

A. Expanding Evaluation to Heterogeneous Systems

To validate the generalizability of our Meta-Rules beyond the Isolette benchmark, we intend to expand our evaluation to diverse cyber-physical components. A primary target is the formalization of verified security appliances, such as the seL4-based firewall architectures developed by DornerWorks [32]. Applying our pipeline to these systems will test the adaptability of our GUMBO generation logic to distinct security policies and packet-filtering domains, moving beyond the control-theoretic patterns of the Isolette.

B. Integrating Closed-Loop Repair Heuristics for Rust Application Code

Currently, our copilot was tested on Scala/Slang [33] core application logic. We plan to integrate repair heuristics inspired by VeriStruct, a framework recently developed at Stanford for generating and repairing verified Rust code [34]. By adapting VeriStruct's syntax-guided and verification-aware repair strategies—specifically targeting the Verus toolchain—we aim to extend our SCP self-healing loop to the full synthesis of verified application logic in Rust.

C. Scalability and Physically-Aware Specification

Current LLMs often lack grounding in the physical constraints of cyber-physical systems. Future work will explore generating SysML v2 specifications using multimodal models capable of reasoning about physical world dynamics (e.g., spatial constraints, physics-based failure modes). Additionally, we aim to migrate the current local architecture to a cloud-based infrastructure, using SDD frameworks such as AWS Kiro, to parallelize the verification workload. Finally, the symbolic capability of our pipeline will naturally expand with the maturity of the underlying formal tools; specifically, analogous to [35], [36], we plan to leverage ongoing advancements in GUMBO that support compositional reasoning [9], enabling the verification of larger, hierarchically complex avionics systems.

IX. Conclusion

The SCP Codex Open Platform demonstrates that using Meta-Rules to extract and reuse formalization knowledge can significantly outperform direct, unconstrained model querying in a verification-centric MBSE toolchain. This neuro-symbolic approach enables the generation of formally verified system code with substantial improvements in efficiency and cost while preserving auditability and human governance.

Acknowledgment

This work was funded by DARPA contract FA—. The views, opinions, and findings expressed are those of the authors and do not necessarily reflect the official views or policies of the U.S. Department of War or the U.S. Government.

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] OpenAI, "GPT-5.2 Codex System Card," https://openai.com/index/gpt-5-2-codex-system-card/, 2025, accessed: 2026-02-12.

[3] S. Bubeck, C. Coester, R. Eldan, T. Gowers, Y. T. Lee, A. Lupsasca, M. Sawhney, R. Scherrer, M. Sellke, B. K. Spears, D. Unutmaz, K. Weil, S. Yin, and N. Zhivotovskiy, "Early science acceleration experiments with gpt-5," 2025. [Online]. Available: https://arxiv.org/abs/2511.16072

[4] OpenAI, "GPT-5 System Card," https://openai.com/index/gpt-5-system-card/, 2025, accessed: 2026-02-12.

[5] Object Management Group (OMG), "Sysml v2 model formalism working group," https://www.omgwiki.org/OMGSysML/doku.php?id=sysml-roadmap:sysml_v2_model_formalism_working_group, accessed: 2026-02-10.

[6] Object Management Group, "Omg system modeling language (sysml) v2 specification," 2025. [Online]. Available: https://www.omg.org/spec/SYSML

[7] J. Hatcliff, D. Stewart, J. Belt, Robby, and A. Schwerdfeger, "An AADL contract language supporting integrated model- and code-level verification," *Ada Letters*, vol. 42, no. 2, p. 45–54, April 2023. [Online]. Available: https://doi.org/10.1145/3591335.3591339

[8] Galois, "Gumbo: Grand unified modeling of behavioral operators," 2024. [Online]. Available: https://www.galois.com/project/gumbo

[9] J. Hatcliff, J. Belt, Robby, and T. Carpenter, "HAMR: An AADL multi-platform code generation toolset," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 274–295.

[10] *Sireum HAMR: High Assurance Modeling and Rapid Engineering for Embedded Systems*, Kansas State University, 2025. [Online]. Available: http://hamr.sireum.org/

[11] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08, 2008, pp. 337–340.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd international conference on Computer aided verification*, ser. CAV'11, 2011, pp. 171–177.

[13] J. von Oswald, E. Niklasson, E. Randazzo, J. Sacramento, A. Mordvintsev, A. Zhmoginov, and M. Vladymyrov, "Transformers learn in-context by gradient descent," in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023, pp. 35 151–35 174.

[14] AWS News Blog, "Prevent factual errors from llm hallucinations with mathematically sound automated reasoning checks (preview)," AWS Blog Post, 2024, accessed: 2025-05-11.

[15] 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).

[16] 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).

[17] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *arXiv preprint arXiv:2009.03393*, 2020.

[18] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," *arXiv preprint arXiv:2303.04910*, 2023.

[19] A. Tahat, D. Hardin, A. Petz, and P. Alexander, "Proof repair utilizing large language models: A case study on the copland remote attestation proofbase," in *Proceedings of International Symposium On Leveraging Applications of Formal Methods Verification and Validation (AISolA)*, 2024.

[20] Axiom, "Territory," https://axiommath.ai/territory, 2026, accessed: 2026-02-12. [Online]. Available: https://axiommath.ai/territory

[21] T. Achim, A. Best, A. Bietti *et al.*, "Aristotle: Imo-level automated theorem proving," arXiv:2510.01346, 2025. [Online]. Available: https://arxiv.org/abs/2510.01346

[22] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin, "Dependent types and multi-monadic effects in F," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 256–270. [Online]. Available: https://doi.org/10.1145/2837614.2837655

[23] Amazon Web Services, "Automated reasoning checks in amazon bedrock guardrails," 2025. [Online]. Available: https://docs.aws.amazon.com/bedrock/latest/userguide/guardrails-automated-reasoning-checks.html

[24] S. I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=AjXkRZIvjB

[25] *OMG Systems Modeling Language (SysML), Version 2.0 Beta 2*, Object Management Group, February 2024. [Online]. Available: https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF

[26] Imandra, "Imandra sysml: Formal verification for sysml v2 models," 2024. [Online]. Available: https://www.imandra.ai/sysml

[27] S. Neema, S. Jha, A. Nagel, E. Lew, C. Sureshkumar, A. Gordic, C. Shimmin, H. Nguygen, and P. Eremenko, "On the evaluation of engineering artificial general intelligence," 2025. [Online]. Available: https://arxiv.org/abs/2505.10653

[28] A. Tahat, D. Hardin, A. Petz, and P. Alexander, "Metrics for large language model generated proofs in a high-assurance application domain," in *High Confidence Software and Systems Conference (HCSS'24)*, 2024.

[29] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[31] I. Velickovic, *Microkit User Manual*, 2025. [Online]. Available: https://github.com/seL4/microkit/blob/main/docs/manual.md

[32] J. Hatcliff, "Model-based development for seL4 Microkit/Rust with integrated formal methods using HAMR," Presentation at the seL4 Summit 2025, 2025, accessed: 2025-05-19. [Online]. Available: https://sel4summit2025.sched.com/event/26GD3

[33] Robby and J. Hatcliff, "Slang: The Sireum programming language," in *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, ser. LNCS, vol. 13036, 2021, pp. 253–273.

[34] C. Sun, Y. Sun, D. Amrollahi, E. Zhang, S. Lahiri, S. Lu, D. Dill, and C. Barrett, "VeriStruct: Ai-assisted automated verification of data-structure modules in verus," 2025. [Online]. Available: https://arxiv.org/abs/2510.25015

[35] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140.

[36] A. Tahat, I. Amundson, D. Hardin, and D. Cofer, "Agree-dog copilot: A neuro-symbolic approach to enhanced model-based systems engineering," in *Bridging the Gap Between AI and Reality (AISoLA 2025), Selected Papers*, 2025. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-032-07132-3_8