Neurosymbolic Automated Contract and Code 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*—A significant challenge in Model-Based Systems Engineering (MBSE) for digital avionics is maintaining allocation and traceability between system models and corresponding implementations, while ensuring that assurance properties established at the model level remain valid in code. This challenge is heightened by the sophistication of modern avionics systems, exposure to cyber threats, and strict domain certification requirements. Without strong model-to-implementation traceability, flaws are often discovered late in integration or after fielding. Attempts to integrate Generative AI (GenAI) into these pipelines often exacerbate these problems due to the high complexity of the avionics domain, limitations in open-source training data, and the tendency of Large Language Models (LLMs) to hallucinate—particularly when translating English requirements into formal models or code, or during model-to-code transformations. To address these barriers, this paper introduces the *SCP Codex Copilot*, a deep neuro-symbolic open platform designed to automate the translation of natural language requirements into trustworthy, formally verified SysML v2 contracts and system code. Unlike standard LLM fine-tuning, our approach utilizes "Meta-Rules" to extract and reuse transferable formalization patterns within a self-healing, multi-agent workflow integrated with the INSPECTA verification toolchain. We evaluate the platform using the "Isolette System" benchmark. While a direct formalization baseline failed to produce valid specifications despite high computational cost, the Meta-Rules approach achieved 100% code-level verification success. The results demonstrate orders-of-magnitude improvements in token efficiency, execution time, and correctness, offering a viable path for scalable, trustworthy automated MBSE engineering.**

***Index Terms*—SCP Codex, Neuro-symbolic AI, Formal Methods, SysML v2, MBSE, Self-Healing.**

I. Introduction

The adoption of formal methods in industrial workflows faces significant barriers regarding scalability and expertise. A primary challenge in MBSE for critical systems is ensuring that high-level assurance properties map correctly to low-level implementation. This difficulty is amplified in domains like avionics and space, where systems are highly sophisticated and subject to rigorous certification [**?**].

While Generative AI (GenAI) offers huge potential to accelerate development, its direct application is fraught with risks. LLMs frequently suffer from hallucinations and lack deep knowledge of specialized formalisms like GUMBO contracts. Additionally, the scarcity of open-source training data for these niche languages, combined with restrictions on fine-tuning commercial models, limits the utility of mainstream LLM standard approaches, such as fine-tuning the latest GPT -5 Max models.

To bridge this gap, the SCP Codex Copilot Open Platform leverages the **INSPECTA** toolchain (Industrial-Scale Proof Engineering for Critical Trustworthy Applications) [**?**]. INSPECTA makes formal methods accessible to non-experts by integrating SysML v2 [] modeling with contract specification [**?**], verification, and code generation. It enables a trustworthy MBSE lifecycle supporting model and system-level assume/guarantee reasoning and refinement to memory-safe languages like Rust/Slang with verification support (e.g., Verus [], Slang [], Z3 [], and cvc5 []) using HAMR/logika [] tool-chain [].

This paper details the SCP-Codex Copilot, a deep neuro-symbolic approach that utilizes Meta-Rules to extract transferable formalization knowledge. By automating the generation and repair of formal Gumbo contracts and skeletal code, SCP Copilot enables human users to integrate formal requirements and application logic reliably, with minimal formal methods background, while minimizing impact on existing development processes and integrating proof into CI/CD pipelines.

The main contributions of this paper are:

1) The SCP Codex Copilot, a multi-agent neuro-symbolic open platform for trustworthy English-to-contract and model-to-implementation generation for SysML v2 models.
2) A novel Meta-Rules methodology for extracting and reusing transferable formalization patterns from golden examples, documentation context, and human experts.
3) Empirical evaluation on a realistic life-critical medical device system model, the Isolette incumbent benchmark, demonstrating 100% verification success compared to a baseline failure.

The remainder of this paper is organized as follows. Section II describes the System Architecture and the Meta-Rules integration. Section III presents the key moments from the demonstration, highlighting the baseline failure and the subsequent success. Section IV details the evaluation metrics and performance comparison. Finally, Section V 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

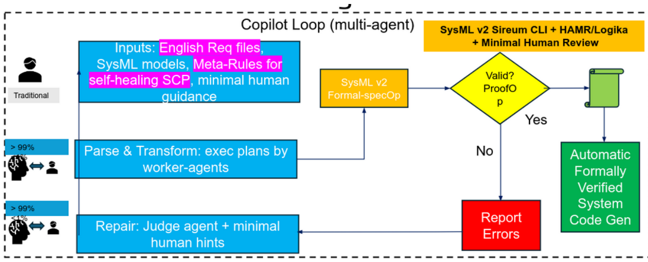

Fig. 1. SCP copilot self-healing neuro-symbolic code generation loop. It parses English requirements, applies meta-rules, and utilizes the INSPECTA toolchain for verification.

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. **??**, 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 [**?**].

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 **??**, 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 **??** 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. **??**. 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 **??**.

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 **??**), 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.

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 **??**), the SCP Copilot successfully formalized all 40+ requirements. The listing below (Listing **??**) displays the copilot's summary report after this successful run.

Listing **??** 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 [**?**].

C. Algorithm

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

```
- 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.

```
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.

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

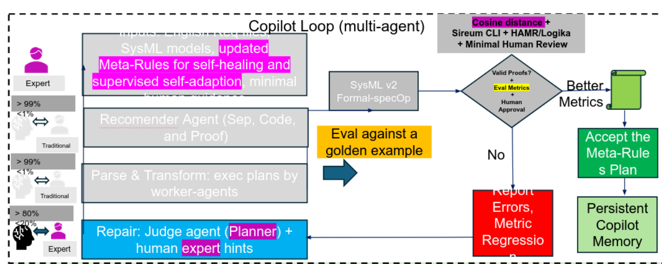

Fig. 2. The semi-automatic supervised learning loop used to generate transferable Meta-Rules from requirement/GUMBO pairs.

Phase II: Rule-Guided Formalization. The second phase 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. **??**.

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

Listing 5. Meta-Rules

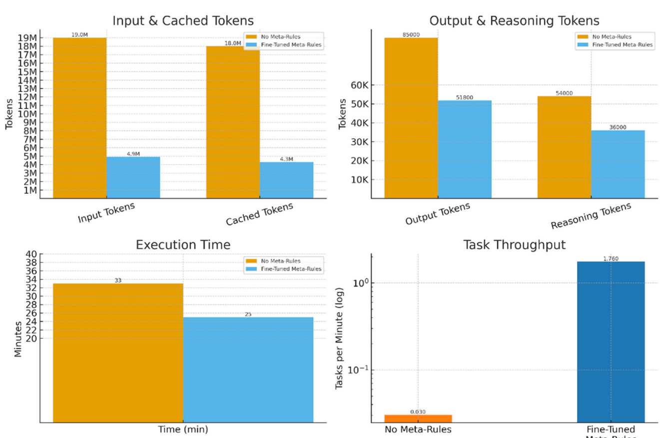

Fig. 3. Performance comparison: Before (Orange) vs. After Meta-Rules (Blue). The charts demonstrate a massive reduction in input/output tokens and execution time, while Task Throughput increases by orders of magnitude.

A. Performance Comparison

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

- **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 [**?**].

```
# 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.

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 ← ∅            // persistent rule memory
C ← ∅; F ← ∅
```

Phase I: Meta-Rule extraction and validation

```
foreach (r_g, g_g) ∈ G do
    m ← ExtractRule(r_g, g_g)
    okCount ← 0
    foreach r ∈ Sample(R) do
        ĝ ← ApplyRule(m, r)
        if Distance(ĝ, g_g) > ε then
            continue
        (ok, fb) ← V(ĝ, B)
        while not ok and
         RepairBudgetRemaining(B) do
            ĝ ← Repair(ĝ, fb, B)
            (ok, fb) ← V(ĝ, B)
        if ok then
            okCount ← okCount + 1
    if okCount ≥ τ then
        M ← M ∪ {m}  // retain only if it
         verifies and generalizes
    else
        DiscardOrRefineWithHuman(m)
```

Phase II: Rule-guided formalization with bounded repair

```
foreach r ∈ R do
    ĝ ← SynthesizeWithRules(r, M)
    (ok, fb) ← V(ĝ, B)
    while not ok and RepairBudgetRemaining(B)
     do
        ĝ ← Repair(ĝ, fb, B)
        (ok, fb) ← V(ĝ, B)
    if ok then
        C ← C ∪ {(r, ĝ)}
    else
        F ← F ∪ {(r, ĝ, fb)}
```

Quality assessment and logging:

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