

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—This document illustrates the key moments from the SCP Codex demo.

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

I. INTRODUCTION

As outlined in the SCP Program presentation, the adoption of formal methods in industrial workflows faces significant barriers. The SCP Codex Copilot Open Platform addresses these by leveraging Generative AI (GenAI) to accelerate Model-Based Systems Engineering (MBSE) while ensuring correctness through formal verification. This report details the "SCP-Codex Copilot," a deep neuro-symbolic approach that utilizes "Meta-Rules" to extract and reuse transferable formalization knowledge, rather than relying solely on base-model fine-tuning [1].

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 [1].

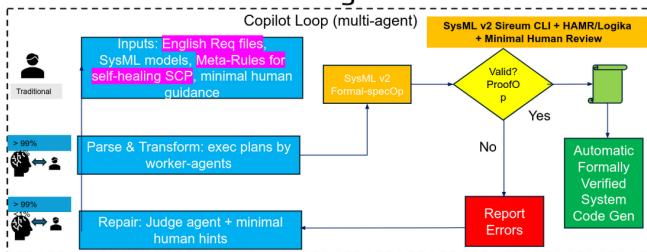


Fig. 1. SCP copilot self-healing neuro-symbolic code generation loop. It parses English requirements, applies meta-rules, and utilizes the INSPECTA toolchain for verification.

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 Listing ??, 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 1 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 1. 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

```
- 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 2. Baseline Failure Log: Transcript from the failed session showing parser rejection and massive token usage.

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 2), 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."

```
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 3. Success Log: Transcript showing successful formalization and verification after applying Meta-Rules.

B. Moment 2: Success via Meta-Rules

Upon applying the Meta-Rules (as shown in Listing 1), the SCP Copilot successfully formalized all 40+ requirements. The listing below (Listing 3) displays the copilot's summary report after this successful run.

Listing 3 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 [1].

C. Algorithm

Algorithm 1 (Figure 2) 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 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

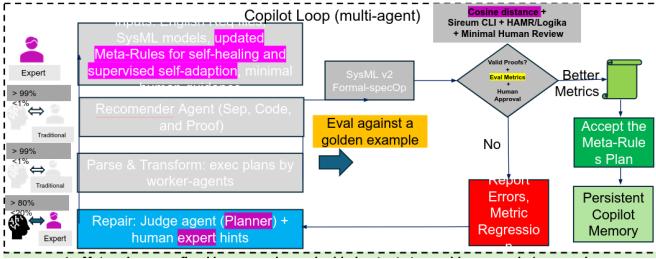


Fig. 2. The semi-automatic supervised learning loop used to generate transferable Meta-Rules from requirement/GUMBO pairs.

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 formalizations¹. 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 syntactic 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. Using the specific error feedback (fb) from the compiler or prover, the Repair function iteratively adjusts the specification. This loop is strictly constrained by a budget (B) of tokens and repair cycles to ensure termination. Acceptance: Only rules that successfully verify across the sample set—meeting the acceptance threshold (τ)—are committed to the persistent Meta-Rules library (M). Rules that fail are flagged for DiscardOrRefineWithHuman supervision²². Phase II: Rule-Guided Formalization. The second phase represents the production deployment of the Copilot. Instead of relying on the stochastic generation of a raw Large Language Model (LLM), the system utilizes the validated library (M) to synthesize formal contracts (\hat{g}) for the full requirements set (R). Synthesis: The SynthesizeWithRules function applies the deterministic patterns learned in Phase I, significantly reducing the search space and the likelihood of hallucination. Self-Healing: A similar verification-guided repair loop is employed. The system attempts to automatically resolve semantic errors (e.g., type mismatches, range violations) using formal feedback (fb). Outcome: Specifications that pass verification are added to the set of verified contracts (C), while stubborn failures that exceed the repair budget are isolated in set (F) for expert human review³³³³. This algorithmic structure ensures that the high computational cost of reasoning is front-loaded into the learning phase, while the production phase benefits from efficient, rule-based execution with guaranteed formal correctness.

2) *Discussion: Human-in-the-Loop Refinement and Metric-Driven Generalization.* 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 for grounding the "Teacher" agent's understanding of system behavior. A critical aspect of the framework is the Generalization-Specialization Trade-off, which is managed through a hybrid of mechanical metrics and human review:
1. Semantic Proximity and Cosine Distance: The system employs a cosine distance metric to evaluate the semantic similarity between the contract generated by a candidate Meta-Rule and the "Golden" reference formalization. This metric is not merely a pass/fail check but a signal for tuning:
Overfitting Signal (Distance ≈ 0): Counter-intuitively, a cosine distance of zero may signal that a rule is too specific. If the generated output matches the training example perfectly but lacks abstraction, the rule may fail to transfer to a larger class of similar problems. In such cases, the human expert intervenes to generalize the rule structure, ensuring it captures the intent rather than just the instance.
Semantic Sufficiency: The objective is not identity, but semantic equivalence. The distance threshold is learned experimentally to identify when an instantiation is "semantically close enough" to be correct while remaining general enough to be reusable.
2. Human-Guided Refinement: The human expert reviews the output of learned rules to correct biases that statistical metrics might miss:
Specialization: If a candidate rule is too general, it risks producing hallucinations when applied to corner cases. The expert addresses this by providing additional, specific examples to constrain the rule's application logic.
Generalization: Conversely, if a rule is overly rigid (valid only for the specific training examples), the expert modifies the template to broaden its applicability to a wider class of requirements. This feedback loop ensures that the final Meta-Rules Library (M) consists of verified teaching artifacts that are neither effectively "hard-coded" to the training data nor so vague that they compromise formal correctness.

IV. EVALUATION AND METRICS

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

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.

- **Token Reduction:** The baseline (Orange) consumed $\approx 19M$ input tokens. After applying Meta-Rules (Blue), input tokens dropped to $\approx 4.9M$ (cached) and only $\approx 200k$ uncashed.

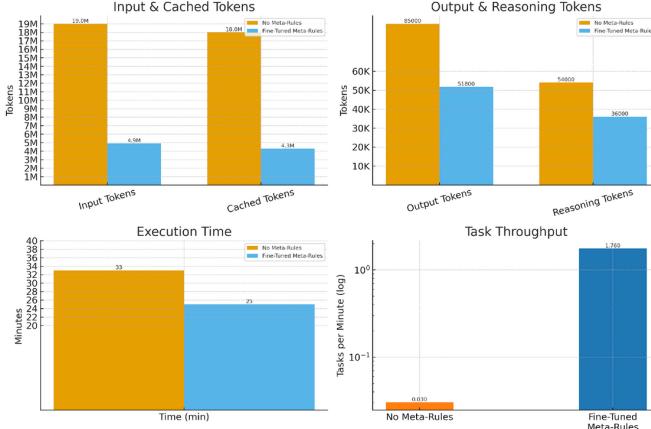


Fig. 3. Performance comparison: Before (Orange) vs. After Meta-Rules (Blue). The charts demonstrate a massive reduction in input/output tokens and execution time, while Task Throughput increases by orders of magnitude.

- **Reasoning Efficiency:** Output and reasoning tokens saw significant reductions, dropping from $> 80k$ to $\approx 51k$ 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 [1].

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] A. Tahat, D. Hardin, I. Amundson, and D. Cofer, “SCP-Codex Open Platform: A Deep Neuro-symbolic Approach for Trustworthy MBSE-driven Code Generation,” Collins Aerospace & DARPA PROVERS, Dec. 2025.

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
            REPAIRBUDGETREMAINING( $B$ )
             $\hat{g} \leftarrow \text{REPAIR}(\hat{g}, fb, B)$ 
             $(ok, fb) \leftarrow V(\hat{g}, B)$ 
        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
        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 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.
