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

Amer Tahat, David Hardin, Isaac Amundson, Junaid Babar, Timothy A. Barclay, Janet Liu, Karl Hoech, and Darren Cofer
Collins Aerospace, an RTX Business
{amer.tahat, david.hardin, isaac.amundson, junaid.babar, timothy.a.barclay, janet.liu, karl.hoech, 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 INSPECTA formal methods pipeline, developed on the DARPA Pipelined Reasoning of Verifiers Enabling Robust Systems (PROVERS) program. The pipeline translates natural-language requirements into SysML v2 models containing formal GUMBO contracts, generates HAMR infrastructure code, and discharges Logika proofs, ensuring properties are mathematically preserved in the implementation. To enable specialization without fine-tuning, we introduce a *Meta-Rule* learning methodology that functions as a verifiable surrogate for weight adaptation. Core to SCP is supervised self-healing and self-adaptation: successful in-context interactions are crystallized into persistent, expert-curated abstraction patterns, and Meta-Rules then drive a generate–verify–repair loop until the selected verification targets are satisfied within a given budget.

We evaluate our methodology on the Isolette infant-incubator system, a multi-subsystem HAMR/INSPECTA benchmark sourced from the FAA Requirements Engineering Management Handbook. Within the defined Isolette/HAMR target scope, our copilot achieved 100% code-level verification and an 18× adjusted increase in development task throughput compared to a prompt-only baseline that, lacking Meta-Rules, failed to produce verifiable artifacts. This demonstrates a practical path to certification-facing automation under strict data and model governance constraints, while scoping the claim to HAMR-centric, toolchain-aware GUMBO repair; broader cross-system validation and additional rule books remain ongoing work.

***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 to executable 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 model-level assurance and the software implementation correctness. The resulting drift is often detected late—during integration, verification/validation, or penetration testing—driving rework, schedule risk, and certification friction [1].

Large Language Models (LLMs) appear well-suited to assist with this decomposition 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]) are typically restricted by vendor policy and/or cost [2], [4].

In this setting, the key challenge therefore is not merely to generate plausible formal text, but rather to produce a *verified* artifact within bounded time, token, and human-review budgets. Recent theory on AI agents in verifiable settings provides a useful lens for this problem: past experience matters to the extent that it captures reusable algorithmic structure that reduces the time needed to solve new tasks [5]. Viewed through this lens, the important quantity is not raw fluency alone, but *cost per verified task*. This framing is particularly natural in MBSE pipelines where a formal checker is available and all candidate artifacts are filtered through hard acceptance gates. These restrictions leave three highly desirable capabilities unmet: (i) reducing token-heavy, brittle prompting; (ii) filtering or rejecting hallucinated behaviors with machine-checkable acceptance gates; and (iii) working from natural language specifications to produce 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 natural language interface while accumulating governed, version-controlled specialization knowledge. Our approach integrates Codex-class software engineering agents into the Industrial-Scale Proof Engineering for Critical Trustworthy Applications (**INSPECTA**) toolchain [1], developed on the DARPA PROVERS program. INSPECTA provides the formal backbone: **SysML v2** [6], [7] captures the system architecture, **GUMBO** [8], [9] expresses behavioral assume/guarantee contracts, **HAMR** [10], [11] generates executable memory-safe infrastructure code, and **Logika** [11] discharges proof obligations using SMT solvers (e.g., Z3 [12] and CVC5 [13]).

This setting is challenging because a HAMR application is a layered assurance object, not a single text-to-code artifact. Requirements, SysML v2 architecture, GUMBO contracts, HAMR-generated Slang/Scala code, Logika/SMT obligations, build artifacts, diagnostics, and repair traces must remain mutually consistent; with seL4/Microkit targets, the model, generated implementation, platform glue, and proof-facing artifacts can span tens of thousands of lines across languages and verification layers [10], [14], [15]. In a naive agentic workflow, upstream stochastic edits can trigger downstream failures and repeated replay of models, code, logs, and counterexamples. The research question is therefore how to combine stochastic generation with symbolic feedback so high-assurance AI assistance remains bounded, auditable, and repairable.

A major goal of INSPECTA is to create 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 exercise. Ideally, an agentic copilot would assist engineers in synthesizing these rigorous specifications from natural language directly; however, realizing this capability faces the critical barrier of 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 [16], but the effect is transient: once the context window is cleared, the learned behavior is lost. In high-assurance settings, this ephemerality is a significant obstacle: it forces repeated token-heavy prompting and makes it difficult to govern, audit, or reuse what the model has "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. In the evaluated FSE rule book, a rule binds four kinds of reusable knowledge: linguistic triggers in requirements, semantic formalization choices, structural placement in SysML v2/GUMBO, and repair actions induced by HAMR/Slang/Logika feedback. 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 INSPECTA tool (e.g., 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 [16] 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 practical formally verifiable output at low cost (see Section V-C), and thus 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 AWS Kiro [17] 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 formal GUMBO contracts, then discharged through HAMR/Logika verification. This yields machine-checkable evidence alongside generated code, enabling a workflow aligned with certification demands for traceability, repeatability, as well as 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 Self-Adaptation Algorithm:** A HAMR/INSPECTA-scoped adaptation method that externalizes successful in-context learning into persistent, verifier-gated Meta-Rules. Within the evaluated GUMBO–HAMR–Slang benchmark, semantic screening, HAMR/Logika feedback, and human approval guide a governed generate–verify–repair loop toward the defined verification targets.

3) **System-Level HAMR Benchmark Evaluation:** An evaluation on the Isolette infant-incubator system [18], a realistic multi-subsystem HAMR/INSPECTA benchmark spanning requirements, SysML v2 models, GUMBO contracts, HAMR-generated Slang/Scala obligations, and Logika proof discharge. Within the scoped GUMBO FSE Rule book evaluation, Meta-Rules learned from a small solved file/subsystem example were reused on the remaining target files, converting a prompt-only failure into a fully verified target-scope run.

The remainder of this paper is structured as follows. After establishing the SysML v2 GUMBO and HAMR/Logika background in Section II-A, Sections III and IV introduce the SCP architecture, the Meta-Rules framework, and the supervised self-adaptation algorithm. Section IV-D compares Meta-Rules to fine-tuning and transient prompting. Section V then defines the Isolette/HAMR benchmark and reports the evaluation metrics and traces. Sections VI–VIII discuss related work, limitations, and conclusions.

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 [19].

II. Background

A. INSPECTA High-Assurance System Development and Verification

The development of safety-critical systems, including digital avionics systems, requires rigorous evidence that implementations faithfully realize their requirements. The DARPA PROVERS INSPECTA toolchain provides this verifiable setting by automatically integrating architectural design, code generation, and formal verification capabilities. Specifically,

- SysML v2 and GUMBO: Architecture models are captured in SysML v2, which natively supports a textual representation amenable to LLM processing. These models are annotated with GUMBO contracts. GUMBO allows engineers to attach formal specifications in the form of *assumptions* and *guarantees* directly to components (parts) in the architecture model.
- HAMR and Logika: The HAMR framework analyzes these models to generate memory-safe executable infrastructure code in languages like Slang/Scala (default and target of this paper), C, or Rust; mapping GUMBO assumptions and guarantees to formal verifiable *requires* and *ensures* clauses, respectively. Subsequently, in the case of Scala/Slang, the Logika formal verification framework uses SMT solvers (e.g., Z3, CVC5) to discharge proof obligations, ensuring the component application code (developed manually or via LLM assistance) satisfies its GUMBO contracts across the entire state space.
- Finally, Rust and Verus: For Rust targets, HAMR generates Verus verification annotations. Verus allows developers to specify and prove properties using Rust's ownership model, mapping GUMBO assumptions and guarantees to Verus *requires* and *ensures* blocks, respectively. In this paper, our SCP copilot supports the default behavior.

Figure 1 merges the INSPECTA assurance stack with the SCP agentic repair loop used in this paper. The specification agent proposes or updates SysML v2/GUMBO contracts under guidance from persistent Meta-Rules. The verifier stack then acts as a hard gate: successful checks emit proof artifacts, while failed checks return diagnostics and counterexamples to the repair agent. The repair agent converts those diagnostics into a suggested repair and modified plan, which is fed back to the specification agent for a bounded retry. In this sense, Meta-Rules and verifiers act together as guided generation: they constrain the search space before expensive multilayer verification is attempted and prevent unbounded token-heavy trial-and-error.

```
language "GUMBO" /* { state lastCmd: On_Off;
  initialize
    guarantee initLastCmd: lastCmd == Off;
  compute
    assume lowerLEUpper: lower <= upper;
    guarantee lastCmdTracksOutput: lastCmd ==
        heat_control;
  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 example of a GUMBO contract on a SysML v2 component.

A typical GUMBO block includes optional state variables, initialization guarantees, assumptions constraining inputs, global guarantees, and `compute_case` clauses that align conditional requirements with case-specific guarantees. Listing 1 provides an illustrative example.

III. Neuro-Symbolic Contract Generation with Meta-Rules: An Overview

A. 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 INSPECTA-compatible artifacts. In this paper's quantitative evaluation, we instantiate the mechanism with a single Formal Specification Engineer rule book, the GUMBO FSE Rule book. The book is GUMBO-centered because its primary rules concern state, initialization, assumptions, guarantees, and `compute_case` construction; however, it is not GUMBO-isolated because successful contracts must also attach to the correct SysML v2 artifact, preserve names used by generated Slang/Scala code, and respond to HAMR/Logika diagnostics. In contrast to transient prompt-only conditioning, the rule book is a local, version-controlled artifact that can be reviewed, curated, replayed, and audited. Conceptually, solved requirement-to-GUMBO episodes serve as worked cases: the extracted Meta-Rule is a reusable lemma-like hypothesis, semantic screening estimates where it is likely

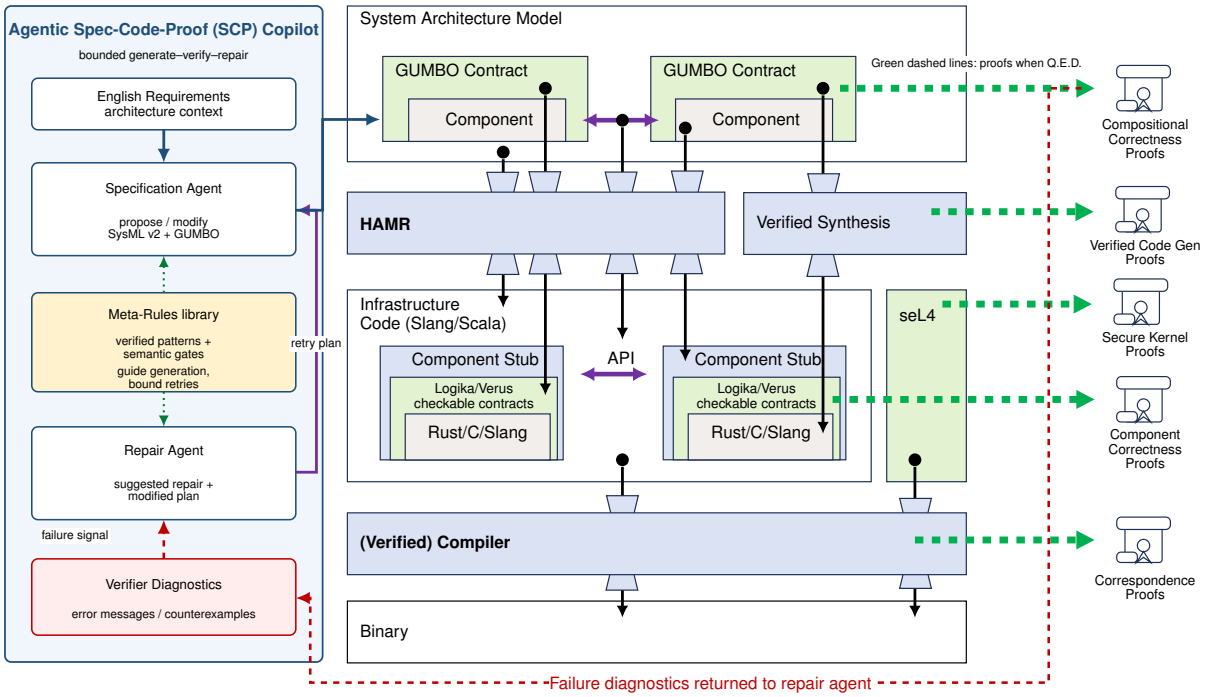

Fig. 1. Agentic SCP/INSPECTA workflow integrating the high-assurance development layers with a Meta-Rule-guided generate–verify–repair loop. Green dashed arrows denote successful proof artifacts; the red dashed arrow denotes verifier-failure diagnostics returned to the repair agent for a bounded retry.

to apply, and the HAMR/Logika verifier stack plus human review decide whether the hypothesis is worth retaining within the available budget.

B. Operational Modes of Use: User Mode vs. Developer Mode

Operationally, we distinguish two 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 (supervised self-adaptation learning).** 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 explicit base-model fine-tuning.

Scoped GUMBO FSE rule book. The evaluated library is deliberately scoped to the GUMBO FSE Rule book. Its quantitative role is GUMBO formalization, but its repair knowledge is toolchain-facing: it records where contracts attach in SysML v2, how requirement and port names must remain consistent, which GUMBO constructs translate to generated Slang/Scala obligations, and how HAMR/Logika diagnostics should guide bounded repair. Broader rule books for process synthesis, port/connection construction, seL4/Microkit mappings, Rust/Verus repair, and other systems are treated as future work rather than quantitative evidence in this paper.

1) Geometric Intuition: Practically Verified Semantic Cone: Figure 2 provides intuition for why Meta-Rules act as a verifiable surrogate for fine-tuning in SCP. After injecting a curated Meta-Rule candidate $\mathcal{R}$, verification-aligned performance forms a local maximum over the base-model manifold $\mathcal{M}$, centered around a golden example direction $\mathbf{g}$. For the GUMBO-focused setting, an applicability screen can be instantiated as a cosine-distance gate $d_{\cos}(r, m) = 1 - \frac{e(r)\cdot c_m}{\|e(r)\|\|c_m\|}$, where $e(r)$ embeds the new requirement and c_m is the centroid of the rule's solved examples; normalized token-difference checks can be used as a syntax-sensitive alternative. Candidate applications with $d \leq \epsilon$ are attempted, and HAMR/Logika verification plus human review decide whether the resulting contract is retained.

2) Toy Example Meta-Rule: Listing 2 shows a compact excerpt from the GUMBO FSE Rule book. The rule directs the agent to introduce explicit GUMBO state when the English requirement implies memory, a failure mode observed in the prompt-only baseline.

```
Rule: introduce explicit GUMBO state for memory.
Trigger: hysteresis, "hold previous", "shall not
    change".
Semantic choice: model memory as state +
    initialization.
Structural target: attach compute_cases to owning
    SysML thread.
Repair gate: keep only if HAMR/Slang/Logika checks
    succeed.
```

Listing 2. Compact FSE rule excerpt linking linguistic, semantic, structural, and repair patterns.

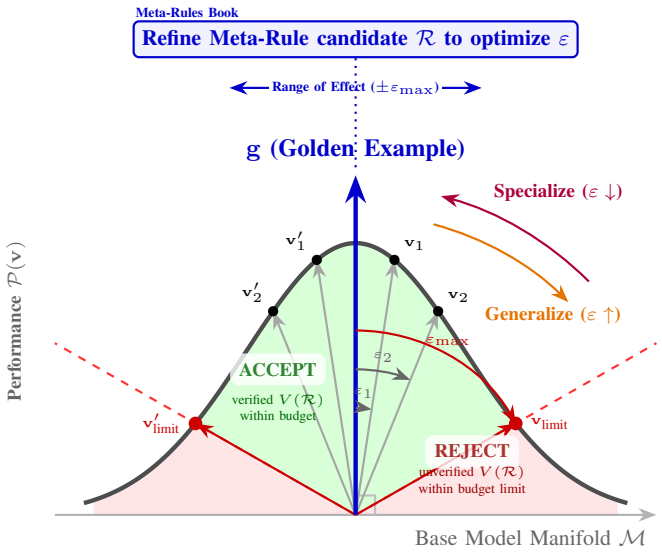

Fig. 2. Practically verified semantic cone for supervised Meta-Rule refinement. A golden example direction $\mathbf{g}$ anchors the accepted region. A candidate direction $\mathbf{v}_i$ is screened by a distance such as $d_{\cos} = 1 - \cos(\angle(\mathbf{v}_i, \mathbf{g}))$ and is applied only when $d_{\cos} \leq \epsilon$; it is retained only if the resulting GUMBO contract verifies within the budget. The accepted region therefore represents budget-verifiable generalization, while the rejected region corresponds to semantic, budget, or toolchain mismatch.

IV. Supervised Meta-Rule Self-Adaptation (Learning) and Acceptance Criteria

A. The Algorithm

Algorithm 1 introduces a dual-phase neuro-symbolic process. Let R denote the target set of natural-language requirements, and let $\textsc{Sample}(R) \subset R$ denote the sampled training subset used in developer mode during Phase I—in our setting, its size was 1/6 of R. Let $G = \{(r_g, g_g)\}$ denote the set of golden examples, where r_g is a golden requirement and g_g is its verified formalized GUMBO contract. Let M denote the retained Meta-Rule library, C the set of verified contracts, and F the set of flagged failures.

Phase I: Meta-Rule extraction, sample-based training, and predictive ϵ-learning. For each golden pair $(r_g, g_g) \in G$, $\textsc{ExtractRule}$ derives a candidate Meta-Rule m. For each sampled requirement $r \in \textsc{Sample}(R)$, $\textsc{ApplyRule}$ uses m to produce a candidate formalized contract $\hat{g}$. The quantity $\textsc{Distance}(\hat{g}, g_g)$ denotes a *semantic* distance between the generated formalized contract and the golden formalized contract, rather than a literal textual comparison. The threshold ϵ therefore controls the admissible degree of abstraction away from the golden exemplar. The objective is not to drive ϵ toward zero: if ϵ is too small, the rule becomes overspecialized and loses reusability; if ϵ is too large, the rule admits overly broad generations that require excessive repair or fail to verify within budget B. Accordingly, in developer mode ϵ is tuned across repeated calibration attempts to find a satisfactory intermediate value that best balances rule reuse and bounded repair. Verification is performed by $V(\hat{g}, B)$, where B is the repair budget in time and tokens for full repair cycles; the verifier returns a Boolean result ok together with feedback fb. While $\textsc{RepairBudgetRemaining}(B)$ holds, $\textsc{Repair}(\hat{g}, fb, B)$ revises the candidate using verifier feedback. The counter $okCount$ records how many sampled requirements are successfully verified for the candidate rule m, and τ is the minimum acceptance threshold required to retain m in M. If performance is unsatisfactory, $\textsc{DiscardOrRefineWithHuman}(m)$ permits human refinement of the rule, of ϵ, or both, and the developer-mode loop repeats.

Phase II: held-out validation and rule-guided formalization with bounded repair. After Phase I, the retained pair (M, ϵ) is evaluated beyond the sampled training subset. In developer mode, the Phase II loop is instantiated on requirements not used in $\textsc{Sample}(R)$, testing whether the learned Meta-Rules and calibrated threshold ϵ generalize to unseen requirements while remaining repairable within budget B. In user mode, the same rule-guided synthesis procedure is applied to the full deployment set R. For each requirement r, $\textsc{SynthesizeWithRules}(r, M)$ produces a candidate contract $\hat{g}$, which is checked again by $V(\hat{g}, B)$ and, if needed, revised by $\textsc{Repair}(\hat{g}, fb, B)$ while budget remains. If verification succeeds, the pair $(r, \hat{g})$ is added to C; otherwise the tuple $(r, \hat{g}, fb)$ is added to F. Operationally, the developer-mode loop (Phase I and Phase II) repeats until a satisfactory ϵ is found, namely one for which the retained Meta-Rule library remains reusable on unseen requirements within the available repair budget.

B. Scoped GUMBO Rule Book and Function Templates

The evaluated Meta-Rule artifact is the single GUMBO-centered Formal Specification Engineer rule book, referred to here as the *GUMBO FSE Rule book*. We use it as the retained library M in Algorithm 1; broader rule books are excluded from the quantitative evaluation and left to future work. Although the book mainly targets GUMBO formalization, it is toolchain-aware: retained rules record SysML v2 attachment, naming consistency, generated Slang/Scala compatibility, and HAMR/Logika repair cues. The book was extracted/refined from a small solved GUMBO file/subsystem example and then applied to the remaining Monitor/Regulate target files.

```
ExtractRule(req, golden, log):
  record trigger, formal pattern, target construct
  attach SysML/GUMBO placement, naming, Slang,
    verifier expectations, repair_hints, confidence
  return auditable Meta-Rule record

ApplyRule(rule_book, req, active_model):
  load frozen GUMBO FSE rules and active context only
  if semantic_distance(req, rule) > epsilon: flag
  emit GUMBO state/init/compute_cases on owner thread
  report rule ids, ambiguities, checks; run HAMR/Logika

Repair(candidate, feedback, budget):
  classify syntax/attachment/name/type/Slang/Logika error
  minimally edit SysML/GUMBO; do not weaken contracts
  rerun verifier while budget remains; return ok/failure
```

Listing 3. Compact ExtractRule, ApplyRule, and Repair templates from the goal files.

The complete rule book, extraction/application/repair templates, evaluation logs, and run videos are linked in the supplementary artifact package [19].

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

Input: Requirements set R; golden examples G (English + formalized 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.

Listing 3 gives compact templates adapted from the learning and user-mode goal files. They show the three roles used in Algorithm 1: extract an auditable rule, apply the frozen book to active project context, and repair only the smallest contract/model fragment when verifier feedback requires it.

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

	Fine-tuning	Prompting/ICL	Meta-Rules
Persistent across runs	Yes	No	Yes (external)
Model weights tuning	Explicit	Implicit	Implicit
Cost	High	Med	Low–Med
Auditability / review	Low	Low–Med	High
Governance / rollback	Low	Low	High
Runtime context overhead	Low	High	Low–Med
Data / IP constraints	High	Low–Med	Low
FV acceptance gates	N/A	N/A	Primary
HAMR toolchain reuse	N/A	Ad hoc	Primary

C. Practical Considerations and Human Governance

Meta-Rules are stored as local, version-controlled artifacts and 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. While extraction and validation are highly automated, human oversight remains essential to prevent drift and to decide whether a candidate rule is worth retaining. The threshold ϵ is therefore calibrated as a satisficing engineering operating point rather than as a theoretical maximum: it is accepted once the learned pair (M, ϵ) achieves the user-required verification performance on unseen requirements within the available budget. Human review remains part of this calibration loop because the final choice of ϵ reflects trade-offs among generality, verification success, repair effort, and cost.

D. Comparison to Fine-Tuning and Transient Prompting

This subsection situates Supervised Meta-Rules Adaptation relative to parameter fine-tuning and transient prompting/ICL. Fine-tuning modifies model weights and can yield strong performance, but it requires suitable data, training authorization, compute, and checkpoint governance that are often difficult in aerospace settings. Prompt-only ICL avoids weight updates, and can behave like "implicit fine-tuning" during inference [16], but its effects are ephemeral: when the context is reset, the learned behavior is lost, encouraging repeated long-context prompting.

Meta-Rules externalize successful ICL behavior into persistent, version-controlled, verifier-gated artifacts. The surrogate-for-fine-tuning claim is therefore operational rather than universal: Meta-Rules retain task specialization without modifying base-model parameters, while preserving inspectability, rollback, and formal verification-gated acceptance. Table I summarizes the comparison.

The next section defines the Isolette/HAMR benchmark and reports the two evaluation traces used to measure verification coverage, token cost, and HAMR-scoped reuse.

V. EXPERIMENTAL EVALUATION: THE ISOLETTE GUMBO BENCHMARK

We use the Isolette infant-incubator system as a system-level HAMR/INSPECTA benchmark rather than an isolated text-

to-contract example [18]. Isolette contains interacting monitor and regulator subsystems, operator-interface behavior, device/sensor/actuator interactions, and multiple SysML v2 specification/library files. It exercises the HAMR assurance path: requirements become SysML v2/GUMBO artifacts, HAMR generates Slang/Scala proof obligations, and Logika/SMT diagnostics drive repair. The quantitative evaluation focuses on one rule book, the GUMBO FSE Rule book; claims are limited to GUMBO-centered, HAMR-toolchain-aware formalization and repair.

A. Experimental Design and Scope

Inputs. The experiment uses (i) natural-language requirements from a realistic FAA-style source containing text, tables, and figures, and (ii) SysML v2 models/shared libraries for the Isolette architecture, including `Monitor.sysml` and `Regulate.sysml` [20]. These artifacts include terminology, cross-references, and memory-bearing requirements such as hysteresis that must be encoded in tool-accepted GUMBO. Naive repair can replay requirements, model fragments, generated Slang/Scala code, build logs, proof obligations, and counterexamples, producing long-context blow-up across the stack.

Scope of evaluation. The target set contains 42 proof-bearing units: GUMBO contract-bearing blocks, generated or modified `compute_cases`, and corresponding HAMR-generated methods checked by Logika for the Monitor/Regulate target. This scope is explicitly bounded, but remains system-level and multi-layered because a GUMBO decision must survive SysML v2 placement, HAMR generation, generated Slang/Scala obligations, and Logika proof discharge.

Runs and training split. We compare a prompt-only baseline with a Meta-Rule run using the GUMBO FSE Rule book at $\epsilon = 0.1$. Developer mode extracts/refines the rule book from a small solved GUMBO file/subsystem example, then applies the retained knowledge to the remaining target files. This tests intra-system, cross-subsystem reuse under HAMR/INSPECTA; the reported adjusted trace already includes extraction/refinement and successful verification.

B. Observed Failure and Self-Healing Behavior

The baseline repeatedly attempted incompatible pre-state idioms for memory-bearing requirements and did not converge on tool-accepted GUMBO state syntax. The Meta-Rule-guided run uses the GUMBO FSE Rule book to introduce explicit state, initialization guarantees, and structured `compute_cases`; it also uses toolchain-facing repair cues for SysML attachment, naming consistency, generated Slang/Scala obligations, and HAMR/Logika diagnostics. Listing 4 gives the condensed baseline log; the verified Meta-Rule-guided result and adjusted cost are reported in Table II and Fig. 3 to avoid duplicating the same cost trace in a separate listing.

```
Baseline: Parser rejected memory idioms:
  "rejects every variant of the state ... syntax"
  "@pre as an unsupported classification test"
Result: could not encode hysteresis/no-change cases.
Tokens: input=981,770 (+18,673,920 cached),
        output=85,833.
```

Listing 4. Condensed baseline failure excerpt: parser rejection and high token usage.

C. Metrics, Pricing Model, and Results

The evaluation measures more than final correctness. In a multi-layer HAMR workflow, early formalization errors can propagate into generated implementation artifacts and proof obligations, increasing both the number of repair paths and the amount of context that must be reprocessed. The token, throughput, and adjusted-cost metrics therefore capture whether the copilot reaches verifier-accepted artifacts without allowing stochastic repair to expand beyond the available budget.

For a run r, we report wall-clock time $T_{wall}(r)$, uncached input tokens $N_{in}(r)$, cached input tokens $N_{cache}(r)$, and output-side tokens $N_{out}(r)$. We do not separately tabulate reasoning tokens in the main results because the pricing model used here distinguishes input, cached input, and output-side tokens; any separately logged reasoning count is therefore treated as part of the output-side cost discussion rather than as a separate priced class. We define the context-processed total

$$N_{ctx}(r) = N_{in}(r) + N_{cache}(r) + N_{out}(r). \tag{1}$$

This distinction matters because current provider pricing differentiates uncached input, cached input, and output tokens.[1] Let $pm = (p_{in}, p_{cache}, p_{out})$ denote the selected pricing model. The illustrative dollar cost of a logged run is

$$C(r; pm) = \frac{p_{in}N_{in}(r) + p_{cache}N_{cache}(r) + p_{out}N_{out}(r)}{10^6}. \tag{2}$$

Because the illustrative pricing model sets $p_{cache} = 0$, cached tokens affect context-size and burn-rate metrics but not the dollar estimates in Table II.

Adjusted cost. The adjusted trace in Table II is a complete logged run that already includes the one-time developer-mode extraction/refinement effort for the GUMBO FSE Rule book and the successful verification run. Therefore, for that trace we do not add a separate user-mode run again. We define

$$N_{adj}(r) = N_{ctx}(r), \qquad C_{adj}(r; pm) = C(r; pm), \tag{3}$$

and report the adjusted per-verified-unit quantities

$$K_{adj}(r) = \frac{N_{adj}(r)}{|U_{verified}(r)|}, \qquad K^{adj}_{\$}(r) = \frac{C_{adj}(r; pm)}{|U_{verified}(r)|}. \tag{4}$$

When developer-mode and later user-mode logs are measured separately, the amortized form for H future reuse opportunities is

$$C^{(H)}_{adj}(r; pm) = C_{use}(r; pm) + \frac{C_{dev}(pm)}{H}. \tag{5}$$

[1] Pricing reference used for the model-cost discussion: https://developers.openai.com/api/docs/pricing. At the time of writing, the illustrative standard Codex price tags used in Table II are p_{in} = \$60/M long-context input tokens, p_{cache} = \$0/M cached tokens, and p_{out} = \$270/M output-side tokens. Prices and thresholds change over time, so the paper reports token classes and symbolic prices rather than treating dollar amounts as invariant.

This paper reports the more conservative complete adjusted trace for the scoped GUMBO experiment.

Let $U_{verified}(r) \subseteq U$ denote the verified subset of the 42 targeted units. We define

$$\rho(r) = \frac{|U_{verified}(r)|}{|U|}, \tag{6}$$

$$\Theta(r) = \frac{|U_{verified}(r)|}{T_{wall}(r)}, \tag{7}$$

$$K(r) = \frac{N_{ctx}(r)}{|U_{verified}(r)|}. \tag{8}$$

For runs with $T_{wall}(r) > 0$ and $|U_{verified}(r)| > 0$, the context-token burn rate is $\Gamma_{ctx}(r) = N_{ctx}(r)/T_{wall}(r) = K(r)\Theta(r)$. For partial verification, comparisons are defined only over the common target subset, not merely by the number of verified units. Table II reports the adjusted values and Fig. 3 visualizes the same scoped comparison.

Interpretation. Table II reports the adjusted cost for the evaluated Meta-Rule condition, and Fig. 3 restores the visual comparison for the same scoped traces. The adjusted $\epsilon = 0.1$ trace includes the one-time extraction/refinement effort for the GUMBO FSE Rule book and the successful verification run in the same logged trace. Even without amortizing that one-time investment across future uses, the Meta-Rule-guided trace reduces cost per verified unit from $\approx$ 9.87M to $\approx$ 329K context tokens and from \$41.04 to \$0.92 in the illustrative pricing model. The later user-mode run, after the rule book exists, gives the usage-phase throughput reference $\Theta = 1.68$ units/min; the adjusted complete run gives the more conservative $\Theta = 1.08$ units/min after including the one-time effort.

VI. Related Work

LLMs have increasingly been used for formal reasoning, including theorem proving, autoformalization, proof repair, symbolic reasoning, and code verification. GPT-f explored generative theorem proving in Metamath [21], while Baldur and CoqDog-style systems study proof generation, repair, and diagnosis in Isabelle/HOL and Coq [22], [23]. Other neuro-symbolic and verification-oriented workflows integrate LLMs with SMT-backed checks, symbolic benchmarks, or code verification tasks [24]–[26]. These works are conceptually related, but they typically operate at the level of proof scripts, theorem libraries, agent-world models, or standalone code analysis rather than at the MBSE-to-generated-system boundary.

Formal methods are also entering MBSE standards and tools: SysML v2 was designed with formal semantics in mind [27], Imandra has demonstrated model-level SysML v2 verification [28], and AGREE-Dog supports AADL/AGREE model-level explanation and repair [29], [30]. SCP differs in its evaluated boundary and learned artifact: its GUMBO FSE Rule book helps produce contracts placed in SysML v2, preserved through HAMR generation, reflected in generated Slang/Scala obligations, and discharged by Logika. Although our evaluated build uses HAMR's JVM/Slang execution path rather than the seL4 Microkit path, HAMR supports JVM, Linux, and seL4 targets and has been used for model-driven construction of seL4 and avionics-style systems [1]. Thus, SCP targets the upstream bottleneck for such high-assurance builds: learning HAMR-centric memories of linguistic triggers, GUMBO choices, structural placement decisions, and verifier-guided repairs.

VII. Limitations and Future Work

Our evaluation uses the Isolette infant-incubator system [18], a realistic HAMR/INSPECTA benchmark spanning natural-language requirements, SysML v2 modeling, GUMBO contract synthesis, HAMR-generated Slang/Scala proof obligations, and Logika discharge. Within this scope, SCP shows that reusable formalization and repair knowledge can be learned from a small set of solved file- and subsystem-level examples and reused to verify remaining target files. In particular, it converts a prompt-only failure into an end-to-end fully verified target-scope run while substantially reducing cost and time, despite the restriction that the base model was not fine-tuned.

The current evaluation is currently limited to toolchain-aware SysML v2/GUMBO formalization and repair for the Isolette/HAMR target; broader validation remains future work. Planned extensions include additional HAMR rule books for SysML synthesis, Rust/Verus repair, and other high-assurance systems, including seL4-based firewall architectures [31], as well as repair heuristics inspired by VeriStruct [32], cloud-scale verification, multimodal specification, and compositional reasoning for GUMBO/AGREE [29], [33]. Because rule-book bloat may occur as Meta-Rules scale across systems and HAMR artifact families, ongoing work studies budget-aware rule selection, where the SCP orchestrator classifies the input or verifier diagnostic and loads only the relevant rule book; early artifacts already explore this direction [19].

VIII. Conclusion

In this paper we introduced our SCP copilot which operates at the MBSE-to-generated-system boundary: its GUMBO FSE Rule Book learns HAMR-centric formalization and repair knowledge that connects linguistic requirement triggers to SysML v2 contract placement, HAMR-generated Slang/Scala obligations, and Logika discharge largely autoamtically. Although our evaluation uses HAMR's JVM/Slang path rather than the seL4 Microkit path, HAMR also supports Linux and seL4 targets and has been used for model-driven seL4 and avionics-style systems [1]. Thus, that learned Meta-Rules can reduce an upstream bottleneck in high-assurance HAMR workflows by lowering the cost of the formal context, and repair effort needed to synthesize verifier-checked GUMBO contracts, turning a prompt-only failure into a fully verified target-scope run. The present results remain scoped to Isolette/HAMR; broader cross-system validation, additional HAMR artifact families, and budget-aware rule selection to control rule-book growth remain future work.

Acknowledgment

This work was funded by DARPA contract FA8750-24-9-1000. The views, opinions, and/or findings expressed are those

TABLE II

REPRESENTATIVE ADJUSTED RUN METRICS FOR THE SCOPED GUMBO EVALUATION. THE ADJUSTED COLUMN IS A COMPLETE LOGGED RUN, SO THE ONE-TIME EXTRACTION/REFINEMENT EFFORT IS ALREADY INCLUDED. DOLLAR COSTS USE THE CURRENT PRICING MODEL IN THE FOOTNOTE; CACHED-TOKEN PRICE IS SET TO ZERO IN THAT MODEL.

Metric	Baseline: no Meta-Rules	Adjusted $\epsilon = 0.1$: the GUMBO FSE Rule book
Rule-book scope	None	GUMBO-centered FSE rules with toolchain-aware repair
Phase reported	Prompt-only user run	Adjusted complete run
Training / reuse source	None	One solved/golden GUMBO file/subsystem; evaluated on remaining target files
Toolchain layers exercised	SysML/GUMBO attempted; verification fails	SysML v2 placement, GUMBO, generated Slang/Scala obligations, Logika repair
Price tags (per M tokens)	input \$60; cached \$0; output \$270	same
T_{wall} (min)	33	39
N_{in}	981,770	381,624
N_{cache}	18,673,920	13,391,360
N_{out}	85,833	57,821
Logged priced tokens $N_{in} + N_{out}$	1,067,603	439,445
Illustrative C / C_{adj} ($p_{cache} = 0$)	\$82.08	\$38.51
N_{ctx}	19,741,523	13,830,805
Γ_{ctx} (tokens/min)	≈598K	≈355K
$\lvert U \rvert$ targeted units	*42*	*42*
$\lvert U_{verified} \rvert$	*2*	*42*
ρ	*4.76%*	*100%*
Θ (units/min)	*0.06*	*1.08*
K / K_{adj} (context tokens/verified unit)	≈9.87M	≈329K
$K_{\$}$ / $K_{\$}^{adj}$ (dollars/verified unit)	\$41.04	\$0.92
S (end-to-end success)	0	1

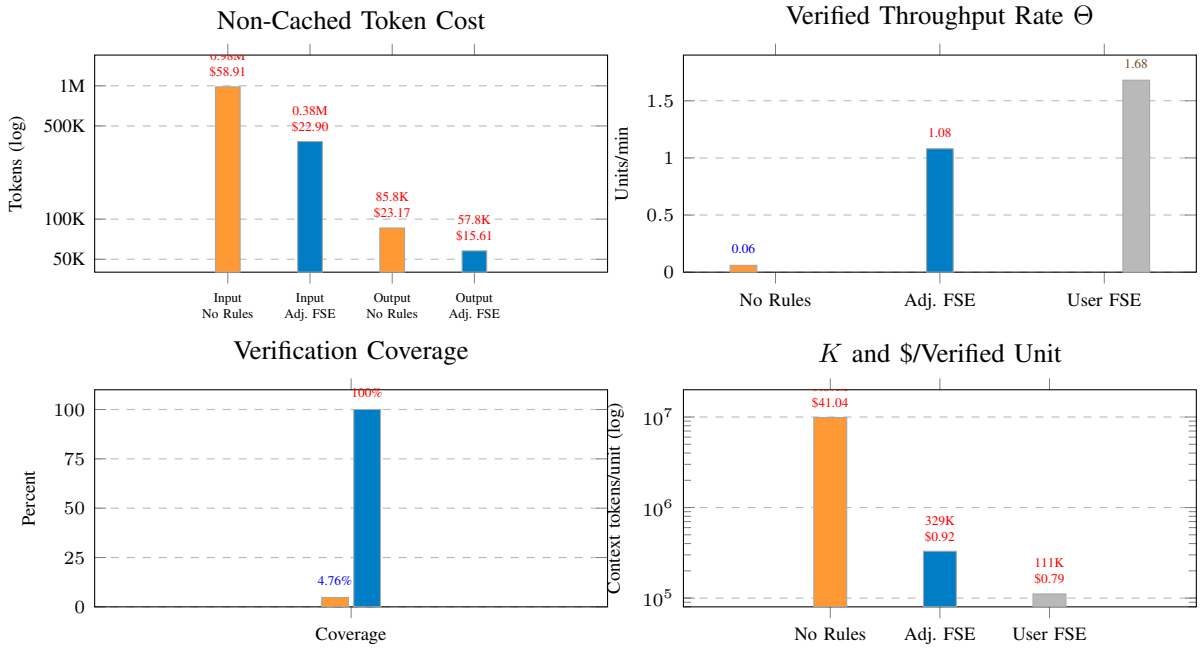

Fig. 3. Adjusted quantitative comparison for the scoped Isolette/GUMBO FSE evaluation. The chart uses the same baseline and adjusted $\epsilon = 0.1$ trace as Table II. The first panel shows only the non-cached priced token classes, input and output, with the same style of dollar tags used in the cost panel; cached tokens are still reported in Table II because they affect context size but have zero cost in the illustrative pricing model. The throughput panel reports Θ for baseline, adjusted complete run, and the later user-mode run after the rule book exists; the cost panel reports K and dollar cost per verified unit.

of the authors and should not be interpreted as representing the official views or policies of DARPA, the U.S. Department of Defense, 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] A. Achille and S. Soatto, "AI agents as universal task solvers,"

Entropy, vol. 28, no. 3, 2026. [Online]. Available: https://www.mdpi.com/1099-4300/28/3/332

[6] 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.

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

[8] 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

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

[10] 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.

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

[12] 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.

[13] 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.

[14] J. Belt, J. Hatcliff, Robby, J. Shackleton, J. Carciofini, T. Carpenter, E. Mercer, I. Amundson, J. Babar, D. Cofer, D. Hardin, K. Hoech, K. Slind, I. Kuz, and K. Mcleod, "Model-driven development for the seL4 microkernel using the HAMR framework," *Journal of Systems Architecture*, vol. 134, no. C, February 2023.

[15] 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

[16] 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.

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

[18] 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).

[19] Loonwerks, "INSPECTA Spec-Code-Proof Copilot," GitHub repository, 2026, accessed: 2026-05-06. [Online]. Available: https://github.com/loonwerks/INSPECTA-Spec-Code-Proof-Copilot/tree/main?tab=readme-ov-file

[20] D. L. Lempia and S. P. Miller, *Requirements Engineering Management Handbook*. Federal Aviation Administration, 2009, DOT/FAA/AR-08/32.

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

[22] 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.

[23] 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.

[24] 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

[25] 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

[26] 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

[27] Object Management Group (OMG), "Systems modeling language (SysML) v2 specification," , 2024, accessed: 2025-05-19.

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

[29] 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

[30] 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.

[31] 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

[32] 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

[33] 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. Springer, 2012, pp. 126–140.