Shift-Left Requirements Verification: Integrating LLMs and Formal Methods for Automotive Systems

Zipong Lim[1,2], Bozhi Wu[3⋆], Yon Shin Teo[2], Shang-wei Lin[4], and Yi Li[1]

[1] Nanyang Technological University, Singapore
{zipong001,bozhi001}@e.ntu.edu.sg, yi_li@ntu.edu.sg
[2] AUMOVIO Singapore Pte. Ltd., Singapore
{zi.pong.lim,yon.shin.teo}@aumovio.com
[3] AUMOVIO-NTU Corporate Lab, Singapore
[4] Singapore Institute of Technology, Singapore
shangwei.lin@singaporetech.edu.sg

Abstract. Requirement defects are a major source of late-stage failures in automotive systems, yet rigorous validation is rarely applied during early development. While formal methods offer strong guarantees, their adoption at the requirements level is limited by high formalization cost and expertise barriers. We present an industry-oriented, shift-left verification approach that integrates Large Language Models (LLMs) with formal methods to enable requirements-level validation. Requirements are classified and decomposed by LLMs, translated into CSP system models and assertions, and refined through a CEGAR-inspired loop using the FDR4 model checker. Validation is decomposed into requirement–assertion pairs supported by natural-language back-translations and confidence scores, preserving expert control without manual formal modeling. Domain knowledge–based validation further leverages historical defect data to identify implicit requirement gaps. We evaluate the approach on three real-world automotive case studies. Results show high automation for small-to-medium systems (80–100% synthesis success for up to ∼60 requirements), effective expert validation guided by LLM confidence estimates, and 100% detection of known historical defects alongside 22 novel gaps. The workflow completes within 10–35 minutes per project at negligible cost (<$8 per project), with limited expert effort. Our results demonstrate that LLM+formal method hybridization can provide scalable, rigorous, and industrially viable requirements-level verification, supporting practical shift-left adoption in automotive systems.

Keywords: Requirements Engineering · Requirements Verification · Formal Methods · Large Language Models · Model Checking · CSP · Automotive System

⋆ Bozhi Wu is the corresponding author.

1 Introduction

As a Tier-1 automotive supplier, **Aumovio** (formerly Continental Automotive) operates within a complex supply chain in which system requirements are primarily provided by Original Equipment Manufacturers (OEMs). Based on these documents, Aumovio develops safety-critical automotive systems that must meet stringent correctness and compliance requirements. However, at the requirements stage, no formal system model typically exists. Although requirement specifications are extensive and carefully written, they frequently contain ambiguities, inconsistencies, and implicit assumptions that remain undetected until later development phases. Without early-stage validation, such latent defects propagate downstream and often surface only during integration testing or system validation, where their correction incurs substantial cost and risk.

Automotive development traditionally follows the V-model, in which requirements engineering occurs at the earliest phase, while verification and validation activities are deferred to later stages. Decades of industrial experience show that defects discovered during integration or system testing are one to two orders of magnitude more expensive to fix than those detected during requirements engineering [3,25]. Importantly, many late-stage failures are not caused by implementation errors, but by deficiencies in the requirements themselves, including vague formulations, hidden inconsistencies, missing corner cases, and undocumented assumptions. These observations motivate a shift-left approach that brings rigorous verification techniques into the requirements phase.

Despite its importance, requirements verification faces a fundamental tension between scalability and assurance. Industrial specifications are large, informal, and evolving, making manual formalization prohibitively expensive, while safety-critical systems demand correctness guarantees that cannot be compromised.

On one hand, formal methods offer strong assurance by enabling mathematically precise reasoning about system behavior, and they have been successfully applied at the software and design levels in automotive development [14,8,6]. However, their adoption at the requirements stage remains challenging due to the *formalization barrier*: manual translation of informal text into formal specifications requires dual expertise in both domain knowledge and formal methods, which is a combination rarely available in industrial settings. Domain experts with deep system knowledge often lack formal methods training, while formal methods specialists often lack domain expertise. This skills gap makes formal verification approaches difficult to adopt at the requirements phase. On the other hand, recent works [12,2,1] leverage Large Language Models (LLMs) to automate requirements analysis, addressing the scalability of the problem. While LLMs excel at processing large volumes of text, they provide no inherent correctness guarantees: their outputs may be fluent and plausible yet semantically incorrect, making them unsuitable as the sole basis for verification in safety-critical contexts. These complementary strengths and weaknesses suggest that neither formal methods nor LLMs alone can adequately address requirements-level verification. Instead, an effective solution must combine the automation and scalability of LLMs with the soundness and rigor of formal methods.

Therefore, we propose a practical hybrid methodology that integrates LLMs with formal methods to enable scalable yet trustworthy requirements verification in industrial automotive settings. Our key idea is to treat LLMs as automation assistants, responsible for accelerating transformation tasks, while delegating all correctness-critical decisions to established formal techniques, specifically Communicating Sequential Processes (CSP) [13] and the FDR4 model checker [10].

The workflow supports *shift-left* verification where free-form natural-language requirements are automatically transformed into formal assertions, checked for correctness and consistency, and used to synthesize a verified system model before architectural or implementation decisions are finalized. Critically, the approach is human-in-the-loop: domain experts retain control over high-level validation while repetitive formal reasoning tasks are automated.

A central technical element is an iterative, counterexample-guided refinement loop inspired by CEGAR. Starting from an initial system model synthesized from validated assertions, all properties are checked in parallel using FDR4. When violations are detected, concrete counterexample traces are fed back to the LLM, which refines the model while preserving previously satisfied properties. To reduce latent requirement gaps, we integrate retrieval-augmented verification that leverages institutional knowledge such as historical defect reports, proactively identifying missing or underspecified requirements.

In summary, this paper makes the following contributions[5]:

- An end-to-end industrial workflow synthesizing formal CSP system models from free-form natural-language requirements through human-LLM-FM collaboration, employing a CEGAR-inspired refinement methodology where LLMs iteratively repair system models based on FDR4 counterexamples.
- A decomposed validation approach transforming the cognitively demanding task of validating complex system models into manageable assertion-requirement pair reviews, combined with retrieval-augmented verification exploiting historical defect knowledge.
- Empirical evaluation on real-world automotive projects demonstrating substantial reductions in expert effort and effective early defect detection.
- Successful trial deployment with an industry partner OEM, where our tool identified a better fix for a previously known issue, demonstrating practical value beyond defect detection to include solution recommendation.

2 Motivation

Automotive systems are typically developed under the V-model, where requirements engineering is expected to provide a correct foundation for all downstream tasks. In practice, however, requirement-related defects remain a major source of late-stage failures. Defects introduced during requirements engineering often propagate into implementation and are only discovered during integration testing

[5] Due to space constraints, additional details and examples can be found in the online appendix: `https://sites.google.com/view/shift-left-requirement-supp`.

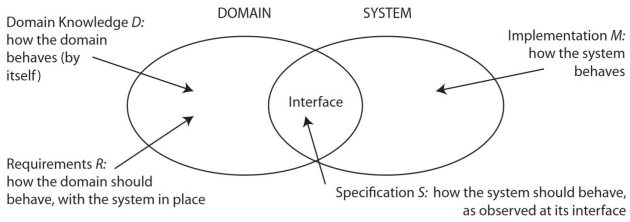

Fig. 1. Relationship between domain knowledge D, requirements R, specification S, and implementation M, adapted from [26].

or operation, where remediation costs are substantially higher. Despite informal reviews, requirements are rarely subjected to rigorous validation before being used for development, resulting in a persistent requirements validation gap.

Formal methods offer a principled means to bridge this gap by enabling mathematical validation of system behavior against requirements. However, their industrial adoption in automotive practice remains limited and restricted to later development stages, where verification is applied to artifacts such as models or code. Applying formal methods directly at the requirements level is therefore desirable for shift-left verification, but introduces several fundamental challenges.

Challenge 1: Formalization Barrier Traditional formal methods rely on manually translating natural-language requirements into formal specifications, requiring expertise in both the application domain and formal modeling languages. In industrial automotive settings, such dual expertise is rarely available: domain experts typically lack formal methods training, while formal methods specialists often lack detailed system knowledge. This mismatch confines formal verification to late development stages and prevents its effective use during requirements engineering.

Challenge 2: Implicit Knowledge Problem Requirements documents primarily capture *explicit* stakeholder intent and rarely document implicit assumptions, edge cases, or lessons learned from past failures. Consequently, even a formally verified specification may permit undesirable behaviors simply because critical domain constraints were never stated as requirements. In practice, organizations accumulate substantial domain knowledge in the form of historical defect reports, safety standards, and engineering experience, yet this knowledge remains largely tacit and disconnected from requirements-level validation.

Challenge 3: Formal Model Validity These challenges converge on a fundamental question: how can the *validity* of a formal model be established? As illustrated in Figure 1, formal specifications must reflect not only stated system requirements R but also relevant domain knowledge D. Traditional formal methods primarily address the $R \rightarrow S$ relationship, ensuring that specifications encode documented requirements, while often neglecting the $D \rightarrow S$ relationship. Many late-stage defects originate from this gap, where a specification satisfies all stated requirements but violates implicit domain constraints.

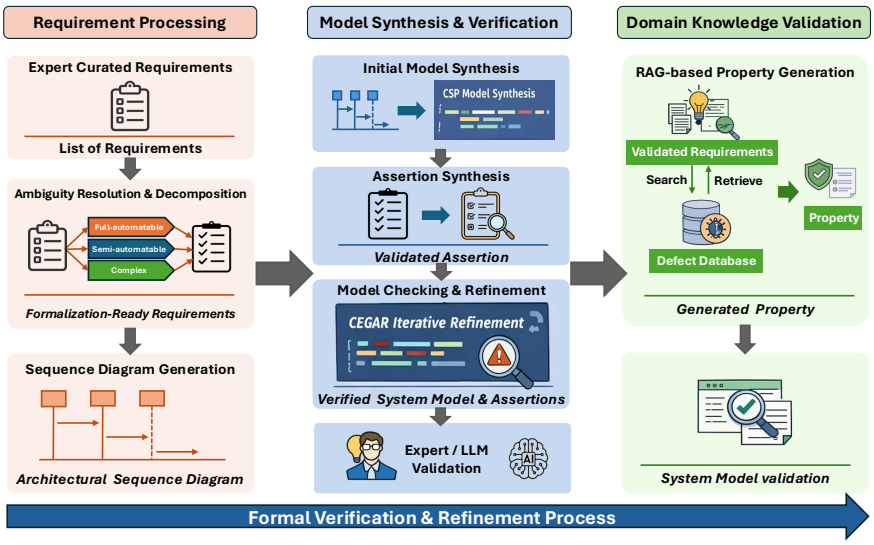

Fig. 2. Overview of the LLM+FM hybrid verification workflow.

Our Approach To address these challenges, we propose an industry-oriented LLM+FM hybrid approach following the principle: LLMs for automation, formal methods for verification, humans for validation. For challenge 1, we decompose validation into assertion-requirement pair reviews with LLM-generated back-translations, shifting human expert effort from low-level formal reasoning to high-level semantic validation. For challenge 2, retrieval-augmented validation transforms tacit organizational knowledge into actionable verification properties. For challenge 3, we address both the $R \to S$ relationship (through decomposed assertion-requirement reviews) and the $D \to S$ relationship (through retrieval-augmented validation against historical defect knowledge).

3 Methodology

Our methodology consists of three phases that incrementally construct and validate a formal system specification, with human experts involved at key decision points, as illustrated in Figure 2.

3.1 Requirement Processing

OEM requirement documents typically contain heterogeneous content, including tables, flowcharts, diagrams, and free-form text. To extract concrete requirements from such multimodal artifacts, Aumovio employs an internally developed AI-assisted toolchain based on Large Language Models (LLMs) and Vision–Language Models (VLMs), which converts complex documents into a list of textual requirements. Before formalization, however, these requirements must be further processed to ensure semantic clarity and modeling tractability.

Specifically, we employ an LLM (prompt provided in Appendix L.1) to classify each requirement according to its formalizability into three categories: *Fully Automatable (FA)* requirements that are clear and can be directly formalized; *Semi-Automatable (SA)* requirements that contain ambiguities requiring expert clarification; and *Complex (CX)* requirements that involve substantial domain complexity or multiple interacting concerns. The LLM output is subsequently reviewed and confirmed by domain experts, combining LLM automation efficiency with human validation for reliability. Ambiguities identified in SA requirements are resolved through expert clarification, while CX requirements are decomposed into simpler sub-requirements using a divide-and-conquer strategy. We now have a set of processed requirements that have been classified, disambiguated, and decomposed as needed, suitable for formal modeling.

After ambiguity resolution and decomposition, we construct an intermediate architectural abstraction in the form of a sequence diagram. At this stage, the LLM analyzes the processed requirements to identify system components, message flows, and temporal ordering constraints, and generates a textual sequence diagram (e.g., in PlantUML or Mermaid). This representation is validated by domain experts and serves as a structured bridge between natural-language requirements and subsequent formal CSP modeling. Examples of requirement processing are provided in Appendix A and Appendix B.

3.2 Model and Assertion Synthesis

Initial Model Synthesis. After requirement processing, we obtain a set of processed requirements and a validated sequence diagram that serves as an architectural scaffold for model synthesis. By explicitly identifying components, channels, and causal message flows, the sequence diagram reduces architectural ambiguity. Given the diagram together with the processed requirements, we prompt LLMs (Appendix L.2) with CSP^M syntax documentation to synthesize an initial CSP system model M_0.

Specifically, model construction follows a structured, diagram-guided workflow. First, project-specific datatypes and channels are declared to reflect the vocabulary introduced by the requirements. Next, CSP processes are synthesized to correspond to individual system components identified in the sequence diagram. Inter-component interactions are realized through channel synchronization, and the overall system behavior is defined as a parallel composition of these component processes with appropriate synchronization constraints. This ensures that the synthesized model preserves both the component boundaries and interaction semantics implied by the requirements and architectural context.

For requirements involving timing constraints (e.g., "within 200 ms"), they are abstracted into ordering constraints that are verifiable in untimed CSP. Rather than modeling explicit time, we preserve the essential causal dependencies between events using trace refinement combined with alphabet hiding. For example, in CS1 (Section 4), to capture the requirement that `relay.ON` must occur after `ignition.ON`, we define: `AlphaSpec_G3 = {| ignition, relay |}; SPEC_G3`

`= ignition.ON -> relay.ON -> SPEC_G3; assert SPEC_G3 [T= SYSTEM \ diff(Events, AlphaSpec_G3)`. Here, alphabet selection is performed by LLM based on the events referenced in each requirement, and alphabet hiding is used to restrict the refinement check to the relevant events, ensuring that the required ordering between `ignition.ON` and `relay.ON` is preserved in all system traces. During expert validation (Section 3.2), the back-translation explicitly describes which events are observed and which are hidden, enabling experts to verify that the chosen alphabet correctly captures the requirement's scope. If experts identify that relevant events are incorrectly hidden, the assertion is flagged for revision.

Assertion Synthesis. Before model checking can be performed, the processed requirements must be translated into formal assertions that can be verified against the synthesized system model. To this end, we employ an LLM to translate all processed requirements into CSP assertions in a single batch operation (prompt provided in Appendix L.3). Batch synthesis ensures consistency across assertions and alignment with the datatypes, channels, and process structure introduced during system model synthesis. The generated assertions typically fall into several categories, including trace refinement for ordering constraints, failure refinement for availability properties, deadlock freedom, and determinism.

Following assertion generation, we perform explicit syntax validation of both the initial system model M_0 and the synthesized assertion set using the FDR4 syntax checker. Reported syntax errors, including undefined channels, datatype mismatches, and malformed expressions, are collected and provided to the LLM together with the relevant code fragments and diagnostic messages. The LLM is instructed to correct these errors, and this syntax refinement loop is repeated until all errors are resolved or a maximum of 20 attempts is reached, yielding a syntactically valid system model M_1 and assertion set $\mathcal{A}_1$.

CEGAR-Inspired Iterative Model and Assertion Refinement. After a syntactically valid system model M_1 and assertion set $\mathcal{A}_1$ are obtained, we refine both the model and assertions using a counterexample-guided refinement process inspired by CEGAR. To this end, all assertions $a_i \in \mathcal{A}_1$ are first checked against the current model in parallel, with a timeout of 60 seconds per assertion. Based on the verification results, the assertions are classified into three groups: $\mathcal{A}_{\text{pass}}$, which contains satisfied assertions; $\mathcal{A}_{\text{fail}}$, which includes violated assertions together with their counterexample traces; and $\mathcal{A}_{\text{timeout}}$, which consists of assertions that exceed the timeout due to state-space explosion.

Assertions in $\mathcal{A}_{\text{pass}}$ require no further action and are treated as invariants that must be preserved throughout subsequent refinement iterations. In contrast, for assertions in $\mathcal{A}_{\text{fail}}$, we apply a CEGAR-inspired global refinement strategy that processes all failures simultaneously in order to avoid local optimization traps, as detailed in Algorithm 1. In each refinement step, the LLMRefine function constructs a prompt (Appendix L.4) containing the current model, all failed assertions with their counterexamples and corresponding requirements, the set of passing assertions as constraints, and explicit refinement instructions. The LLM proposes updates to the model and the assertions (only when necessary), in

Algorithm 1 Counterexample-Guided System Model and Assertion Refinement

Input: Model M, assertions $\mathcal{A}$
Output: Refined M^* and $\mathcal{A}^*$, or stalemate report
$iteration \leftarrow 0$, $no\_progress \leftarrow 0$, $history \leftarrow \emptyset$
while $iteration <$ MAX_ITER **do**
  $(\mathcal{A}_{\text{pass}}, \mathcal{A}_{\text{fail}}, \mathcal{A}_{\text{timeout}}) \leftarrow$ ParallelCheck$(M, \mathcal{A})$
  if $\mathcal{A}_{\text{fail}} = \emptyset \wedge \mathcal{A}_{\text{timeout}} = \emptyset$ **then**
    return $(M, \mathcal{A})$
  end if
  $pattern \leftarrow$ ComputePattern$(\mathcal{A}_{\text{pass}}, \mathcal{A}_{\text{fail}})$
  if $pattern \in history$ **then**
    return Stalemate("Oscillation", M, $\mathcal{A}$, $\mathcal{A}_{\text{fail}}$)
  end if
  $history \leftarrow history \cup \{pattern\}$
  $(M', \mathcal{A}') \leftarrow$ LLMRefine$(M, \mathcal{A}, \mathcal{A}_{\text{fail}}, \mathcal{A}_{\text{pass}})$
  $(\mathcal{A}'_{\text{pass}}, \mathcal{A}'_{\text{fail}}, \_) \leftarrow$ ParallelCheck$(M', \mathcal{A}')$
  if $|\mathcal{A}'_{\text{pass}}| \leq |\mathcal{A}_{\text{pass}}|$ **then**
    $no\_progress \leftarrow no\_progress + 1$
  else
    $no\_progress \leftarrow 0$
  end if
  if $no\_progress \geq$ THRESHOLD **then**
    return Stalemate("No progress", M', $\mathcal{A}'$, $\mathcal{A}'_{\text{fail}}$)
  end if
  $M \leftarrow M'$, $\mathcal{A} \leftarrow \mathcal{A}'$, $iteration \leftarrow iteration + 1$
end while
return Stalemate("Max iterations", M, $\mathcal{A}$, $\mathcal{A}_{\text{fail}}$)

a structured diff format. These updates are automatically parsed and applied to produce a refined model M' and assertion set $\mathcal{A}'$. After each update, all assertions are re-verified to detect potential regressions introduced by the refinement. This iterative refinement process continues until one of four termination conditions is met: (1) Success, when all assertions pass; (2) Oscillation, when the same pass–fail pattern recurs; (3) No progress, when the number of passing assertions does not increase for N consecutive iterations; or (4) Maximum iterations, when the limit is reached. Concrete refinement examples are provided in Appendix C.

Assertions in $\mathcal{A}_{\text{timeout}}$ are handled separately through LLM-guided state-space reduction. For each such assertion, we guide the LLM (Appendix L.4) to simplify the model by removing processes not involved in the assertion alphabet, abstracting datatypes to smaller value sets, eliminating internal events not visible to the property, and applying process inlining. The resulting simplified model is then re-verified against the assertion, with up to 10 iterations with increasingly aggressive simplifications. Additional details of the procedure are provided in Appendix D and L.5.

Expert Validation and Golden Standard Establishment. Once the refinement loop yields a system model satisfying all verifiable assertions, we validate

assertion correctness through expert review. While refinement guarantees that the model satisfies the assertions ($M \models \mathcal{A}$), it does not ensure that the assertions faithfully capture the intent of the original requirements ($\mathcal{A} \equiv \mathcal{R}$). This validation addresses this gap through human–LLM collaboration.

For each verified assertion $a_i \in \mathcal{A}_{\text{verified}}$, experts review a structured package comprising the original requirement, the corresponding CSP assertion, an LLM-generated natural-language back-translation explaining what the assertion verifies (Appendix L.6), an LLM confidence score (0-1) with rationale (Appendix L.7), and the verification evidence. For example, from CS1 requirement G-2: "System shall provide four grille shutter opening degrees" was formalized as `SPEC_G2 = (actuator.FULL_OPEN -> SPEC_G2) [] (actuator.LEVEL_1 -> SPEC_G2) [] ...` with back-translation "This verifies all four degrees can be commanded" and confidence 0.85. Additional examples are in Appendix E.

Experts assess each assertion–requirement pair for semantic alignment, completeness, abstraction adequacy, and case coverage. Assertions may be *Approved*, *Modified* (triggering re-verification), or *Rejected*. After review, all approved assertions are consolidated into the golden standard specification $\mathcal{A}_{\text{golden}}$, together with a traceability matrix, coverage report, and documented limitations.

By decomposing validation into requirement–assertion pairs rather than validating the system model monolithically, this process enables clear semantic traceability, supports incremental expert review, and provides a robust and scalable validation suitable for industrial practice.

Final System Model Refinement. With $\mathcal{A}_{\text{golden}}$ established, we perform final model refinement to ensure it satisfies this authoritative set. We re-run parallel verification of M^* against all $\mathcal{A}_{\text{golden}}$. The final refinement follows Algorithm 1 but with: **Golden standard priority**, under which all assertions are treated as inviolable; **No assertion modifications**, meaning that only the model may be refined; and **Stricter stalemate criteria**. When stalemate occurs, diagnostic reports are escalated to experts. Upon successful completion, this phase produces final verified model M_{final} with formal guarantee $\forall a_i \in \mathcal{A}_{\text{golden}}, M_{\text{final}} \models a_i$.

Two-Phase Refinement Design. A key design decision is the separation between *iterative refinement* and *golden standard establishment*. During iterative refinement, both the model and assertions may be modified to resolve verification failures. This flexibility is intentional as early-stage assertions may contain formalization errors or overly strict interpretations that do not reflect requirement intent. However, this flexibility introduces risks of assertion weakening, where the loop converges by relaxing properties rather than correcting the model. To mitigate this, we impose a strict two-phase structure: (1) during iterative refinement, modifications are permitted but all changes are logged and presented to experts; (2) once the golden standard is established through expert validation, assertions become immutable and only model refinement is permitted. This separation ensures that any assertion modifications undergo explicit human review before being accepted as authoritative, preventing silent semantic drift.

3.3 Domain Knowledge-Based Validation

This phase addresses a critical limitation: even when a system model satisfies all explicitly stated requirements, it may still exhibit undesirable behaviors or fail to account for implicitly assumed scenarios. To mitigate this risk, we perform domain knowledge-based validation that leverages institutional knowledge, particularly historical defect reports, to identify requirement gaps.

RAG-Based Approach. We adopt a retrieval-augmented generation (RAG) workflow. Given the current requirements and the verified system model, relevant historical defect databases are retrieved based on semantic similarity. An LLM then generates candidate properties intended to prevent the recurrence of similar defects, formalizes them into CSP assertions, and checks them against the system model. Properties that fail verification are treated as potential requirement gaps.

Defect Database. The defect base is constructed from historical defect reports, lessons learned documents, and test failure reports. Each entry is embedded using pre-trained language models and stored in a vector database. To retrieve relevant defects, we construct a query from current requirements and retrieve top-K most similar defect reports ($K = 10$) using cosine similarity with relevance filtering (threshold > 0.7). Defect database structure is available in Appendix F.

Property Generation and Validation. Retrieved defect reports are fed to an LLM (Prompt in Appendix L.8) with current requirements and golden standard assertions to generate candidate properties $\mathcal{P}_{\text{candidate}}$. Each candidate property includes a natural-language description, an explanation of which historical defect(s) motivated it, an assessment of whether it is already covered by existing requirements, and a severity rating. Domain experts review these properties and produce a validated set $\mathcal{P}_{\text{validated}}$.

Verification and Gap Analysis. Each property $p_i \in \mathcal{P}_{\text{validated}}$ is checked against M_{final} using FDR4. For each gap, we generate a report containing: executive summary, failed property description, historical context, counterexample analysis with FDR4 trace and LLM explanation (Prompt available in Appendix L.9), root cause assessment, and recommended actions. Example gap reports are provided in Appendix G.

Stakeholder Decision-Making. Based on these reports, stakeholders decide whether to *accept and fix* the gap, *accept with alternative mitigation*, *reject* it as not applicable, or *defer* the decision. When gaps are accepted and addressed, new requirements are formalized and added to the golden standard specification, the system model is refined, and domain knowledge-based validation is re-executed. The process terminates when all high-priority gaps are resolved, no new gaps emerge, or practical constraints require freezing the specification.

Overall, this phase does not replace human validation but restructures it into a focused and efficient workflow. By grounding gap detection in historical evidence and validating candidate properties through formal verification, the approach systematically exposes implicit requirement gaps and complements requirements-level verification in industrial settings.

Table 1. Overview of Real-world Automotive Cases. (Details in Appendix H)

Case	#Req	Complexity	Domain	Key Characteristics
CS1	23	Low	Active Grille Shutter Control	Actuator control, sensor integration, power state transitions, diagnostic fault code generation
CS2	56	Medium	Sunroof Auto-Identification	Automatic hardware detection, state-machine-based configuration, CAN communication, non-volatile memory persistence
CS3	83	High	Remote Services Gateway	Multi-ECU distributed communication, remote control protocol compliance, multi-step request–response flows with security handshaking

4 Evaluation

4.1 Experimental Setup

We evaluate our methodology through case studies on three real-world automotive projects, assessing its feasibility, correctness, defect discovery capability, and industrial practicality. Specifically, we address the following four research questions: **RQ1:** Can LLMs automatically generate correct CSP system models and assertions from natural language requirements within practical iteration limits? **RQ2:** Do LLM-generated assertions accurately capture requirement intent, and can LLM confidence scores predict formalization quality? **RQ3:** Can domain knowledge-based verification identify real requirement gaps by leveraging historical failure modes? **RQ4:** Is the LLM+FM hybrid approach practical for industrial use in terms of time, cost, and expert effort?

Dataset. Table 1 summarizes three real-world automotive projects with increasing system complexity and diverse functional scope. The cases cover representative subsystem types in V-model development, including local control logic, state-machine–based configuration, and distributed protocol interactions, providing a realistic and varied evaluation setting.

Configuration. All experiments use Claude 4.5 Sonnet as the LLM and FDR4 for model checking, with iteration limits of 20 for both syntax and model-checking refinement loops. Each case study is run five times for consistency. The iteration limit of 20 was determined empirically: beyond this threshold, failed runs consistently exhibited oscillation patterns where the same assertion pass/fail configurations recurred, indicating fundamental conflicts rather than insufficient refinement attempts. Increasing the limit further did not improve success rates for complex systems.

4.2 RQ1: Synthesis Feasibility

Table 2 summarizes the synthesis success rates and average iteration counts across three automotive case studies. CS1 (23 requirements) achieved a 100% success rate in only 6.8 total iterations on average, demonstrating high reliability for small-scale systems. CS2 (56 requirements) reached an 80% success

Table 2. Success Rates and Average Iteration Counts (MC=Model Checking).

Case Study	Success Rate	Syntax Iter.	MC Iter.	Total Iter.
CS1 (23 req.)	5/5 (100%)	1.0	5.8	6.8
CS2 (56 req.)	4/5 (80%)	1.6	5.4	7.0
CS3 (83 req.)	2/5 (40%)	4.6	13.4	18.0

Table 3. Distribution of Confidence Scores for Generated Assertions.

Case Study	High (0.80–1.00)	Medium (0.60–0.79)	Low (<0.60)	Avg. Score
CS1 (23)	8 (35%)	11 (48%)	4 (17%)	0.75
CS2 (56)	24 (43%)	30 (54%)	2 (4%)	0.77
CS3 (83)	31 (37%)	50 (60%)	2 (2%)	0.77

rate; the single failed run exhibited oscillation patterns caused by conflicting assertions rather than synthesis limitations. In contrast, CS3 (83 requirements) showed a reduced success rate of 40% with an average of 18.0 total iterations, reflecting the higher complexity of its distributed multi-ECU architecture. Failed runs in CS3 consistently entered stalemates where violated assertions no longer decreased despite further refinement. More iterations further highlight the increasing difficulty of large-scale synthesis.

Answer to RQ1: LLMs can reliably synthesize correct CSP system models for small-to-medium automotive projects (up to ∼60 requirements), achieving success rates between 80% and 100%. For larger and structurally more complex projects (80+ requirements), synthesis success drops to around 40%. This reduced success rate reflects fundamental complexity rather than insufficient iterations: failed runs consistently exhibited oscillation patterns where conflicting assertions could not be simultaneously satisfied. For industrial adoption at larger scales, several mitigation strategies are viable: improved LLM capabilities, enhanced prompting strategies, hierarchical requirement decomposition, and targeted expert intervention for conflicting assertion resolution. The iterative refinement loop is essential, with most successful runs requiring 5-10 total iterations.

4.3 RQ2: Assertion Formalization Accuracy

Table 3 and Table 4 summarize the correctness of LLM-generated assertions and their relationship with confidence scores. Expert validation on CS1 shows that the majority of assertions correctly capture requirement intent: 14 out of 23 assertions (61%) required no changes, accurately modeling event ordering, state transitions, and causal relationships. An additional 22% required only minor adjustments, typically clarifying abstraction choices rather than correcting semantic errors. Only 17% required major revisions, indicating that substantially incorrect assertions are a minority.

Table 4. Expert Validation Outcomes for CS1 with LLM Confidence Scores.

Category	Count	%	R1	R2	R3	Avg. Conf.
No change required	14	61%	16	18	18	0.81
Minor changes needed	5	22%	3	1	2	0.74
Major revisions required	4	17%	4	4	3	0.59

Table 5. Results of Domain Knowledge-based Validation.

Case Study	Retrieved	Failed	Passed	Known Issues	Novel Issues
CS1	11	7	4	0/0	7
CS2	8	8	0	0/0	8
CS3	10	8	2	1/1 (100%)	7
Total	**29**	**23**	**6**	**1/1 (100%)**	**22**

Confidence scores provide supporting evidence of correctness. Assertions accepted without modification exhibit the highest average confidence (0.81), whereas those requiring major revisions have significantly lower confidence (0.59). For example, assertion G-2 was correctly formalized, received a high confidence score (0.85), and was unanimously approved by experts, while assertion G-16 received lower confidence (0.72) alongside an explicit LLM disclaimer that aligned with expert feedback. Detailed expert review data are provided in Appendix J.

Answer to RQ2: Most LLM-generated assertions correctly capture requirement intent, with 61% requiring no expert changes and only a small fraction requiring major revisions. LLM confidence scores correlate strongly with correctness and effectively flag assertions needing closer expert review.

4.4 RQ3: Defect Detection Capability

Table 5 summarizes the results of domain knowledge–based validation across the three case studies. In total, 29 candidate properties were generated from retrieved historical defect reports. Of these, 23 properties failed FDR4 verification, indicating that the current system model permits behaviors that could lead to defects similar to historical failures. The remaining 6 properties passed verification, meaning the system already satisfies these constraints. The 79% actionable finding rate (23/29) represents properties where formal verification exposed concrete gaps; these were subsequently reviewed by domain experts who confirmed their validity as genuine requirement deficiencies requiring attention. More importantly, the approach detected 1 out of 1 known historical defect (100%), while uncovering 22 previously undocumented gaps, demonstrating its effectiveness beyond reproducing known issues. Cross-case analysis reveals recurring requirement gaps, dominated by initialization omissions (65%) where safe or well-defined post-reset states are unspecified. Temporal ordering gaps (26%)

Table 6. Objective Metrics for Industrial Practicality Assessment. Expected costs account for synthesis success rates from Table 2.

Case Study	Requirements	Pages	Time	API Cost	Exp. Cost
CS1	23	3	10 min	$1.80 ($0.078/req)	$1.80
CS2	56	11	35 min	$7.80 ($0.139/req)	$9.75
CS3	83	9	20 min	$3.30 ($0.040/req)	$8.25
Avg.	**54**	**7.7**	**22 min**	**$4.30**	**$6.60**

arise from unconstrained execution order, while obligation-versus-capability gaps (22%) reflect missing mandatory conditions. A smaller fraction involves security and protocol gaps (9%), such as omitted handshakes or validation steps.

Known Defect Detection (CS3). In CS3, a historical integration defect occurred when control requests were issued before completion of a security handshake. While the original OEM fix relied on a fixed delay (500ms), our domain knowledge–based validation retrieved the defect and generated a property enforcing handshake completion before control operations. Verification against M_{final} revealed a violation, confirming that the system could bypass the handshake. Both Aumovio and OEM stakeholders agreed that the handshake-based property provides a superior, timing-independent solution. Details are in Appendix K.

Novel Gap Examples. Among the 22 novel gaps: *CS1 Initialization Gap:* "After power-on, the system SHALL initialize to a known safe state (grille fully open) before accepting control commands" failed, revealing that the model allowed control commands immediately after power-on without guaranteed initialization. *CS2 State Persistence Gap:* "After diagnostic clear of sunroof presence, the system SHALL re-execute detection logic on next ignition cycle" failed, showing the model allowed the system to remain in an indeterminate state.

Answer to RQ3: Domain knowledge–based validation detected 100% of known historical issues and identified 22 novel requirement gaps, achieving a 79% actionable finding rate. The approach is particularly effective at surfacing initialization sequencing and temporal ordering gaps. Beyond defect detection, it enables formally justified improvements to system behavior, demonstrating value that extends beyond reproducing known failures.

4.5 RQ4: Industrial Practicality and Efficiency

Table 6 reports objective metrics assessing the industrial practicality of our method. Across the case studies, the automated pipeline completed within 10–35 minutes of wall-clock time, even for the largest project with 83 requirements. Notably, processing time scales sub-linearly with requirement document pages, indicating that the approach remains practical as system complexity increases. API costs are consistently low, with all cases remaining well below $8 per successful run. The "Expected Cost" column accounts for synthesis failure probability

(success rates from Table 2), computed as API Cost divided by success rate. Even with potential re-runs, expected costs remain below $10 per project. In practice, failed synthesis attempts provide diagnostic information (oscillation patterns, conflicting assertions) that can guide targeted expert intervention, potentially reducing subsequent attempt costs.

Comparison with Traditional Workflows. Our approach provides: **Formal guarantees** (mathematical proof that $M_{\text{final}} \models \mathcal{A}_{\text{golden}}$), **Exhaustive checking** (FDR4 explores complete state space within computational limits), **Concrete counterexamples** (precise diagnostic traces guide fixes), and **Traceability** (full requirement-to-assertion-to-model mapping for regulatory compliance). These benefits come at minimal additional cost: <1 hour expert validation time and <8 API cost per project.

Answer to RQ4: The LLM+FM hybrid approach demonstrates strong industrial practicality: (1) automated processing completes within 10–35 minutes for projects with 23–83 requirements, (2) API costs remain low (<$8 per project, <$0.15 per requirement), (3) and processing time scales sub-linearly with document size. The approach shifts expert effort toward semantic validation rather than formal modeling, enabling practical adoption in industrial settings where domain experts typically lack formal methods training.

5 Related Work

Formal methods have a long history in safety-critical domains, with model checking [20,5,4,14,23], theorem proving [11,8,6,18], and static analysis [21,7,17,15] increasingly adopted at design and implementation levels in automotive, aerospace, and railways [22,13]. Standards such as ISO 26262 drive formal technique adoption, yet applying these methods at the requirements level remains challenging due to the manual effort for translating requirements into formal specifications.

To reduce this burden, structured specification techniques have been proposed. Controlled natural languages such as Attempto Controlled English restrict grammar and vocabulary [9] to enable deterministic parsing into logic. Requirement boilerplates and patterns, including EARS-style requirements syntax [19] and temporal property specification patterns, provide templates for expressing common behavioral constraints [3,25]. While these techniques improve consistency and analyzability, they impose strict syntactic discipline and require significant process changes and training.

Recent work has explored using LLMs for requirements engineering tasks, including requirements classification [12,2] and ambiguity detection [1]. However, these approaches typically lack formal verification guarantees, relying on LLM outputs without rigorous validation. Our work differs fundamentally by treating LLMs as automation assistants rather than verification authorities, with all outputs validated through formal model checking or expert review.

Retrieval-augmented generation (RAG) has been applied in various software engineering contexts [24,16], but its use for proactive requirement gap detection

through historical defect analysis is, to our knowledge, novel. By combining RAG with formal verification, we enable systematic validation against institutional knowledge that would otherwise remain tacit.

6 Discussion and Limitations

Dependence on LLM Quality. The effectiveness of our approach depends on the capabilities of the underlying LLM. LLMs may misinterpret domain-specific terminology or struggle in complex refinement scenarios. To mitigate this risk, LLMs are never treated as trusted verification authorities: all outputs are validated through formal checking or expert review, ensuring that errors are exposed rather than remaining latent. As LLMs improve, we expect corresponding gains in workflow performance without changes to the formal backbone. We informally evaluated several LLMs and found that Claude 4.5 Sonnet provided the best results for CSP synthesis and refinement; however, we did not conduct systematic comparisons across models. Results may vary with different LLMs, and we acknowledge this as a reproducibility consideration.

Timing Abstraction Limitations. Our approach abstracts timing constraints (e.g., "within 200ms") into ordering constraints verifiable in untimed CSP. This abstraction is appropriate for requirements where the essential property is causal ordering (e.g., "A must occur before B"), but may be insufficient for requirements where bounded response time is fundamental. Examples of requirements that may be misrepresented include: watchdog timeouts where the bound itself is safety-critical, jitter constraints requiring bounded variance, and rate-limiting requirements specifying maximum event frequency. In our case studies, timing requirements were predominantly ordering-based, and expert validation did not identify cases where the untimed abstraction materially changed requirement semantics. However, for systems with strict real-time constraints, complementary verification using timed formal methods (e.g., Timed CSP, timed automata) may be necessary. We consider integration with timed verification as future work.

Quality and Coverage of Domain Knowledge. Retrieval-augmented verification is influenced by the quality and coverage of the domain knowledge base. Historical defect records may be incomplete or biased toward certain issue classes, limiting recall. In our evaluation, missed issues were primarily due to insufficient documentation or hardware-dependent failures. Nevertheless, even partial knowledge proved valuable, identifying both known and previously undocumented high-priority gaps, and is expected to improve as organizations accumulate experience.

Human-in-the-Loop Requirement. Although the workflow significantly reduces expert effort, human involvement remains necessary for validating assertions, resolving conflicts, and handling verification stalemates. The approach is designed to support, rather than replace, expert judgment by automating low-level reasoning and providing precise diagnostics.

Scalability and Abstraction. While the method performs well for small-to-medium projects, larger systems may require hierarchical decomposition and compositional verification. The reduced success rate for CS3 (40%) indicates that fully automated synthesis has limitations at scale; however, the diagnostic information from failed attempts (oscillation patterns, conflicting assertions) provides actionable guidance for expert intervention.

Communicating Results to Non-Formal Stakeholders. The current workflow produces formal CSP models and assertions as primary outputs. For stakeholders who rely on natural-language or graphical specifications, we provide several mechanisms: (1) natural-language back-translations accompany each assertion, explaining its meaning in domain terms; (2) FDR4 can generate state transition diagrams from verified models, providing visual representations of system behavior (see Appendix I); and (3) the traceability matrix links formal assertions back to original requirements.

Reproducibility and Artifact Availability. Our evaluation uses proprietary OEM requirements, an internal defect database, and commercial LLM APIs, which limits direct reproducibility. To support replication and adaptation, we provide: complete prompts for all LLM interactions (Appendix L), detailed examples of refinement interactions and gap reports (Appendices C, G), the defect database schema with anonymized examples (Appendix F), and expert review data with inter-reviewer analysis (Appendix J). The methodology is designed to be LLM-agnostic, though performance may vary across models.

7 Conclusion

This paper presented an industry-oriented, shift-left approach to requirements-level verification for automotive systems by integrating large language models with formal methods. The proposed workflow automatically translates natural-language requirements into CSP system models and assertions, refines them through a CEGAR-inspired loop using FDR4 counterexamples, and validates correctness via expert review supported by natural-language back-translations.

Evaluation on real-world automotive case studies shows that the approach is practical and effective for small-to-medium projects, achieving high synthesis success with limited expert effort. Domain knowledge–based validation further extends requirements verification by uncovering implicit gaps using historical defect knowledge. Overall, this work demonstrates that LLM+FM hybridization can enable scalable and trustworthy requirements-level verification without major process changes, providing a concrete step toward practical shift-left adoption in industrial settings.

Acknowledgements. This research/project is supported by A*STAR under the RIE2025 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative (Award: I2501E0045), as well as cash and in-kind contributions from the industry partner(s), and the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23).

Appendix A Requirement Classification and Decomposition

A.1 Classification Categories

Requirements are categorized into three classes based on formalizability:

1. **Fully Automatable (FA):** Requirements with clear, unambiguous logic that can be directly formalized without additional clarification. These typically specify concrete behaviors, timing constraints, or state transitions.
   Example from CS1: `G-3` ("When IG ON, the ECU shall turn ON the grille shutter power relay") is classified as FA because it specifies a clear causal relationship between an event (ignition-on) and a response (relay activation).
2. **Semi-Automatable (SA):** Requirements containing ambiguities, undefined terms, or missing parameters that require user clarification before formalization.
   Example: "The system shall respond quickly to user input" is classified as SA with the flagged ambiguity: *What constitutes "quickly"? Please specify a maximum response time.*
3. **Complex (CX):** Requirements with high domain complexity, multiple interacting concerns, or implicit assumptions that benefit from decomposition into simpler sub-requirements.
   Example: "The HVAC system shall maintain cabin comfort while optimizing energy consumption" is classified as CX because it involves multiple objectives (comfort, energy efficiency) that require decomposition into measurable sub-requirements such as temperature control logic, fan speed control, and power consumption limits.

A.2 Handling Complex Requirements

For requirements classified as CX, the LLM applies a divide-and-conquer strategy to decompose each requirement into simpler, independently formalizable sub-requirements. Each sub-requirement is assigned a derived identifier (e.g., `REQ-HVAC-005.1`, `REQ-HVAC-005.2`) and is processed independently in subsequent phases.

Example Decomposition:

```
REQ-HVAC-005: The HVAC system shall maintain cabin comfort
while optimizing energy consumption.
```

Decomposed into:

```
- REQ-HVAC-005.1: The HVAC system shall maintain cabin
  temperature within 2 degree of the user-set target
  temperature.
- REQ-HVAC-005.2: The HVAC fan speed shall not exceed 80% of
  maximum when ambient temperature is within 5 degree of target
  temperature.
```

```
- REQ-HVAC-005.3: The HVAC compressor shall enter standby
  mode when cabin temperature is within degree of target for
  more than 5 minutes.
```

Appendix B Sequence Diagram Generation

B.1 Process

The LLM analyzes the complete set of requirements (after classification and decomposition) along with any available system context (e.g., component lists, interface specifications) to identify:

```
- System components/processes: Entities that send or receive messages
  (e.g., BCM, PowerManager, GS-ECU).
- Message flows: Events, commands, and data exchanges between
  components (e.g., ignition_on, set_relay(state)).
- Temporal ordering: Causal relationships and sequencing constraints
  derived from requirements.
```

B.2 Output Format

The sequence diagram is generated in a textual format such as PlantUML or Mermaid, which can be automatically rendered for visualization and easily parsed for subsequent CSP synthesis.

Example from CS1 (Active Grille Shutter Control):

```
@startuml
participant User
participant BCM
participant GS_ECU as "GS-ECU"
participant Relay
participant Actuator
participant Sensor

User -> BCM: ignition_switch(ON)
BCM -> GS_ECU: can_ignition_status(ON)
GS_ECU -> Relay: set_relay(ON)
Relay -> GS_ECU: relay_status(ON)

loop Operational Cycle
  Sensor -> GS_ECU: refrigerant_pressure(analog)
  GS_ECU -> GS_ECU: binarize_pressure()
  GS_ECU -> BCM: can_tx_pressure(analog, binary)
  GS_ECU -> GS_ECU: determine_position()
  GS_ECU -> Actuator: set_position(degree)
  Actuator -> GS_ECU: position_feedback()
end
```

```
User -> BCM: ignition_switch(OFF)
BCM -> GS_ECU: can_ignition_status(OFF)
GS_ECU -> Relay: set_relay(OFF)
Relay -> GS_ECU: relay_status(OFF)
@enduml
```

Appendix C Refinement Examples

C.1 Example Refinement Interaction from CS1

Failed Assertions: G-5, G-6, G-8, G-9, G-23, G-25, G-30

Error Analysis Summary:

```
G-5, G-6: Specifications expect environmental inputs in a
specific order, but the system allows them in any order. After
one input (e.g., temperature_condition_input.TEMP_LOW), the
spec does not allow the same event to repeat, but the system
does.

G-8, G-9: Specifications expect ignition state changes to
strictly alternate with power relay changes. However, the
system allows ignition_status.IGN_ON to occur multiple times in
succession.

G-23, G-25, G-30: After transitioning from EOL_ACTIVE to
EOL_INACTIVE, the specifications do not allow function toggles,
but the system does (because the BDC can still process these
events).
```

LLM Analysis:

```
Root Cause: The issue is NOT in the system model. The system
model correctly represents realistic behavior where:
- Environmental inputs can arrive in any order and can repeat
- Ignition status can be read multiple times without changing
- Function toggles can occur even after EOL mode changes

The issue is in the ASSERTIONS - they are overly restrictive
and do not properly model the system's ability to:
1. Repeat the same event multiple times
2. Accept events in any order
3. Continue offering certain events after state transitions

Key Issues:
- G-5, G-6 expect environmental inputs to follow a strict
  collection pattern without repetition
- G-8, G-9 expect ignition status to alternate with power relay
  changes, but ignition can be read multiple times
```

```
- G-23, G-25, G-30 prohibit function toggles after EOL mode
  deactivation, but the system correctly allows them

Decision: Modify ASSERTIONS (not system model) because:
1. System model correctly represents the requirements
2. Specifications incorrectly restrict valid system behaviors
3. Specifications need to be more permissive to accept repeated
   events and events in any order
4. Modifying the system model would make it unrealistic and
   incorrect
```

LLM Proposed Changes (Selected Examples):
Example 1: G-5 - Allowing environmental inputs in any order with repetition

```
‘‘‘ SEARCH
SPEC_G5 = SPEC_G5_CollectInputs(false, false, false, false,
false)

SPEC_G5_CollectInputs(has_dc, has_tc, has_cs, has_av, has_bv) =
  driving_condition_input?dc ->
    SPEC_G5_CollectInputs(true, has_tc, has_cs, has_av, has_bv)
  [] temperature_condition_input?tc ->
    SPEC_G5_CollectInputs(has_dc, true, has_cs, has_av, has_bv)
  [] component_status_input?cs ->
    SPEC_G5_CollectInputs(has_dc, has_tc, true, has_av, has_bv)
  [] sensor_analog_value?av ->
    SPEC_G5_CollectInputs(has_dc, has_tc, has_cs, true, has_bv)
  [] sensor_binarized_value?bv ->
    SPEC_G5_CollectInputs(has_dc, has_tc, has_cs, has_av, true)
  [] (has_dc and has_tc and has_cs and has_av and has_bv) &
    determine_opening?opening -> SPEC_G5
====
SPEC_G5 = SPEC_G5_Idle

SPEC_G5_Idle =
  driving_condition_input?dc -> SPEC_G5_Idle
  [] temperature_condition_input?tc -> SPEC_G5_Idle
  [] component_status_input?cs -> SPEC_G5_Idle
  [] sensor_analog_value?av -> SPEC_G5_Idle
  [] sensor_binarized_value?bv -> SPEC_G5_Idle
  [] determine_opening?opening -> SPEC_G5_Idle
>>>> REPLACE
‘‘‘
```

Example 2: G-8 - Allowing ignition status to be read multiple times

```
‘‘‘ SEARCH
SPEC_G8_Off =
  ignition_status!IGN_ON -> power_relay_control!POWER_ON ->
```

```
    SPEC_G8_Active
  [] ignition_status!IGN_OFF -> SPEC_G8_Off

SPEC_G8_Active =
  ignition_status!IGN_ON -> power_relay_control!POWER_ON ->
    SPEC_G8_Active
  [] ignition_status!IGN_OFF -> power_relay_control!POWER_OFF
    ->
    SPEC_G8_Off
====
SPEC_G8_Off =
  ignition_status!IGN_ON -> power_relay_control!POWER_ON ->
    SPEC_G8_Active
  [] ignition_status!IGN_OFF -> SPEC_G8_Off

SPEC_G8_Active =
  ignition_status!IGN_ON -> SPEC_G8_Active
  [] ignition_status!IGN_OFF -> power_relay_control!POWER_OFF
    ->
    SPEC_G8_Off
  [] power_relay_control!POWER_ON -> SPEC_G8_Active
>>>> REPLACE
‘‘‘
```

Example 3: G-23 - Allowing function toggles regardless of EOL mode

```
‘‘‘ SEARCH
SPEC_G23 = SPEC_G23_Inactive

SPEC_G23_Inactive =
  eol_mode_change!EOL_ACTIVE -> SPEC_G23_EOLActive
  [] eol_mode_change!EOL_INACTIVE -> SPEC_G23_Inactive
  [] shutter_function_toggle?state -> SPEC_G23_Inactive
  [] sensor_function_toggle?state -> SPEC_G23_Inactive

SPEC_G23_EOLActive =
  shutter_function_toggle?state -> SPEC_G23_EOLActive
  [] sensor_function_toggle?state -> SPEC_G23_EOLActive
  [] eol_mode_change!EOL_INACTIVE -> SPEC_G23_Inactive
  [] eol_mode_change!EOL_ACTIVE -> SPEC_G23_EOLActive
====
SPEC_G23 =
  eol_mode_change?mode -> SPEC_G23
  [] shutter_function_toggle?state -> SPEC_G23
  [] sensor_function_toggle?state -> SPEC_G23
>>>> REPLACE
‘‘‘
```

Explanation:

```
Fix Strategy Applied:
1. G-5, G-6: Changed from state-tracking specifications to
   simple processes that accept all events in any order and
   allow repetition. This matches the system's correct behavior
   where environmental inputs can arrive in any order and
   repeat.
2. G-8, G-9: Modified to allow ignition_status events to repeat
   without requiring immediate power relay changes. This
   matches realistic behavior where ignition status can be read
   multiple times. The key requirement (IGN_ON -> POWER_ON and
   IGN_OFF -> POWER_OFF) is still enforced.
3. G-23, G-25, G-30: Simplified to accept all events freely,
   which matches the system's behavior where function toggles
   can occur regardless of EOL mode state. The requirements do
   not prohibit function toggles outside EOL mode, they just
   require them to be allowed in EOL mode.

All assertion markers preserved. These changes preserve the
core requirements while removing artificial restrictions that
do not appear in the original requirements. The system model
remains unchanged because it correctly represents the
requirements.
```

Verification Result After Refinement:

```
All 7 assertions (G-5, G-6, G-8, G-9, G-23, G-25, G-30) PASSED.
All previously passing assertions remained satisfied.
```

C.2 Example Refinement Interaction from CS2

Failed Assertions: `G-3, G-10, G-20, G-21`

Error Analysis Summary:

```
G-3, G-10: Specifications expect ig1_switch.IG1_OFF to occur
first, but the implementation can perform ig1_switch.IG1_ON
initially.

G-20, G-21: Specifications expect enable_comm_sunroof.ENABLE
to occur first, but the implementation can perform
ig1_switch.IG1_OFF initially.

Pattern: Specifications enforce a specific sequence of events
(e.g., "first IG1_OFF, then ACC_OFF, then door open..."), but
in a concurrent system, these events can occur in ANY order.
```

LLM Analysis:

```
Root Cause: The specifications use sequential composition (->)
which enforces event ordering. However, the requirements
specify CONDITIONS (e.g., "when IG1 is OFF AND door is open
AND...") not SEQUENCES (e.g., "first IG1 goes OFF, then door
opens, then...").

The system model correctly allows events in any order (as it
should in a concurrent system), but the assertions incorrectly
try to enforce event ordering rather than checking state-based
conditions.

Key Issues:
- G-3, G-10 expect a specific sequence: IG1_OFF -> ACC_OFF ->
  DOOR_OPEN -> ..., but events can occur in any order
- G-20, G-21 expect: ENABLE -> IG1_OFF -> KLE_LOCK -> ...,
  but events can occur in any order
- Requirements specify when outputs should occur based on
  current state, not based on event history

Decision: Modify ASSERTIONS (not system model) because:
1. System model correctly implements concurrent behavior where
   events can occur in any order
2. Requirements specify state-based conditions, not event
   sequences
3. Assertions must use state-tracking approach instead of
   sequential composition
4. Modifying system model to enforce ordering would
   artificially constrain concurrency and not reflect real-
    world
   behavior
```

LLM Proposed Changes (Selected Examples):
Example 1: G-3 - State-tracking approach for sunroof warning

```
``` SEARCH
SPEC_G3 = SPEC_G3_WAIT

SPEC_G3_WAIT =
  ig1_switch.IG1_OFF ->
  acc_switch.ACC_OFF ->
  driver_door.DOOR_OPEN ->
  enable_comm_sunroof.ENABLE ->
  can_id_426_received.TRUE ->
  sunroof_position?pos ->
    if pos != SR_INVALID and pos != SR_CLOSE and
       pos != SR_RESERVED_E and pos != SR_RESERVED_F then
      sunroof_open_warning.TRUE -> SPEC_G3_WAIT
    else SPEC_G3_WAIT
====
SPEC_G3 = SPEC_G3_TRACK(IG1_ON, ACC_ON, DOOR_CLOSED,
```
```

```
                        DISABLE, FALSE, SR_INVALID)

SPEC_G3_TRACK(ig, acc, door, comm, can, pos) =
  ig1_switch?new_ig ->
    SPEC_G3_TRACK(new_ig, acc, door, comm, can, pos)
  [] acc_switch?new_acc ->
    SPEC_G3_TRACK(ig, new_acc, door, comm, can, pos)
  [] driver_door?new_door ->
    SPEC_G3_TRACK(ig, acc, new_door, comm, can, pos)
  [] enable_comm_sunroof?new_comm ->
    SPEC_G3_TRACK(ig, acc, door, new_comm, can, pos)
  [] can_id_426_received?new_can ->
    SPEC_G3_TRACK(ig, acc, door, comm, new_can, pos)
  [] sunroof_position?new_pos ->
    SPEC_G3_TRACK(ig, acc, door, comm, can, new_pos)
  [] sunroof_open_warning?warn ->
    if ig == IG1_OFF and acc == ACC_OFF and
       door == DOOR_OPEN and comm == ENABLE and
       can == TRUE and pos != SR_INVALID and
       pos != SR_CLOSE and pos != SR_RESERVED_E and
       pos != SR_RESERVED_F then
      (warn == TRUE) &
        SPEC_G3_TRACK(ig, acc, door, comm, can, pos)
    else
      SPEC_G3_TRACK(ig, acc, door, comm, can, pos)
>>>> REPLACE
‘‘‘
```

Example 2: G-10 - State-tracking for complete warning conditions

```
‘‘‘ SEARCH
SPEC_G10 = SPEC_G10_WAIT

SPEC_G10_WAIT =
  ig1_switch.IG1_OFF ->
  acc_switch.ACC_OFF ->
  enable_comm_sunroof.ENABLE ->
  driver_door.DOOR_OPEN ->
  can_id_426_received.TRUE ->
  sunroof_position?pos ->
    if pos != SR_INVALID and pos != SR_CLOSE and
       pos != SR_RESERVED_E and pos != SR_RESERVED_F then
      sunroof_open_warning.TRUE -> SPEC_G10_WAIT
    else SPEC_G10_WAIT
====
SPEC_G10 = SPEC_G10_TRACK(IG1_ON, ACC_ON, DOOR_CLOSED,
                          DISABLE, FALSE, SR_INVALID)

SPEC_G10_TRACK(ig, acc, door, comm, can, pos) =
  ig1_switch?new_ig ->
```

```
    SPEC_G10_TRACK(new_ig, acc, door, comm, can, pos)
  [] acc_switch?new_acc ->
    SPEC_G10_TRACK(ig, new_acc, door, comm, can, pos)
  [] driver_door?new_door ->
    SPEC_G10_TRACK(ig, acc, new_door, comm, can, pos)
  [] enable_comm_sunroof?new_comm ->
    SPEC_G10_TRACK(ig, acc, door, new_comm, can, pos)
  [] can_id_426_received?new_can ->
    SPEC_G10_TRACK(ig, acc, door, comm, new_can, pos)
  [] sunroof_position?new_pos ->
    SPEC_G10_TRACK(ig, acc, door, comm, can, new_pos)
  [] sunroof_open_warning?warn ->
    if ig == IG1_OFF and acc == ACC_OFF and
       door == DOOR_OPEN and comm == ENABLE and
       can == TRUE and pos != SR_INVALID and
       pos != SR_CLOSE and pos != SR_RESERVED_E and
       pos != SR_RESERVED_F then
      (warn == TRUE) &
        SPEC_G10_TRACK(ig, acc, door, comm, can, pos)
    else
      SPEC_G10_TRACK(ig, acc, door, comm, can, pos)
>>>> REPLACE
‘‘‘
```

Example 3: G-20 - State-tracking for answerback with KLE lock

```
‘‘‘ SEARCH
SPEC_G20 = SPEC_G20_WAIT

SPEC_G20_WAIT =
  enable_comm_sunroof.ENABLE ->
  ig1_switch.IG1_OFF ->
  lock_request.KLE_LOCK ->
  sunroof_position?pos ->
    if pos != SR_CLOSE and pos != SR_INVALID then
      (answerback_visual.TRUE -> SPEC_G20_WAIT
       [] answerback_audible.TRUE -> SPEC_G20_WAIT)
    else SPEC_G20_WAIT
====
SPEC_G20 = SPEC_G20_TRACK(IG1_ON, NO_LOCK, SR_INVALID, DISABLE)

SPEC_G20_TRACK(ig, lock, pos, comm) =
  ig1_switch?new_ig ->
    SPEC_G20_TRACK(new_ig, lock, pos, comm)
  [] lock_request?new_lock ->
    SPEC_G20_TRACK(ig, new_lock, pos, comm)
  [] sunroof_position?new_pos ->
    SPEC_G20_TRACK(ig, lock, new_pos, comm)
  [] enable_comm_sunroof?new_comm ->
    SPEC_G20_TRACK(ig, lock, pos, new_comm)
```

```
  [] answerback_visual?vis ->
    if ig == IG1_OFF and lock == KLE_LOCK and
       comm == ENABLE and pos != SR_CLOSE and
       pos != SR_INVALID then
      SPEC_G20_TRACK(ig, lock, pos, comm)
    else
      SPEC_G20_TRACK(ig, lock, pos, comm)
  [] answerback_audible?aud ->
    if ig == IG1_OFF and lock == KLE_LOCK and
       comm == ENABLE and pos != SR_CLOSE and
       pos != SR_INVALID then
      SPEC_G20_TRACK(ig, lock, pos, comm)
    else
      SPEC_G20_TRACK(ig, lock, pos, comm)
>>>> REPLACE
‘‘‘
```

Explanation:

```
Fix Strategy Applied:
The new state-tracking approach:
1. Accepts events in any order - no sequential composition
   enforcing ordering
2. Tracks current state - maintains the current value of each
   input variable
3. Checks conditions - verifies that outputs are correct when
   required conditions are met
4. Matches requirements - requirements specify conditions, not
   sequences

For G-3 and G-10: The specification now tracks the state of
all inputs (IG1, ACC, door, comm, CAN, position) and only
checks that when all conditions are true, the warning output
is TRUE.

For G-20 and G-21: The specification tracks the state and
allows answerback events to occur when the right conditions
exist, without enforcing when they must occur in the event
sequence.

This approach correctly captures the intent of the requirements
while allowing the natural concurrent behavior of the system.
All assertion markers preserved.
```

Verification Result After Refinement:

```
All 4 assertions (G-3, G-10, G-20, G-21) PASSED. All
previously passing assertions remained satisfied.
```

C.3 Example Refinement Interaction from CS3

Failed Assertions: `G-11, G-13, G-18, G-19, G-20, G-21, G-22, G-23, G-24, G-25, G-28, G-29, G-88`

Error Analysis Summary:

```
G-11, G-13, G-18, G-25, G-28, G-29: Assertions expect to
observe specific event sequences, but the system performs
events without the expected preceding events being visible
due to parallel composition or hidden events.

G-19, G-20, G-21, G-22, G-23, G-24: Assertions expect the
system to loop back after certain events, but the
implementation continues with complete sequences.
Specifications observe individual event pairs in isolation
rather than complete flows.

G-88: EVC process non-deterministically chooses between
PreACAllowed and PreACNotAllowed, but the specification expects
only PreACAllowed after ecgw_update_remote_start.ON.
```

LLM Analysis:

```
Root Cause: The assertions are incorrectly specified. They
attempt to observe partial sequences and intermediate states
that do not align with the system model structure. The system
model correctly implements requirements as complete sequences,
but assertions are written as if each requirement operates in
isolation.

Key Issues:
- G-11 expects to observe dcm_control_judgment.Success between
  server_to_dcm.SecurityTerminationReq and
  dcm_to_target_security_term, but intermediate events may be
  hidden in the composition.
- G-19/G-20 expect loops after
  dcm_to_ecgw_service30.RemoteStart, but this is part of a
    larger
  sequence.
- G-88 expects deterministic behavior, but does not account for
  EVC's non-deterministic process.

Decision: Modify ASSERTIONS (not system model) because:
1. System model correctly implements complete requirement
   sequences
2. Assertions observe partial behaviors mismatched with
   composition
3. Assertions must account for hidden/tau events in parallel
   composition
```

```
4. System model modifications would break correct end-to-end
   behavior
```

LLM Proposed Changes (Selected Examples):
Example 1: G-11 - Adding intermediate event to alphabet

```
‘‘‘ SEARCH
AlphaSpec_G11 = {|server_to_dcm, dcm_to_target_security_term|}

SPEC_G11 = server_to_dcm.SecurityTerminationReq ->
           dcm_control_judgment.Success ->
           dcm_to_target_security_term -> SPEC_G11
====
AlphaSpec_G11 = {|server_to_dcm, dcm_control_judgment,
                  dcm_to_target_security_term|}

SPEC_G11 = server_to_dcm.SecurityTerminationReq ->
           dcm_control_judgment.Success ->
           dcm_to_target_security_term -> SPEC_G11
>>>> REPLACE
‘‘‘
```

Example 2: G-19 - Rewriting to observe complete sequence

```
‘‘‘ SEARCH
AlphaSpec_G19 = {|server_to_dcm, dcm_to_ecgw_service30|}

SPEC_G19 = server_to_dcm.EngineStartReq ->
           dcm_to_ecgw_service30.RemoteStart -> SPEC_G19
====
AlphaSpec_G19 = {|server_to_dcm, dcm_to_ecgw_service30,
                  ecgw_to_dcm_response30, dcm_to_server|}

SPEC_G19 = server_to_dcm.EngineStartReq ->
           dcm_to_ecgw_service30.RemoteStart ->
           ecgw_to_dcm_response30?status ->
           (if status == Success then
              dcm_to_ecgw_service30.RemoteAC_GS ->
              ecgw_to_dcm_response30?ac_status ->
              dcm_to_server!ac_status -> SPEC_G19
            else
              dcm_to_server!status -> SPEC_G19)
>>>> REPLACE
‘‘‘
```

Example 3: G-88 - Accounting for non-determinism

```
‘‘‘ SEARCH
AlphaSpec_G88 = {|ecgw_update_remote_start, evc_to_ecgw_srhv,
```

```
                  evc_to_ecgw_pacnnghv,
                  ecgw_update_forced_shutdown,
                  ecgw_ac_operable_power|}

SPEC_G88 = ecgw_update_remote_start.ON ->
           evc_to_ecgw_srhv.PreACAllowed ->
           evc_to_ecgw_pacnnghv.Normal ->
           ecgw_update_forced_shutdown.ON ->
           ecgw_ac_operable_power.ON -> SPEC_G88
====
AlphaSpec_G88 = {|ecgw_update_remote_start, ecgw_to_evc_query,
                  evc_to_ecgw_srhv, evc_to_ecgw_pacnnghv,
                  ecgw_update_forced_shutdown,
    ecgw_ac_operable_power|}

SPEC_G88 = ecgw_update_remote_start.ON ->
           ecgw_to_evc_query ->
           evc_to_ecgw_srhv.PreACAllowed ->
           evc_to_ecgw_pacnnghv.Normal ->
           ecgw_ac_operable_power.ON -> SPEC_G88
        [] ecgw_update_remote_start.ON ->
           ecgw_to_evc_query ->
           evc_to_ecgw_srhv.PreACNotAllowed ->
           evc_to_ecgw_pacnnghv?pacnnghv ->
           ecgw_ac_operable_power.OFF -> SPEC_G88
        [] ecgw_update_remote_start.ON ->
           ecgw_to_evc_query ->
           evc_to_ecgw_srhv.PreACAllowed ->
           evc_to_ecgw_pacnnghv.Abnormal ->
           ecgw_ac_operable_power.OFF -> SPEC_G88
>>>> REPLACE
‘‘‘
```

Explanation:

```
Fix Strategy Applied:
1. G-11: Added dcm_control_judgment to alphabet to make
   intermediate events observable
2. G-13, G-18: Added context events (server_to_dcm,
   target_to_dcm_control_resp) to properly sequence
   observations within complete flow
3. G-19-G-25: Rewrote to observe complete sequences rather than
   expecting loops after individual events, capturing full
   engine start and AC activation sequence
4. G-28, G-29: Added complete engine start sequence context
   leading to Service21 for proper observation timing
5. G-88: Added ecgw_to_evc_query to alphabet and made
   specification account for all possible EVC responses
   (PreACAllowed and PreACNotAllowed) to handle
   non-deterministic choice
```

```
All assertion markers preserved. Modifications enable
assertions to correctly observe actual system behavior
while validating requirements.
```

Verification Result After Refinement:

```
All 13 assertions (G-11, G-13, G-18, G-19, G-20, G-21, G-22,
G-23, G-24, G-25, G-28, G-29, G-88) PASSED. All previously
passing assertions remained satisfied.
```

Appendix D State Explosion Mitigation

D.1 State Space Reduction Strategy

For assertions that exceed the 60-second timeout due to state explosion, we employ LLM-guided state space reduction through the following steps:

1. **Identify the problematic assertion:** Extract the assertion a_i and its associated alphabet α_i.
2. **Construct simplification prompt:** The LLM receives:
   - The current system model M
   - The problematic assertion a_i and its requirement
   - The alphabet α_i (events relevant to this assertion)
   - FDR4 timeout information
   - Instruction: "Simplify this model to reduce state space while preserving the behavior relevant to this assertion. Consider: (1) reducing parallelism by removing processes not involved in α_i, (2) abstracting datatypes to smaller value sets, (3) removing internal events not visible in α_i, (4) applying process inlining to reduce synchronization overhead."
3. **Generate simplified model:** The LLM produces a reduced model M_i' tailored to verifying a_i.
4. **Re-verify:** Check a_i against M_i' with the same timeout.
5. **Iterate:** If timeout persists, repeat up to 10 times with increasingly aggressive simplifications.

Appendix E Expert Validation Interface

E.1 Validation Interface Components

For each verified assertion $a_i \in \mathcal{A}_{\text{verified}}$, the expert is presented with:

1. **Original requirement:** The natural language requirement text r_i with its unique identifier.
   Example from CS1:

```
G-2: The system shall provide four grille shutter opening degrees:
FULL OPEN, Opening Level 1, Opening Level 2, and CLOSE.
```

2. **Generated assertion:** The formal CSP code for a_i, including alphabet definitions and specification processes
   Example:

```
-- ASSERTION: G-2
AlphaSpec_G2 = {| pcm_to_actuator |}

SPEC_G2 = (pcm_to_actuator.FULL_OPEN -> SPEC_G2)
       [] (pcm_to_actuator.OPENING_LEVEL_1 -> SPEC_G2)
       [] (pcm_to_actuator.OPENING_LEVEL_2 -> SPEC_G2)
       [] (pcm_to_actuator.CLOSE -> SPEC_G2)

assert SPEC_G2 [T= SYSTEM \ diff(Events, AlphaSpec_G2)
```

3. **LLM-generated back-translation:** A natural language explanation of what the assertion verifies.
   Example:

```
This assertion verifies that the system provides all four required
grille shutter opening degrees by ensuring that each of the four
positions (FULL_OPEN, OPENING_LEVEL_1, OPENING_LEVEL_2, and CLOSE)
can be commanded to the actuator via the pcm_to_actuator channel.
The external choice operator ensures that all four options remain
continuously available. The assertion focuses solely on the output
interface to the actuator, verifying that the system's command
repertoire includes all four specified positions.
```

4. **Confidence score:** A numerical score (0-1) generated by the LLM indicating its confidence in the formalization accuracy.
   Example:

```
Confidence: 0.85

Reasoning: This assertion has relatively high confidence because
the requirement is clear and specific: provide four opening
degrees. The formalization directly models this by ensuring all
four values are possible outputs. The use of external choice is
appropriate for modeling availability of options. Limitations
are minor: (1) The assertion verifies possibility but not that
all four will actually be used during system operation (though
this may be acceptable for a 'shall provide' requirement). (2)
It doesn't verify that these are the only four positions (though
in a well-typed CSP model, this would be enforced by the datatype
definition).
```

5. **Verification evidence:** Confirmation that the assertion passed model checking against the refined system model M^*.
   Example:

```
PASSED - FDR4 verification completed in 1.2 seconds with 0
counterexamples.
```

E.2 Expert Actions

Based on the review, the expert can take one of three actions:

Approve: The assertion accurately formalizes the requirement. Mark a_i as validated and include it in the golden standard.

Modify: The assertion is close but requires corrections. The expert edits the CSP code directly. The modified assertion is re-verified against M^* to ensure it still passes.

Example modification from CS1:

> **Issue:** Assertion G-16 ("ECU shall binarize sensor values based on threshold") verifies that binarization occurs but cannot verify correctness of the threshold logic.
> **Expert decision:** Accept the assertion but add a comment noting the limitation:
> – NOTE: This assertion verifies that binarization occurs
> – (binary value is produced) but does not verify the
> – correctness of the threshold logic. Threshold correctness
> – requires separate validation through testing or
> – value-level specifications.

Reject: The assertion fundamentally misinterprets the requirement or is not verifiable in the current framework. The requirement is flagged for:

– Re-formalization (return to assertion synthesis with clarified guidance)
– Requirement refinement (if the requirement itself is ambiguous or not formalizable)
– Manual formalization (if the requirement is too complex for automated translation)

Appendix F Defect Database Structure

F.1 Data Structure

Each knowledge base entry is structured as:

```
{
  "id": "DEF-2021-0892",
  "project": "Remote Control System v2.1",
  "component": "DCM (Diagnostic Communication Manager)",
  "title": "Control requests executed before security
           handshake completion",
  "description": "During integration testing, remote control
                  requests were transmitted to target ECUs
```

```
                  before the security termination handshake
                  (SecurityTermReq/SecurityTermResp) completed.
                  This violated ISO 14230-3 protocol
                  requirements and caused unauthorized
                  operations to execute.",
  "root_cause": "Requirements G-13 to G-15 described the
                 security handshake sequence but did not
                 enforce it as mandatory. The DCM process
                 could skip the handshake under certain
                 timing conditions.",
  "resolution": "Added 500ms delay before processing control
                 requests to ensure handshake completion.
                 This is a timing-based workaround rather
                 than a principled solution.",
  "severity": "High",
  "affected_requirements": ["G-13", "G-14", "G-15"],
  "tags": ["security", "protocol-compliance", "handshake",
           "timing"]
}
```

F.2 Example Entry from CS1

```
{
  "id": "DEF-2019-0453",
  "project": "Active Grille Shutter Control Gen2",
  "component": "GS-ECU",
  "title": "Grille shutter accepts control commands before
           initialization complete",
  "description": "After vehicle power-on, the grille shutter
                  accepted position control commands before
                  completing its initialization sequence.
                  This caused undefined behavior including
                  actuator positioning errors and diagnostic
                  fault codes.",
  "root_cause": "Requirements G-0 through G-4 described the
                 control logic but did not specify
                 initialization behavior or operational
                 readiness conditions. The GS-ECU process
                 transitioned directly from power-on to
                 accepting control commands.",
  "resolution": "Added initialization state machine with
                 explicit readiness flag. System now rejects
                 control commands until initialization
                 completes.",
  "severity": "Medium",
  "affected_requirements": ["G-0", "G-1", "G-2"],
  "tags": ["initialization", "power-on", "state-machine"]
}
```

Appendix G Gap Report Examples

G.1 Complete Gap Report from CS1

Gap Report: GAP-DKV-CS1-001

Severity: Medium (Initialization Safety Risk)
Summary: Grille shutter system does not enforce initialization completion before accepting control commands, potentially causing undefined actuator behavior during cold start.

—

Failed Property:
Description: "After vehicle power-on (relay activation), the system SHALL complete initialization and transition to a known safe state (grille fully open) before accepting any position control commands."
Formal Assertion:

```
AlphaProp1 = {| pcm_to_relay.RELAY_ON,
                pcm_to_actuator,
                system_initialized |}
PROP1 = pcm_to_relay.RELAY_ON ->
        system_initialized ->
        pcm_to_actuator?degree -> PROP1
assert PROP1 [T= M_final \ diff(Events, AlphaProp1)
```

Criticality: Medium—can cause actuator positioning errors and diagnostic faults but does not directly impact safety

—

Historical Context:
Source Defect: DEF-2019-0453 (Active Grille Shutter Control Gen2)
Impact: After vehicle power-on, the grille shutter accepted position control commands before completing its initialization sequence. This caused undefined behavior including actuator positioning errors (shutter stuck at intermediate positions), diagnostic fault codes (DTC P2BAF), and inconsistent system state. The issue was discovered during cold-weather testing where initialization took longer than expected.
Historical Resolution: Added initialization state machine with explicit readiness flag. System now rejects control commands until initialization completes, which includes: sensor calibration, actuator self-test, and transition to default safe position (fully open).

—

Counterexample:

```
Trace: <bcm_to_pcm_ignition.IGN_ON,
        pcm_to_relay.RELAY_ON,
        relay_status.RELAY_ON,
        sensor_to_pcm.PRESSURE_MEDIUM,
        pcm_to_actuator.OPENING_LEVEL_1,
```

```
        actuator_feedback>
```

Explanation: After ignition turns on and the relay is activated, the system immediately begins processing sensor inputs and commanding actuator positions without any initialization phase. The trace shows the system accepting a sensor reading and commanding the actuator to OPENING_LEVEL_1 directly after power-on, with no intermediate initialization or safe-state transition.

—

Root Cause:
Missing Requirement: Requirements G-0 through G-4 describe the grille shutter control logic:

- G-0: "System determines open/close permission based on driving state and conditions"
- G-1: "System varies opening degree according to conditions"
- G-2: "System provides four opening degrees"
- G-3: "When IG ON, turn ON power relay"
- G-4: "When IG OFF, turn OFF power relay"

However, these requirements do not specify initialization behavior, operational readiness conditions, or default safe state after power-on.
Model Gap: The `GS_ECU_MAIN` process transitions directly from `GS_ECU_IDLE` to `GS_ECU_ACTIVE` upon receiving `IGN_ON` and activating the relay, without any initialization phase.

—

Recommended Actions:
New Requirement: Add requirement: "REQ-INIT-001: After power relay activation, the GS-ECU SHALL complete initialization sequence (sensor calibration, actuator self-test) and transition grille shutter to safe default position (FULL_OPEN) before accepting position control commands. Initialization SHALL complete within 2 seconds of relay activation."
Model Modification: Modify `GS_ECU_MAIN` process to include initialization state:

```
GS_ECU_MAIN = GS_ECU_IDLE

GS_ECU_IDLE =
  bcm_to_pcm_ignition?IGN_ON ->
  pcm_to_relay!RELAY_ON ->
  relay_status?RELAY_ON ->
  GS_ECU_INITIALIZING

GS_ECU_INITIALIZING =
  sensor_calibration ->
  actuator_self_test ->
  pcm_to_actuator!FULL_OPEN ->
```

```
    actuator_feedback ->
    system_initialized ->
    GS_ECU_ACTIVE

  GS_ECU_ACTIVE =
    bcm_to_pcm_ignition?IGN_OFF ->
    pcm_to_relay!RELAY_OFF ->
    relay_status?RELAY_OFF ->
    GS_ECU_IDLE
    [] sensor_to_pcm?pressure ->
       pcm_to_actuator?degree ->
       actuator_feedback ->
       GS_ECU_ACTIVE
```

Priority: Should-fix before production release (Medium severity, known issue from previous generation)
Estimated Impact: Medium—requires adding one requirement, introducing new initialization state, and adding initialization events. Estimated effort: 4-6 hours for requirements update, model refinement, and re-verification.

Appendix H Case Studies

Case Study (CS) Selection: We selected three anonymized automotive subsystem projects representing different complexity levels and functional domains:

1. **CS1 (Thermal Management System):** A control subsystem integrating sensor inputs, actuator control, and diagnostic capabilities. Requirements address power management, analog signal processing, state-based control logic, and CAN bus communication.
   Complexity: 23 requirements covering actuator control with discrete positioning levels, sensor integration with analog-to-digital conversion, power state transitions, diagnostic fault code generation, and end-of-line configuration flexibility.
2. **CS2 (Auto-Configuration System):** A runtime hardware detection and feature enablement system with cross-network communication bridging and persistent configuration storage.
   Complexity: 56 requirements spanning automatic hardware detection via network monitoring, state machine-based configuration management, LIN-to-CAN gateway integration, non-volatile memory persistence, UDS diagnostic protocol integration, and cascading enablement of dependent subsystems based on detection results.
3. **CS3 (Remote Services Gateway):** A distributed multi-ECU communication architecture for cloud-connected remote vehicle operations, including telematics integration, security validation, and remote control protocols.

Complexity: 83 requirements covering distributed system communication across multiple ECUs, remote control protocol compliance (ISO 14230-3), multi-step request-response sequences with security handshaking, state-dependent operation permissions, resource arbitration logic, and integration of comfort features with remote activation.

Appendix I State Transition Diagram Visualization

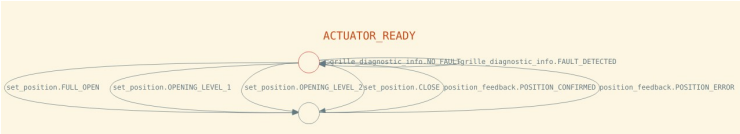

Fig. 3. State transition diagram for the `GRILLE_ACTUATOR` sub-process in the CS1 model, generated by FDR4. Red nodes indicate initial states; labeled edges represent CSP channel communications.

FDR4 provides built-in functionality to generate state transition diagrams from CSP models, enabling visual inspection of system behavior. These diagrams can be provided to stakeholders who prefer graphical representations over formal notation.

Figure 3 shows an example state transition diagram generated from the CS1 (Active Grille Shutter Control) system model. Due to the size of the complete system graph, we present the sub-process graph for the `GRILLE_ACTUATOR` component, which encompasses the `set_position`, `position_feedback`, and `grille_diagnostic_info` processes. In FDR4's visualization, each state of a process is rendered as a circular node, with transitions drawn as labelled edges corresponding to CSP events. The initial state is drawn in red, while named states (corresponding to named CSP processes) are drawn in blue, providing clear visual distinction between entry points and recursive process invocations.

This visualization capability addresses the need to communicate formal verification results to stakeholders who may not be familiar with CSP notation. While the full system graph may be too large for direct inspection, sub-process diagrams for individual components provide tractable views that support design review and stakeholder communication, facilitating broader adoption of formal methods in industrial practice.

Appendix J Expert Review Data

J.1 Complete CS1 Expert Review Results

Table 7 shows detailed expert review results for all 23 CS1 assertions, including individual reviewer decisions, LLM confidence scores, and modification notes.

Table 7. Detailed expert validation results for CS1 assertions.

Req ID	R1	R2	R3	Consensus	Conf.	Notes
G-0	M	M	M	Major	0.70	All: Missing driving state, device temperatures, vehicle device states
G-1	M	M	M	Major	0.75	All: Missing driving state, device temperatures, vehicle device states; R2: Ambiguous input-output mapping
G-2	A	A	A	Approve	0.85	-
G-3	A	A	A	Approve	0.80	-
G-4	A	A	A	Approve	0.82	-
G-7	A	A	M	Minor	0.75	R3: CLOSE level incorrectly added
G-13	A	M	A	Minor	0.88	R2: "Transmitted together" ambiguous; multiplexing assumption unclear
G-14	A	M	A	Minor	0.78	R1: Threshold unspecified; R2: "Acquire" vs "provide" mismatch
G-15	A	M	A	Minor	0.83	R2: Both analog and digital included, requirement mentions only analog
G-16	A	M	M	Major	0.72	R1: Threshold unspecified; R2: Thresholding process not explicit; R3: Threshold missing
G-17	A	A	A	Approve	0.77	-
G-18	A	M	A	Minor	0.85	R2: Analog signals cannot be transmitted via CAN; mechanism unspecified
G-19	A	M	A	Minor	0.80	R2: Bidirectional channel not in requirement (hallucination)
G-20	A	A	A	Approve	0.76	-
G-21	A	A	A	Approve	0.82	-
G-22	M	A	A	Minor	0.68	R1: Sequential composition too strict; order should be relaxed
G-23	A	M	A	Minor	0.79	R2: Separation not necessarily interleaving; additional requirement introduced
G-24	A	A	A	Approve	0.81	R1: Values unspecified
G-25	M	M	A	Major	0.71	R1: Sequential composition too strict; R2: Semantic mismatch (toggling vs affecting)
G-26	A	A	A	Approve	0.83	-
G-27	A	M	A	Minor	0.74	R1: Analog values unspecified; R2: 5-level discretization not in requirement
G-30	M	M	A	Major	0.55	R1: Need LPF channel model; R2: Cannot verify via formal methods
G-31	M	M	A	Major	0.52	R1: Need LPF channel model; R2: Coefficients undefined, requirement unclear

Legend: A = Approve, M = Modify, R = Reject, R1/R2/R3 = Reviewer 1/2/3, Conf. = LLM Confidence Score

J.2 Analysis of Inter-Reviewer Agreement

Perfect Agreement (3/3): 10 assertions (43%) received unanimous decisions from all three reviewers. These assertions had an average confidence score of 0.80 and primarily involved:

- Clear causal relationships (e.g., G-3: "When IG ON, turn ON relay")
- Capability requirements (e.g., G-2: "Provide four opening degrees")
- Simple ordering constraints (e.g., G-17: "Binarize analog values")

Partial Agreement (2/3): 11 assertions (48%) had at least one reviewer disagreeing with the majority. These assertions had an average confidence score of 0.77 and disagreements centered on:

- **Missing contextual information:** For G-13, R2 identified ambiguity in "transmitted together," noting the assertion assumes multiplexing rather than concurrent transmission.
- **Semantic mismatches:** For G-14, R2 noted "acquire" (ECU action) was translated to "provide" (sensor action), changing the requirement's focus.
- **Abstraction choices:** For G-27, R2 flagged that discretizing continuous voltage into 5 levels was not specified in the original requirement.

– **Scope interpretation:** For G-15, R2 observed both analog and digital values were included despite the requirement mentioning only analog values.

Full Disagreement (0/3): 2 assertions (9%) had all three reviewers providing different assessments. These assertions had an average confidence score of 0.73:

– **G-0 and G-1:** All reviewers identified major omissions (driving state, device temperatures, vehicle device states), but R2 additionally noted ambiguity in "vehicle devices" definition and input-output mapping assumptions.

Correlation with LLM Confidence: Assertions with perfect agreement averaged confidence 0.80, while those with partial agreement averaged 0.77, and full disagreement averaged 0.73. This progressive decrease suggests LLM confidence scores are moderately predictive of inter-reviewer agreement, with lower confidence indicating higher likelihood of disagreement or identified issues.

J.3 Common Modification Patterns

Analysis of the 11 assertions requiring minor changes revealed common patterns:

1. **Missing specification details (36%):** For G-14, G-16, G-24, and G-27, reviewers noted unspecified thresholds, values, or analog ranges that should be clarified in the requirement.
2. **Semantic mismatches (27%):** For G-14, G-18, and G-25, reviewers identified discrepancies between requirement wording and assertion semantics (e.g., "acquire" vs "provide," "affecting" vs "toggling").
3. **Abstraction assumptions (27%):** For G-13, G-15, and G-27, reviewers flagged assumptions about signal transmission, data types, or discretization levels not explicitly stated in requirements.
4. **Structural concerns (10%):** For G-7, R3 noted an incorrect addition of CLOSE level not present in the requirement.

J.4 Common Rejection Reasons

Analysis of the 4 assertions requiring major revisions revealed common reasons:

1. **Missing critical context (50%):** G-0 and G-1 omitted essential environmental and system state variables (driving state, device temperatures, vehicle device states) necessary for complete formalization.
2. **Unspecified parameters (25%):** G-16 lacked explicit threshold specification, making the binarization process non-deterministic.
3. **Hardware vs. behavioral requirements (25%):** G-30 and G-31 (LPF circuit requirements) were flagged by R2 as hardware design requirements not verifiable via formal methods, though R1 suggested modeling LPF channels as behavioral abstractions.
4. **Requirement ambiguity (25%):** G-31 was noted by R2 as fundamentally unclear regarding coefficient definition, purpose, and provider.

Appendix K CS3 Known Issue

K.1 Gap Report for Detected Known Issue from CS3

Gap Report: GAP-DKV-CS3-001

Severity: High (Protocol Compliance Risk)
Summary: Security termination handshake not enforced before remote control operations, potentially violating ISO 14230-3 protocol requirements.

—

Failed Property:
Description: "The system SHALL complete security termination handshake (send SecurityTermReq and receive SecurityTermResp) before transmitting any remote control request to target ECUs."
Formal Assertion:

```
AlphaProp1 = {| dcm_to_target.SecurityTermReq,
                target_to_dcm.SecurityTermResp,
                dcm_to_target.RemoteCtrlReq |}
PROP1 = dcm_to_target.SecurityTermReq ->
        target_to_dcm.SecurityTermResp ->
        dcm_to_target?RemoteCtrlReq -> PROP1
assert PROP1 [T= M_final \ diff(Events, AlphaProp1)
```

Criticality: High—protocol compliance violation can cause unauthorized operations and security vulnerabilities

—

Historical Context:
Source Defect: DEF-2021-0892 (Remote Control System v2.1)
Impact: During integration testing, remote control requests were transmitted before security handshake completion, violating ISO 14230-3 protocol requirements. This caused unauthorized operations to execute on target ECUs, requiring emergency system shutdown during testing. The defect was discovered late in integration phase, causing 3-week schedule delay.
Historical Resolution: Added 500ms fixed delay before processing control requests to ensure handshake completion. This is a timing-based workaround rather than a principled solution—it assumes handshake completes within 500ms but provides no formal guarantee.

—

Counterexample:

```
Trace: <server_to_dcm.RemoteControlReq.StartAndAC,
        internal_control_judgment.ProceedControl,
        dcm_to_target.RemoteCtrlReq.StartAndAC,
        target_to_dcm.OpusOperatumResp,
        dcm_to_server.ResultNotification.Success>
```

Explanation: After the server sends a remote control request and the internal control judgment determines to proceed, the system sends the remote control request directly to the target ECU without first executing the security termination handshake. The system then receives the operation response and notifies the server of success. The security handshake (SecurityTermReq/SecurityTermResp) is completely bypassed in this execution trace.

—

Root Cause:

Missing Requirement: Requirements G-13, G-14, and G-15 describe the security handshake sequence:

- G-13: "After proceed judgment, send security termination request"
- G-14: "Receive security termination response"
- G-15: "After receiving response, send remote control request"

However, these requirements describe a suggested sequence without enforcing it as mandatory. The current model implementation allows the DCM process to skip directly from control judgment to remote control request transmission.

Model Gap: The `DCM` process does not enforce the handshake as a mandatory precondition for remote control operations. The process structure allows branching that bypasses the handshake.

—

Recommended Actions:

New Requirement: Add requirement: "REQ-SEC-001: The system SHALL complete security termination handshake (transmit SecurityTermReq and receive SecurityTermResp) before transmitting any remote control request. This handshake is MANDATORY for all remote control operations."

Model Modification: Modify `DCM` process to enforce handshake:

```
DCM = server_to_dcm?RemoteControlReq.op ->
      internal_control_judgment?judgment ->
      (if judgment == TerminateControl then
         dcm_to_server!ResultNotification.TerminationFailure ->
    DCM
       else
         -- MANDATORY handshake before control request
         dcm_to_target!SecurityTermReq ->
         target_to_dcm?SecurityTermResp ->
         dcm_to_target!RemoteCtrlReq.op ->
         target_to_dcm?OpusOperatumResp ->
         dcm_to_server!ResultNotification.Success -> DCM)
```

Priority: Must-fix before production release (High severity, known field failure mode, protocol compliance requirement)

Estimated Impact: Low—requires adding one requirement and modifying one process. Estimated effort: 2-4 hours for requirements update, model refinement, and re-verification.
Advantages over Historical Fix: The recommended handshake-based solution is superior to the original 500ms time-delay fix because:

- Provides formal guarantee of protocol compliance independent of timing assumptions
- Eliminates race conditions that could occur if handshake takes longer than 500ms
- Makes the security requirement explicit and verifiable
- Aligns with ISO 14230-3 protocol specification

Appendix L Synthesis Prompts

L.1 Requirement Classification Prompt

System Prompt

```
You are a requirements engineering expert specializing in automotive
system requirements analysis and formal methods.

Your task is to classify automotive system requirements based on their
formalizability for CSP model checking. Analyze each requirement and
categorize it into one of three classes:

1. Fully Automatable (FA): Requirements with clear, unambiguous logic
   that can be directly formalized into CSPM assertions without
   additional clarification.
   - Characteristics: Well-defined conditions, quantified parameters,
     clear temporal relationships, unambiguous terminology
   - Example: "The system shall send a response within 100ms of
     receiving a request"

2. Semi-Automatable (SA): Requirements containing ambiguities,
   undefined terms, or missing parameters that require user
   clarification before formalization.
   - Characteristics: Vague quantifiers (e.g., "quickly", "often"),
     undefined thresholds, unclear scope, missing timing constraints
   - For these requirements, identify specific ambiguities and
     formulate clarification questions

3. Complex (CX): Requirements with high domain complexity, multiple
   interacting concerns, or compound conditions that would benefit
   from decomposition into simpler sub-requirements.
   - Characteristics: Multiple conjunctions/disjunctions, nested
     conditions, cross-cutting concerns, implicit dependencies
   - For these requirements, propose a decomposition into atomic
```

```
     sub-requirements

Guidelines:
- Focus on formalizability, not domain correctness
- Be specific about what makes a requirement ambiguous or complex
- Formulate actionable clarification questions for SA requirements
- Ensure decomposed sub-requirements for CX are independently testable
- Consider CSP modeling capabilities when assessing complexity

Output format for each requirement:
- Requirement ID
- Classification: FA / SA / CX
- Reasoning: Brief explanation of the classification
- If SA: List of ambiguities and corresponding clarification questions
- If CX: Proposed decomposition into numbered sub-requirements

Maintain a professional, analytical tone suitable for requirements
engineering documentation.
```

User Prompt

```
Classify the following automotive system requirements based on their
formalizability for CSP model checking.

For each requirement, provide:
1. Requirement ID
2. Classification (FA/SA/CX)
3. Reasoning for the classification
4. If Semi-Automatable (SA): Identify ambiguities and suggest
   clarification questions
5. If Complex (CX): Propose a decomposition into simpler
   sub-requirements

<requirements>
{requirement_list}
</requirements>

Example output format:

REQ-001: "The system shall respond quickly to user input"
Classification: SA
Reasoning: The term "quickly" is not quantified, making it impossible
to formalize without additional information.
Ambiguities:
- What constitutes "quickly"?
Clarification questions:
- Please specify the maximum acceptable response time
  (e.g., within 100ms, within 1 second)
- Does this apply to all types of user input or specific inputs?

REQ-002: "When the vehicle speed exceeds 50 km/h and the brake pedal
```

```
is pressed, the system shall activate ABS within 10ms and maintain it
until either the speed drops below 30 km/h or the brake pedal is
released, unless the system detects a fault condition"
Classification: CX
Reasoning: This requirement combines multiple conditions, temporal
constraints, and exception handling that would be clearer as separate
requirements.
Proposed decomposition:
REQ-002.1: "When vehicle speed exceeds 50 km/h and brake pedal is
pressed, the system shall activate ABS within 10ms"
REQ-002.2: "Once ABS is activated, it shall remain active while brake
pedal is pressed and vehicle speed is at or above 30 km/h"
REQ-002.3: "ABS shall deactivate when vehicle speed drops below 30
km/h"
REQ-002.4: "ABS shall deactivate when brake pedal is released"
REQ-002.5: "ABS shall not activate or shall deactivate immediately
when a fault condition is detected"

Provide your classification and analysis for all requirements in the
list above.
```

L.2 CSP Model Synthesis Prompt

System Prompt

```
You are a formal methods expert specializing in CSP (Communicating
Sequential Processes) and CSPM notation for the FDR4 model checker.

Your task is to synthesize a complete CSPM system model from
automotive system requirements and a validated sequence diagram. The
model should accurately capture system behavior, component
interactions, and communication patterns suitable for formal
verification.

You will be provided with:
1. A validated sequence diagram in PlantUML format showing component
   interactions and message flows
2. Classified automotive system requirements with unique identifiers
3. CSPM syntax reference documentation

Generate a complete CSPM model with the following structure:

1. DATATYPES: Define all necessary data types for signals, states,
   parameters, and messages referenced in the sequence diagram and
   requirements

2. CHANNELS: Declare all communication channels between processes,
   ensuring they match the message flows in the sequence diagram

3. COMPONENT PROCESSES: Define one process for each system component
```

```
   (ECU, sensor, actuator, etc.) that captures its behavior and
   interactions

4. SYSTEM COMPOSITION: Compose all component processes using
   appropriate parallel composition operators with proper
   synchronization

Guidelines:
- Use meaningful names that directly correspond to the sequence
  diagram
- Ensure channel synchronization matches the interaction patterns
- For timing constraints (e.g., "within 200ms"), abstract to event
  ordering constraints since CSPM does not model real-time directly
- Avoid numeric ranges like {0..255}; use discrete enumerated types
  instead to prevent state space explosion
- Include comments linking processes and channels to specific
  requirements
- Use simple section headers (e.g., -- DATATYPES) without decorative
  lines
- All comments must start with -- (two hyphens)
- Never use more than 4 consecutive = characters in comments
- Ensure the model is syntactically correct for FDR4
- Keep the model modular and readable

CRITICAL FORMATTING RULES:
- NEVER use decorative comment lines with repeated characters
- Use simple headers like -- SECTION NAME
- All comments MUST start with --
- Avoid excessive decoration in comments

Output format:
Provide the CSPM code enclosed in triple backticks with the cspm
language specifier:

‘‘‘cspm
{code}
‘‘‘

Do not include explanations outside the code block unless there are
ambiguities that need clarification.
```

User Prompt

```
Generate a complete CSPM system model based on the following inputs:

<sequence_diagram>
{sequence_diagram}
</sequence_diagram>

<requirements>
{requirements}
```

```
</requirements>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Create a CSPM model that:
- Defines all necessary datatypes for signals, states, and parameters
- Declares channels for all component interactions shown in the
  sequence diagram
- Implements one process per system component with behavior matching
  the sequence diagram
- Composes all processes using appropriate parallel composition with
  proper synchronization
- Includes comments linking model elements to specific requirements

IMPORTANT FORMATTING:
- DO NOT use decorative comment lines with = characters
- Use simple section headers like -- DATATYPES
- NEVER use more than 4 consecutive = in any comment
- All comments must start with --

IMPORTANT MODELING:
- Avoid numeric ranges like {0..255}
- Use discrete enumerated types instead (e.g., datatype Byte = B0 |
  B1 | B255)
- Abstract timing constraints to event ordering
- Ensure all channels are properly synchronized

Structure your model with these sections:
-- DATATYPES
-- CHANNELS
-- COMPONENT PROCESSES
-- SYSTEM COMPOSITION

Format your response with the CSPM code enclosed in triple backticks:

‘‘‘cspm
[insert generated cspm code here]
‘‘‘
```

L.3 Batch Assertion Synthesis Prompt

System Prompt

```
You are a formal methods expert specializing in CSP (Communicating
Sequential Processes) and CSPM assertions for the FDR4 model checker.

Your task is to translate automotive system requirements into formal
CSPM assertions that verify properties of an existing CSPM system
```

```
model.

You will be provided with:
1. A complete CSPM system model containing datatypes, channels,
   processes, and system composition
2. A list of requirements to formalize (Fully Automatable and
   clarified Semi-Automatable requirements)
3. CSPM syntax reference documentation

For each requirement, generate a CSPM assertion using the appropriate
verification technique:

Assertion Types:
1. Trace Refinement [T=]: Verifies that the system exhibits only the
   allowed sequences of events
   - Use for: Ordering constraints, behavioral sequences
   - Example: "Event A must occur before Event B"

2. Failures Refinement [F=]: Verifies that the system can refuse only
   the allowed sets of events
   - Use for: Availability constraints, choice properties
   - Example: "After event A, event B must be available"

3. Deadlock Freedom: Verifies the system cannot reach a state with no
   enabled events
   - Use for: Liveness properties
   - Syntax: assert SYSTEM :[deadlock free [F]]

4. Determinism: Verifies the system has unique behavior
   - Use for: Predictability requirements
   - Syntax: assert SYSTEM :[deterministic [F]]

Guidelines:
- DO NOT redefine channels or datatypes - use those from the system
  model
- Define assertion-specific alphabets (AlphaSpec) containing only
  events relevant to each requirement
- Use alphabet hiding: SPEC [T= SYSTEM \ diff(Events, AlphaSpec)
  This allows the system to perform other events while verifying the
  required behavior
- For timing constraints (e.g., "within 200ms"), abstract to event
  ordering since CSPM does not model real-time
- Include the requirement ID and text as comments before each
  assertion
- Wrap each assertion with extraction markers:
  -- ASSERTION_START: {req_id}
  -- ASSERTION_END: {req_id}
- Keep assertions focused and independent
- Use meaningful names for alphabets and specification processes
```

```
Output Format:
For each requirement, generate:

-- ASSERTION_START: {req_id}
-- {requirement_text}
AlphaSpec_{req_id} = {| relevant_channels_and_events |}
SPEC_{req_id} = [specification_process_definition]
assert SPEC_{req_id} [T= SYSTEM \ diff(Events, AlphaSpec_{req_id})
-- ASSERTION_END: {req_id}

CRITICAL FORMATTING:
- All comments must start with --
- Never use more than 4 consecutive = characters
- Use simple headers without decorative lines
- Preserve assertion markers exactly as shown

Ensure all assertions are syntactically correct for FDR4 and reference
only channels and datatypes defined in the system model.
```

User Prompt

```
Generate CSPM assertions to formally verify the following automotive
system requirements against the provided system model.

<system_model>
{system_model}
</system_model>

<requirements>
{requirements}
</requirements>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

For each requirement:
1. Identify the property to be verified (ordering, availability,
   liveness, determinism)
2. Select the appropriate assertion type (trace refinement, failures
   refinement, deadlock freedom, determinism)
3. Define the alphabet (AlphaSpec) containing only events relevant to
   this requirement using channels from the system model
4. Create a specification process (SPEC) that captures the required
   behavior
5. Write the assertion statement using alphabet hiding to allow other
   system events

IMPORTANT:
- DO NOT redefine channels or datatypes - reference those from the
  system model
```

```
- Use alphabet hiding: SPEC [T= SYSTEM \ diff(Events, AlphaSpec)
- For timing constraints, abstract to event ordering
- The system model already defines Events as the complete alphabet
- The system process is named SYSTEM

Output format for each requirement:

-- ASSERTION_START: {req_id}
-- {requirement_text}
AlphaSpec_{req_id} = {| channel1, channel2.value, ... |}
SPEC_{req_id} = [process_definition]
assert SPEC_{req_id} [T= SYSTEM \ diff(Events, AlphaSpec_{req_id})
-- ASSERTION_END: {req_id}

Ensure all assertions:
- Reference existing channels and datatypes from the system model
- Include assertion markers for extraction
- Are syntactically correct for FDR4
- Focus on verifying the specific requirement

Format your response with assertions enclosed in a code block:

‘‘‘cspm
[generated assertions]
‘‘‘
```

L.4 Counterexample-Guided Refinement Prompt

System Prompt

```
You are a formal methods expert specializing in CSP (Communicating
Sequential Processes) model refinement and debugging.

Your task is to refine a CSPM system model based on failed assertions
and their counterexamples from FDR4 model checking. The goal is to fix
the model so that all assertions pass while preserving the correctness
of currently passing assertions.

You will be provided with:
1. The current CSPM system model
2. Failed assertions with their counterexamples and requirement texts
3. A list of passing assertion IDs that must remain satisfied
4. CSPM syntax reference documentation

Analysis Process:

1. Counterexample Analysis:
   - Examine each counterexample trace to understand the failure
   - Identify what behavior the model exhibited vs. what was required
   - Determine the root cause: missing constraints, incorrect
```

```
     synchronization, wrong process logic, incorrect assertion
     specification, etc.

2. Root Cause Identification:
   - Is the model too permissive (allows unwanted behaviors)?
   - Is the model too restrictive (prevents required behaviors)?
   - Are there synchronization issues between processes?
   - Are process definitions incorrect?
   - Is the assertion incorrectly specified or does it not match
     the requirement?

3. Solution Strategy - PRIORITIZATION:
   - **FIRST PRIORITY**: Try to fix by modifying the SYSTEM MODEL only
   - **SECOND PRIORITY**: If the assertion is demonstrably incorrect
     (e.g., doesn't match requirement, uses wrong channels/datatypes,
     has logical errors), then modify the assertion
   - Propose targeted modifications to address the root cause
   - Ensure changes are minimal and focused
   - Consider impact on passing assertions
   - Prioritize changes that fix multiple failures if possible

4. Validation:
   - Verify that proposed changes address the counterexamples
   - Check that changes don't introduce new issues
   - Ensure passing assertions remain unaffected

Guidelines:
- PREFER system model modifications over assertion modifications
- Only modify assertions when the assertion itself is clearly wrong
- Make minimal necessary changes
- Preserve the overall model structure
- Keep section headers (DATATYPES, CHANNELS, COMPONENT PROCESSES,
  SYSTEM COMPOSITION)
- Maintain all comment prefixes (--)
- Never use more than 4 consecutive = in comments
- Provide clear rationale for each change
- Use structured diff format for automated extraction
- Justify any assertion modifications explicitly

Output Format:

**ANALYSIS:**
[Explain what the counterexamples reveal about model deficiencies.
For each failed assertion, describe the root cause of the failure.
If considering assertion modification, explain why system model fix
is insufficient.]

**FIX STRATEGY:**
[Describe your approach to fixing the issue]
[Explicitly state whether you're modifying system model, assertions,
```

```
or both]
[Justify any assertion modifications - explain why the assertion is
wrong rather than the model]

**CHANGES:**
‘‘‘diff
<<<< SEARCH
[exact original code block from the model or assertion to be replaced]
====
[corrected code block]
>>>> REPLACE
‘‘‘

[Additional SEARCH/REPLACE blocks as needed for other changes]

**RATIONALE:**
[Explain why these changes fix the failed assertions while preserving
passing assertions. Describe how each change addresses specific
counterexamples. If modifying assertions, explain why system model
fix was not viable.]

CRITICAL CONSTRAINTS:
- PRIORITIZE SYSTEM MODEL MODIFICATIONS - Only modify assertions if
  absolutely necessary
- NEVER remove or modify assertion markers:
  -- ASSERTION_START: {identifier} and -- ASSERTION_END: {identifier}
- These markers are used for extraction and MUST remain exactly as
  they are
- Preserve all section headers and structure
- All comments must start with --
- Keep changes minimal and targeted
- Ensure SEARCH blocks exactly match existing code
- Ensure REPLACE blocks contain valid CSPM syntax

Modification Guidelines:
When to modify SYSTEM MODEL (preferred):
  - Behavioral issues (model doesn't match requirements)
  - Missing synchronizations or constraints
  - Incorrect process compositions
  - Model is too permissive or too restrictive

When to modify ASSERTIONS (only if necessary):
  - Assertion specification doesn't match the requirement
  - Assertion uses incorrect channels/datatypes from the model
  - Assertion logic is demonstrably wrong
  - Always explain why system model fix is insufficient
```

User Prompt

```
Refine the CSPM system model and/or assertions to fix all failed
assertions while preserving all passing assertions.
```

```
<current_model>
{current_model}
</current_model>

<failed_assertions>
{failed_assertions_with_counterexamples}
</failed_assertions>

<passing_assertions>
The following assertion IDs currently pass and MUST remain satisfied
after refinement:
{passing_assertion_ids}
</passing_assertions>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
1. Analyze each failed assertion and its counterexample trace
2. Identify the root cause of each failure (in model OR assertion)
3. Propose cohesive changes that fix all failures
4. Ensure changes do not break any passing assertions
5. Provide changes in structured diff format

MODIFICATION PRIORITY:
1. FIRST: Try to fix by modifying the SYSTEM MODEL only
2. SECOND: Only if the assertion is clearly incorrect, modify the
   assertion
3. ALWAYS: Justify any assertion modifications in your analysis

IMPORTANT:
- PREFER system model modifications over assertion modifications
- You CAN modify assertions if necessary, but explain why
- NEVER remove or modify assertion markers:
  -- ASSERTION_START: {identifier}
  -- ASSERTION_END: {identifier}
- Make minimal, targeted changes
- Preserve model structure and section headers
- All comments must start with --
- Ensure SEARCH blocks exactly match the current code
- Ensure REPLACE blocks contain syntactically correct CSPM

Decision Framework:
Modify SYSTEM MODEL if:
  - Model behavior doesn't match requirements
  - Missing or incorrect synchronizations
  - Process composition issues
```

```
Modify ASSERTIONS only if:
  - Assertion doesn't correctly capture the requirement
  - Assertion uses wrong channels/datatypes
  - Assertion logic is demonstrably wrong
  - AND you explain why system model fix won't work

Output your analysis and proposed changes in the following format:

**ANALYSIS:**
[For each failed assertion, explain what the counterexample reveals
about the deficiency and identify the root cause. If considering
assertion modification, explain why system model fix is insufficient.]

**FIX STRATEGY:**
[Describe your approach to fixing the issue]
[Explicitly state whether you're modifying system model, assertions,
or both]
[Justify any assertion modifications]

**CHANGES:**
‘‘‘diff
<<<< SEARCH
[exact code from current model or assertion to be replaced]
====
[corrected code]
>>>> REPLACE
‘‘‘

[Additional SEARCH/REPLACE blocks for other changes]

**RATIONALE:**
[Explain how these changes fix the failed assertions and why they
preserve passing assertions. Reference specific counterexamples and
requirements. If modifying assertions, explain why system model fix
was not viable.]
```

L.5 State Space Reduction Prompt

System Prompt

```
You are a formal methods expert specializing in CSP (Communicating
Sequential Processes) model optimization and state space reduction.

Your task is to simplify a CSPM system model to reduce state space
complexity while preserving the behavior relevant to a specific
assertion that is timing out during FDR4 model checking.

You will be provided with:
1. The current CSPM system model
2. A problematic assertion that times out (exceeds 60 seconds)
```

```
3. The assertion's alphabet (set of events relevant to verification)
4. CSPM syntax reference documentation

Simplification Strategies:

1. Process Elimination:
   - Identify processes that do not interact with events in the
     assertion alphabet
   - Remove or stub out these processes to reduce parallel composition
     complexity

2. Datatype Abstraction:
   - Reduce datatype value sets to only those used in the assertion
   - Replace large numeric ranges with smaller discrete sets
   - Example: If assertion only uses values {0, 1, 255}, define
     datatype Value = V0 | V1 | V255 instead of {0..255}

3. Event Hiding:
   - Hide internal events not visible in the assertion alphabet
   - Use the hiding operator (\) to abstract away irrelevant events

4. Process Inlining:
   - Inline simple processes to reduce synchronization overhead
   - Combine tightly coupled processes where appropriate

5. Alphabet Restriction:
   - Focus system composition on only the channels in the assertion
     alphabet
   - Remove unused channels from process alphabets

Guidelines:
- Preserve all causal relationships relevant to the assertion
- Ensure the simplified model exhibits the same behavior for events
  in the assertion alphabet
- Maintain syntactic correctness for FDR4
- Document all simplifications clearly
- Keep the model structure readable
- All comments must start with --
- Never use more than 4 consecutive = in comments

Output Format:

**SIMPLIFICATION STRATEGY:**
[Explain what will be simplified and why it is safe to do so without
affecting the assertion's verification. List specific processes,
datatypes, or events to be modified.]

**SIMPLIFIED MODEL:**
'''cspm
[Complete simplified CSPM model with all necessary sections]
```

```
'''

**PRESERVED BEHAVIOR:**
[Explain what behavior relevant to the assertion is preserved and how
the simplifications maintain correctness for this specific
verification task.]

CRITICAL CONSTRAINTS:
- The simplified model must be syntactically valid CSPM
- All events in the assertion alphabet must remain in the model
- Causal relationships between assertion-relevant events must be
  preserved
- The model must still be verifiable against the assertion
```

User Prompt

```
Simplify the following CSPM system model to reduce state space
complexity and enable successful verification of a timing-out
assertion.

<current_model>
{current_model}
</current_model>

<problematic_assertion>
{assertion_code}
</problematic_assertion>

<assertion_alphabet>
{assertion_alphabet}
</assertion_alphabet>

<timeout_information>
FDR4 timed out after 60 seconds when checking this assertion. The
state space is likely too large for verification.
</timeout_information>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
1. Analyze the model to identify sources of state space explosion
2. Determine which processes, datatypes, and events are relevant to
   the assertion alphabet
3. Simplify the model by:
   - Removing or stubbing processes not involved in the assertion
   - Abstracting datatypes to smaller value sets
   - Hiding internal events not in the assertion alphabet
   - Inlining simple processes where appropriate
4. Ensure the simplified model preserves behavior relevant to the
```

```
   assertion
5. Verify the simplified model is syntactically correct

IMPORTANT:
- Focus ONLY on behavior relevant to events in the assertion alphabet
- Preserve causal relationships between assertion-relevant events
- Document all simplifications clearly
- Ensure the model remains verifiable against the assertion
- All comments must start with --
- Keep the model structure clean and readable

Output your simplification in the following format:

**SIMPLIFICATION STRATEGY:**
[Explain what will be simplified: which processes will be removed,
which datatypes will be abstracted, which events will be hidden, etc.
Justify why each simplification is safe.]

**SIMPLIFIED MODEL:**
‘‘‘cspm
[Complete simplified CSPM model]
‘‘‘

**PRESERVED BEHAVIOR:**
[Explain what behavior relevant to the assertion is preserved and how
the simplifications maintain correctness for this verification task.]
```

L.6 Back-Translation Prompt

System Prompt

```
You are a formal methods expert specializing in translating formal
CSP (Communicating Sequential Processes) specifications into clear,
accessible natural language for domain experts.

Your task is to translate a formal CSPM assertion back into plain
English to help automotive domain experts validate its correctness
against the original requirement.

You will be provided with:
1. The original requirement in natural language
2. The generated CSPM assertion (formal code)
3. CSPM syntax reference documentation

Translation Guidelines:

1. Behavior Description:
   - Explain what system behavior the assertion verifies
   - Use concrete, domain-specific terminology
   - Avoid formal methods jargon where possible
```

```
   - Relate the assertion back to the original requirement

2. CSP Construct Explanation:
   - Identify key CSP operators used (e.g., [T=, [F=, ||, \, etc.)
   - Explain what each operator means in plain language
   - Describe how these operators combine to verify the requirement
   - Use analogies or examples when helpful

3. Event Observability:
   - Explicitly list which events are OBSERVED (visible) in the
     refinement check
   - Explicitly list which events are HIDDEN (abstracted away)
   - Explain WHY each observed event is relevant to the requirement
   - Explain WHY hidden events are safe to abstract away
   - Help experts verify that the alphabet selection correctly
     captures the requirement's scope
   - Flag any events that might be incorrectly observed or hidden

4. Abstraction Identification:
   - Note any timing abstractions (e.g., "within 200ms" becomes
     event ordering)
   - Identify data value abstractions (e.g., specific values vs.
     ranges)
   - Explain how continuous behaviors are discretized
   - Clarify what aspects are modeled symbolically

5. Limitation Recognition:
   - Identify aspects of the requirement not captured by the assertion
   - Note any assumptions made during formalization
   - Highlight edge cases that may not be covered
   - Be honest about verification scope

Target Audience:
- Automotive engineers and requirements engineers
- Domain experts who understand the system but not CSP
- Stakeholders who need to validate formal specifications

Output Format:

**BACK-TRANSLATION:**
[Clear, natural language explanation of what the assertion verifies.
Describe the behavior being checked in terms a domain expert would
understand.]

**KEY CSP CONSTRUCTS:**
[Explain the CSP operators and constructs used in the assertion. For
each operator, provide a plain English explanation of what it means
and why it is used.]

**EVENT OBSERVABILITY:**
```

```
[Explicitly describe:
OBSERVED EVENTS (events checked by the refinement):
- List each observed event and explain why it is relevant to the
  requirement
- Explain what behaviors involving these events are being verified
HIDDEN EVENTS (events abstracted away):
- List the events that are hidden from the refinement check
- Explain why hiding these events is acceptable for this requirement
- Note any hidden events that might affect the requirement's validity
ALPHABET JUSTIFICATION:
- Explain the rationale for the chosen alphabet
- Help experts verify that no relevant events are incorrectly hidden
- Help experts verify that no irrelevant events are incorrectly
  included]
**ABSTRACTIONS:**
[Describe any abstractions made during formalization, such as timing
abstractions, data value abstractions, or behavioral simplifications.
Explain how these abstractions relate to the original requirement.]

**LIMITATIONS:**
[Note any aspects of the original requirement that are not captured
by the assertion, any assumptions made, or any edge cases that may
not be fully verified.]

Guidelines:
- Use clear, accessible language
- Provide concrete examples from the automotive domain
- Be precise but avoid unnecessary technical detail
- Highlight both what IS verified and what IS NOT verified
- Pay special attention to alphabet selection and event hiding
- Help domain experts understand the scope and limitations of formal
  verification
```

User Prompt

```
Translate the following CSPM assertion back into plain English to help
domain experts validate its correctness.

<original_requirement>
{requirement_text}
</original_requirement>

<generated_assertion>
{assertion_code}
</generated_assertion>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
```

```
1. Explain what behavior the assertion verifies in natural language
2. Describe the key CSP constructs used and what they mean
3. Explicitly identify which events are OBSERVED and which are HIDDEN,
   explaining why the alphabet selection is appropriate
4. Identify any abstractions made during formalization
5. Note any limitations or aspects of the requirement not captured

IMPORTANT:
- Write for automotive domain experts who may not know CSP
- Be precise but accessible
- Use concrete examples from the automotive domain
- Clearly distinguish what IS verified from what IS NOT verified
- Explicitly list observed vs. hidden events so experts can validate
  the alphabet selection
- Help experts identify if any relevant events are incorrectly hidden
- Help stakeholders understand the scope of formal verification

Output your translation in the following format:

**BACK-TRANSLATION:**
[Natural language explanation of what the assertion verifies. Describe
the checked behavior in terms a domain expert would understand,
relating it back to the original requirement.]

**KEY CSP CONSTRUCTS:**
[For each CSP operator or construct used in the assertion (e.g.,
trace refinement [T=, parallel composition [| |], alphabet hiding \,
external choice [], etc.), explain what it means in plain English and
why it is used in this assertion.]

**EVENT OBSERVABILITY:**
[This section helps experts validate the alphabet selection.
OBSERVED EVENTS:
- List each event that IS checked by the refinement
- For each event, explain: (1) what it represents in the system,
  (2) why it is relevant to this requirement
HIDDEN EVENTS:
- List the events that are abstracted away (hidden from the check)
- Explain why hiding these events does not affect the requirement's
  validity
- Flag any hidden events that MIGHT be relevant and warrant review
ALPHABET JUSTIFICATION:
- Summarize the rationale for observing/hiding these specific events
- Pose questions for experts: "Are there other events that should be
  observed?" or "Should any of the hidden events be included?"
- Help experts verify that the scope matches the requirement's intent]

**ABSTRACTIONS:**
[Describe abstractions made during formalization:
- Timing: How are timing constraints (e.g., "within 200ms") handled?
```

```
- Data: How are data values or ranges abstracted?
- Behavior: How are continuous or complex behaviors simplified?
Explain how these abstractions relate to the original requirement.]

**LIMITATIONS:**
[Note any aspects of the original requirement not captured by the
assertion, any assumptions made during formalization, or any edge
cases that may not be fully verified. Be honest about the scope and
limitations of what this assertion can verify.]
```

L.7 Confidence Score Prompt

System Prompt

```
You are a formal methods expert specializing in CSP (Communicating
Sequential Processes) formalization and verification quality
assessment.

Your task is to assess your confidence in the accuracy of a CSPM
assertion formalization by evaluating how well it captures the intent
of the original natural language requirement.

You will be provided with:
1. The original requirement in natural language
2. The generated CSPM assertion (formal code)
3. CSPM syntax reference documentation

Assessment Criteria:

1. Requirement Clarity:
   - Is the requirement unambiguous and well-defined?
   - Are all terms clearly specified?
   - Are conditions and constraints explicit?
   - Is the expected behavior clearly stated?

2. Formalization Accuracy:
   - Does the assertion capture the requirement's core intent?
   - Are all conditions from the requirement represented?
   - Are the CSP operators appropriate for the requirement type?
   - Is the assertion alphabet correctly defined?

3. Alphabet Selection and Event Hiding:
   - Are the OBSERVED events sufficient to capture the requirement?
   - Are any relevant events incorrectly HIDDEN?
   - Are irrelevant events correctly hidden to focus the check?
   - Does the alphabet selection preserve the essential causal
     dependencies?
   - Could hiding certain events mask violations of the requirement?
   - Is the alphabet selection justified by the events referenced
     in the requirement?
```

```
4. Abstractions and Limitations:
   - What abstractions were necessary (timing, data, behavior)?
   - Are these abstractions acceptable for the requirement?
   - Do the abstractions preserve the essential properties?
   - What aspects of the requirement cannot be formalized in CSP?

5. Edge Cases and Completeness:
   - Are edge cases adequately covered?
   - Are there scenarios the assertion might miss?
   - Are there implicit assumptions in the formalization?
   - Could the assertion pass while the requirement is violated?

6. CSP Suitability:
   - Is CSP an appropriate formalism for this requirement type?
   - Are there fundamental limitations of CSP for this requirement?
   - Would other formal methods be more suitable?

Confidence Score Guidelines:

HIGH (0.8 - 1.0):
- Requirement is clear and unambiguous
- Formalization is straightforward with minimal abstractions
- All conditions are captured accurately
- Alphabet selection is clearly justified and complete
- All relevant events are observed; hidden events are truly irrelevant
- Edge cases are well-covered
- CSP is well-suited for this requirement type

MEDIUM (0.6 - 0.79):
- Requirement has minor ambiguities that were reasonably interpreted
- Formalization requires acceptable abstractions
- Most conditions are captured, with minor limitations
- Alphabet selection is reasonable but may warrant expert review
- Some events may be borderline relevant/irrelevant
- Some edge cases may need additional consideration
- CSP is adequate but not ideal for this requirement

LOW (0.0 - 0.59):
- Requirement has significant ambiguities
- Formalization requires substantial abstractions or assumptions
- Important conditions may be missing or incorrectly represented
- Alphabet selection may incorrectly hide relevant events
- Hidden events might mask requirement violations
- Edge cases are not well-covered
- CSP may not be suitable for this requirement type

Output Format:

**CONFIDENCE SCORE:** [numerical value from 0.0 to 1.0]
```

```
**REASONING:**
[Provide a detailed explanation of the confidence score, addressing:
- Factors that increase confidence
- Factors that decrease confidence
- Specific concerns or limitations
- Assumptions made during formalization
- Edge cases or scenarios that may not be covered
- Suitability of CSP for this requirement]

Guidelines:
- Be honest and critical in your assessment
- Identify specific issues rather than general concerns
- Explain the impact of abstractions on verification accuracy
- Note any assumptions that domain experts should validate
- Highlight areas where human review is particularly important
```

User Prompt

```
Assess your confidence in the accuracy of the following CSPM assertion
formalization.

<original_requirement>
{requirement_text}
</original_requirement>

<generated_assertion>
{assertion_code}
</generated_assertion>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
Provide a confidence score (0.0 to 1.0) indicating how confident you
are that the CSPM assertion accurately captures the requirement's
intent.

Consider the following factors:
1. Clarity and completeness of the original requirement
2. Accuracy of the formalization
3. Correctness of alphabet selection and event hiding
4. Abstractions made and their impact
5. Edge cases and completeness of coverage
6. Suitability of CSP for this requirement type

Confidence Score Guidelines:
- HIGH (0.8-1.0): Clear requirement, straightforward formalization,
  correct alphabet selection with all relevant events observed,
  minimal abstractions, well-covered edge cases
```

```
- MEDIUM (0.6-0.79): Minor ambiguities, acceptable abstractions,
  reasonable alphabet selection that may need review, most conditions
  captured
- LOW (0.0-0.59): Significant ambiguities, substantial abstractions,
  alphabet selection may incorrectly hide relevant events, important
  conditions missing or CSP unsuitable

Output your assessment in the following format:

**CONFIDENCE SCORE:** [0.0 to 1.0]

**REASONING:**
[Explain your confidence score by addressing:
- What aspects of the requirement are clearly captured?
- What ambiguities or uncertainties exist in the requirement?
- What abstractions were necessary and are they acceptable?
- What edge cases or scenarios might not be covered?
- Are there assumptions that should be validated by domain experts?
- Is CSP well-suited for this type of requirement?
- What specific concerns or limitations affect confidence?]

Be honest and specific in your assessment. Identify concrete issues
rather than general concerns. Help stakeholders understand where human
review and validation are most important.
```

L.8 Property Generation from Defects Prompt

System Prompt

```
You are a requirements engineering expert specializing in automotive
system verification and defect analysis.

Your task is to generate candidate verification properties based on
historical defect reports to identify gaps in current system
requirements and prevent similar failures.

You will be provided with:
1. A summary of current system requirements
2. A list of golden standard assertions (existing verified properties)
3. Retrieved historical defect reports (top-K most similar to current
   system)
4. CSPM syntax reference documentation

Analysis Process:

1. Defect Analysis:
   - Review each historical defect report
   - Identify the root cause and failure pattern
   - Determine what property was violated
   - Assess relevance to the current system
```

```
2. Gap Identification:
   - Compare defect-derived properties against current requirements
   - Check if golden standard assertions already cover these
    properties
   - Identify gaps where similar failures could occur
   - Look for common patterns across multiple defects

3. Property Generation:
   - Formulate properties that would have prevented the defects
   - Express properties in clear natural language
   - Ensure properties are verifiable in CSP
   - Prioritize based on defect severity and relevance

4. Criticality Assessment:
   - HIGH: Safety-critical, could cause harm or system failure
   - MEDIUM: Functional impact, degrades system performance
   - LOW: Minor issues, edge cases, or low-probability scenarios

Guidelines:
- Focus on gaps in current requirements, not redundant coverage
- Do not duplicate existing golden standard assertions
- Prioritize high-severity historical defects
- Consider common failure patterns across multiple defects
- Generate properties that are practical to verify in CSP
- Be specific about what each property verifies
- Explain the connection between defects and proposed properties

Output Format for Each Property:

**PROPERTY {N}:**

**Description:**
[Clear natural language description of the property the system should
satisfy]

**Motivation:**
[Explain which historical defect(s) motivated this property, including
defect IDs and similarity scores. Describe how this property would
have prevented the defect.]

**Coverage Assessment:**
[Analyze whether this property is already covered by existing
requirements or golden standard assertions. If not, clearly identify
the gap.]

**Criticality:**
[Rate as High/Medium/Low with justification based on potential impact
and severity]
```

```
**Formalization:**
‘‘‘cspm
AlphaProp{N} = {| relevant_events |}
PROP{N} = [specification_process]
assert PROP{N} [T= SYSTEM \ diff(Events, AlphaProp{N})
‘‘‘

Important:
- Generate only properties that address genuine gaps
- Ensure formalizations are syntactically correct CSPM
- Reference specific defect IDs and their characteristics
- Provide actionable insights for requirements improvement
```

User Prompt

```
Generate candidate verification properties based on historical defect
reports to identify gaps in current system requirements.

<current_requirements_summary>
{current_requirements_summary}
</current_requirements_summary>

<golden_standard_assertions>
{golden_standard_list}
</golden_standard_assertions>

<historical_defects>
{retrieved_defects}
</historical_defects>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
1. Analyze the historical defect reports
2. Identify properties that would have prevented these defects
3. Assess whether current requirements already cover these properties
4. Generate new candidate properties for gaps
5. Formalize each property in CSPM

IMPORTANT:
- Focus on gaps not covered by existing requirements or golden
  standard assertions
- Do not duplicate existing assertions
- Prioritize high-severity defects
- Look for common patterns across multiple defects
- Ensure properties are verifiable in CSP

For each candidate property, provide:
```

```
**PROPERTY {N}:**

**Description:**
[Natural language description of what the system should satisfy]

**Motivation:**
[Which historical defect(s) motivated this property? Include defect
IDs and similarity scores. Explain how this property would have
prevented the defect.]

**Coverage Assessment:**
[Is this already covered by existing requirements or golden standard
assertions? If yes, cite the specific requirement/assertion. If no,
explain the gap.]

**Criticality:**
[High/Medium/Low - Justify based on potential impact, safety
implications, and defect severity]

**Formalization:**
‘‘‘cspm
AlphaProp{N} = {| relevant_events |}
PROP{N} = [specification_process]
assert PROP{N} [T= SYSTEM \ diff(Events, AlphaProp{N})
‘‘‘

Generate properties that address genuine gaps and provide actionable
insights for improving system requirements.
```

L.9 Gap Report Generation Prompt

System Prompt

```
You are a requirements engineering expert specializing in automotive
system verification, gap analysis, and defect prevention.

Your task is to generate a comprehensive gap report for a failed
verification property that helps stakeholders understand the issue,
its implications, and how to address it.

You will be provided with:
1. A failed property (description and CSPM formalization)
2. The historical defect that motivated this property
3. The FDR4 counterexample trace showing the failure
4. Relevant excerpts from the current system model
5. Relevant excerpts from current requirements
6. CSPM syntax reference documentation

Report Structure:
```

```
1. Executive Summary:
   - Severity assessment (High/Medium/Low)
   - One-sentence summary of the gap
   - Quick context for decision-makers

2. Failed Property Analysis:
   - Natural language description of what was being verified
   - CSPM formalization
   - Explanation of why this property is critical

3. Historical Context:
   - Source defect information (ID, project, date)
   - Impact of the historical defect (cost, schedule, safety)
   - How the defect was resolved previously
   - Lessons learned

4. Counterexample Analysis:
   - FDR4 trace showing the violation
   - Plain English explanation of what the trace demonstrates
   - Specific scenario where the system violates the property

5. Root Cause Analysis:
   - Missing or incomplete requirements
   - Gaps in the system model
   - Why the current specification allows this violation

6. Recommended Actions:
   - Proposed new or modified requirement text
   - Suggested model modifications (CSPM changes)
   - Priority classification (Must-fix/Should-fix/Nice-to-have)
   - Estimated effort and risk assessment
   - Advantages of the proposed fix

Target Audience:
- Requirements engineers and system architects
- Project managers and decision-makers
- Domain experts who may not know CSP
- Quality assurance and safety teams

Guidelines:
- Write clearly for both technical and non-technical stakeholders
- Provide concrete examples and traces
- Link findings to historical defects for context
- Offer actionable, prioritized recommendations
- Assess implementation effort and risk
- Explain technical concepts in accessible language
- Focus on preventing recurrence of historical failures

Output Format:
```

```
**SEVERITY:** [High/Medium/Low]

**SUMMARY:**
[One-sentence description of the gap and its implications]

**FAILED PROPERTY:**
- Description: [Natural language explanation]
- Formalization: [CSPM code]
- Criticality: [Why this property matters for system correctness]

**HISTORICAL CONTEXT:**
- Source Defect: [Defect ID, project, and date]
- Impact: [What happened, costs, schedule impact, safety implications]
- Historical Resolution: [How it was fixed previously]
- Relevance: [Why this is relevant to the current system]

**COUNTEREXAMPLE:**
- Trace: [FDR4 counterexample trace]
- Explanation: [Plain English description of what the trace shows and
  why it violates the property]

**ROOT CAUSE:**
- Missing Requirement: [What requirement is missing or incomplete]
- Model Gap: [How the current model allows this violation]
- Analysis: [Why this gap exists]

**RECOMMENDED ACTIONS:**
- New Requirement: [Proposed requirement text in natural language]
- Model Modification: [Proposed CSPM changes with explanation]
- Priority: [Must-fix/Should-fix/Nice-to-have with justification]
- Estimated Impact: [Effort level, risk assessment, affected
  components]
- Advantages: [Benefits of this fix, why it's better than
  alternatives]

Important:
- Be specific and actionable
- Provide sufficient context for decision-making
- Balance technical accuracy with accessibility
- Prioritize based on safety, cost, and historical impact
```

User Prompt

```
Generate a comprehensive gap report for the following failed
verification property.

<failed_property>
Description: {property_description}
Formalization: {property_formalization}
</failed_property>
```

```
<historical_defect>
{defect_report}
</historical_defect>

<counterexample>
{fdr4_counterexample}
</counterexample>

<current_model_excerpt>
{relevant_model_excerpt}
</current_model_excerpt>

<current_requirements_excerpt>
{relevant_requirements}
</current_requirements_excerpt>

<cspm_documentation>
{cspm_documentation}
</cspm_documentation>

Your task:
Generate a detailed gap report that helps stakeholders understand:
1. What the gap is and why it matters
2. Historical context and lessons from past defects
3. How the gap manifests in the current system (counterexample)
4. Root cause in requirements and model
5. Actionable recommendations with priority and effort assessment

IMPORTANT:
- Write for both technical and non-technical audiences
- Provide concrete examples and clear explanations
- Link findings to historical defect context
- Offer specific, actionable recommendations
- Assess priority based on safety, cost, and historical impact

Output your gap report in the following format:

**SEVERITY:** [High/Medium/Low]

**SUMMARY:**
[One-sentence description of the gap]

**FAILED PROPERTY:**
- Description: [What was being verified in natural language]
- Formalization: [CSPM code]
- Criticality: [Why this property is important]

**HISTORICAL CONTEXT:**
- Source Defect: [Defect ID, project, date]
- Impact: [What happened, costs, schedule delays, safety issues]
```

```
- Historical Resolution: [How it was fixed before]
- Relevance: [Why this matters for the current system]

**COUNTEREXAMPLE:**
- Trace: [FDR4 counterexample trace]
- Explanation: [Plain English explanation of what the trace shows and
  why it violates the property]

**ROOT CAUSE:**
- Missing Requirement: [What requirement is missing or incomplete]
- Model Gap: [How the model allows this violation]
- Analysis: [Why this gap exists in the current specification]

**RECOMMENDED ACTIONS:**
- New Requirement: [Proposed requirement text]
- Model Modification: [Proposed CSPM changes]
- Priority: [Must-fix/Should-fix/Nice-to-have with justification]
- Estimated Impact: [Effort, risk, affected components]
- Advantages: [Benefits and why this approach is recommended]

Provide actionable insights that help stakeholders make informed
decisions about addressing this gap.
```

References

1. Sarmad Bashir, Alessio Ferrari, Muhammad Abbas, Per Erik Strandberg, Zulqarnain Haider, Mehrdad Saadatmand, and Markus Bohlin. Requirements ambiguity detection and explanation with llms: An industrial study. In *International Conference on Software Maintenance and Evolution*, 2025.
2. Manal Binkhonain and Reem Alfayez. Are prompts all you need? evaluating prompt-based large language models (llm) s for software requirements classification. *Requirements Engineering*, pages 1–21, 2025.
3. Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426(37):426–431, 2005.
4. Edmund M Clarke. Model checking. In *International conference on foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.
5. Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.
6. Byron Cook, Daniel Kroening, and Natasha Sharygina. Accurate theorem proving for program verification. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 96–114. Springer, 2004.
7. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.
8. Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, pages 193–209. Springer, 2005.

9. Norbert E Fuchs and Rolf Schwitter. Attempto controlled english (ace). *arXiv preprint cmp-lg/9603003*, 1996.
10. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3—a modern refinement checker for csp. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 187–201. Springer, 2014.
11. Cordell Green. Application of theorem proving to problem solving. In *Readings in artificial intelligence*, pages 202–222. Elsevier, 1981.
12. Shabnam Hassani, Mehrdad Sabetzadeh, and Daniel Amyot. An empirical study on llm-based classification of requirements-related provisions in food-safety regulations. *Empirical Software Engineering*, 30(3):72, 2025.
13. Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
14. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.
15. Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499, 2024.
16. Hongyan Li, Weifeng Sun, Meng Yan, Ling Xu, Qiang Li, Xiaohong Zhang, and Hongyu Zhang. Retrieval-augmented fine-tuning for improving retrieve-and-edit based assertion generation. *IEEE Transactions on Software Engineering*, 2025.
17. Shiyin Lin. Llm-driven adaptive source–csink identification and false positive mitigation for static analysis. In *Proceedings of the 2025 8th International Conference on Computer Information Science and Artificial Intelligence*, pages 281–285, 2025.
18. Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. Fvel: Interactive formal verification environment with large language models via theorem proving. *Advances in Neural Information Processing Systems*, 37:54932–54946, 2024.
19. Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (ears). In *2009 17th IEEE international requirements engineering conference*, pages 317–322. IEEE, 2009.
20. Kenneth L McMillan. Symbolic model checking. In *Symbolic model checking*, pages 25–60. Springer, 1993.
21. Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.
22. Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
23. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10(2):203–232, 2003.
24. Yuchen Wang, Shangxin Guo, and Chee Wei Tan. From code generation to software testing: Ai copilot with context-based rag. *IEEE Software*, 2025.
25. Stefan Wiesner, Margherita Peruzzini, Jannicke Baalsrud Hauge, and Klaus-Dieter Thoben. Requirements engineering. In *Concurrent Engineering in the 21st Century: Foundations, Developments and Challenges*, pages 103–132. Springer, 2015.
26. Pamela Zave and Tim Nelson. Validation of formal models: A case study. In *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, pages 292–313. Springer, 2024.