It's the Requirements, STUPID!

A Methodology Paper: How AI and RAG Transform Requirements from English to Silicon

Ben Cohen https://SystemVerilog.us

> *"It's the economy, stupid."*
> — James Carville, Clinton Campaign War Room, 1992

Abstract

James Carville's famous sign was not about economics. It was about clarity. Stop overcomplicating the message. Focus on what actually matters. In hardware engineering, the equivalent truth is this: **requirements are everything.** A flawed requirement is a bug that compounds with every downstream artifact — RTL, assertions, testbenches, silicon. This paper presents a methodology in which Artificial Intelligence (AI), augmented by Retrieval-Augmented Generation (RAG) with a structured ruleset, is used to iteratively refine requirements through a closed-loop pipeline:

English → SVA → Formal English → FV → Refinements.

The key thesis is that forcing requirements through multiple representational "lenses" — natural language, formal logic, waveform diagrams, architectural schematics — surfaces ambiguities that no single representation reveals alone.

1. Introduction: The Foundational Problem

Every hardware verification failure can be traced to one of two root causes: a bug in the implementation, or a bug in the requirement. The first kind is well-studied. The second kind is dangerous precisely because it is invisible — the implementation faithfully does what was specified, and what was specified was wrong.

Consider a simple requirement: *"When req asserts, the design must eventually grant access."* This sentence appears complete. It is not. It fails to specify:

- How many cycles is "eventually"? (temporal ambiguity — RAG Rule RQM0 Category 2)
- Is `req` level or edge-sensitive? (activity ambiguity — RQM0 Category 1)
- Does `gnt` need to hold, or is a pulse sufficient? (duration ambiguity — RQM0 Category 3)
- What happens if `req` deasserts before `gnt`? (corner case — missing obligation)

These are not hypothetical gaps. They are the exact failure modes catalogued in the RQM0 Unified Quality Gate, which consolidates seven ambiguity categories into a single pre-generation validation framework. The insight this paper advances is that the process of resolving these ambiguities — through AI-guided translation and back-translation — is itself a requirements engineering methodology.

2. The Political Analogy: Why Clarity Wins

Bill Clinton's 1992 presidential campaign succeeded, in part, because it refused to dilute its message. Carville's mantra — "It's the economy, stupid" — was not a policy position. It was a discipline of focus. Voters were not confused about what the campaign stood for.

Requirements suffer from the opposite problem. They accumulate language. They inherit hedges ("should," "may," "typically"), implicit assumptions, and undocumented conventions. By the time a requirement reaches a verification engineer, it has passed through product managers, architects, and documentation writers — each adding their own interpretation. What arrives is not a requirement. It is a sediment of partially-resolved intentions.

The parallel is exact. In politics, the cure is to return to the core issue. In engineering, the cure is to force the requirement through a representation that admits no ambiguity: formal logic. The moment you attempt to encode "eventually grants access" in SystemVerilog, the questions above are no longer optional. The language demands answers.

3. Requirement Evolution: The Four-Stage Lifecycle

Requirements do not emerge fully formed. They evolve through four stages, each introducing the possibility of refinement — and regression.

Stage 1: Natural Language (English)

The initial specification lives in English. It is written for human readers: product managers, architects, customers. English is expressive and ambiguous in equal measure. The word "after" can mean "one cycle later," "within N cycles," or "at any future point." None of these are equivalent in silicon.

What AI RAF-SVA offers at this stage: RQM1 classification and summarization — parsing the requirement into distinct behavioral clauses, generating a bulleted list of what is being verified, and identifying which temporal phrases require disambiguation before code generation can proceed.

Stage 2: Signal Tabularization and INTENTION Assignment

Before any formal encoding, every signal must be classified in a structured 6-column table:

PORTPARAM	DIRECTION	TYPE	NAME	INTENTION	ACTIVITY
PORT	input	logic	req	assume	Rising edge trigger
PORT	output	logic	gnt	obligation	Level — DUT must assert
PORT	input	logic [7:0]	addr	assume	Stable during valid
LOCAL	—	logic	v_req	obligation	Captured at rose(req)

This table enforces a discipline that English conceals: **INTENTION always overrides port direction.** An input port may be an obligation (the DUT must monitor it and respond). An output port may be assumed (the testbench controls it in a particular checker configuration). Mixing assume and obligation signals within a single implication is not a style violation — it is a logical error that generates properties incapable of failing correctly.

Stage 3: SVA Encoding — The First Formal Lens

SystemVerilog Assertions provide the first representation with no tolerance for ambiguity. The translation from English to SVA forces resolution of every gap identified in Stage 1. Consider the following transformation:

English: "If req rises, gnt must assert within 5 cycles and hold until req falls."

After RAG-SVA guided decomposition (RQM0 Category 5 — multiple implications):

```
// Property 1: Liveness — gnt responds to req
// CLASS: LIVENESS-B | AXIS2: STRONG | AXIS3: BOUNDED
property ap_req_to_gnt;
  @(posedge clk) disable iff (!rst_n)
  $rose(req) |-> ##[1:5] gnt;
endproperty
```

```
ap_req_to_gnt: assert property (ap_req_to_gnt)
  else $error("%0t FAIL: gnt did not assert within 1-5 cycles after req", $realtime);

 // REQ-002: Safety — gnt holds while req is high
property ap_gnt_holds_while_req;
(gnt && req) |-> ##1 gnt;
endproperty
ap_gnt_holds_while_req: assert property (
  @(posedge clk) disable iff (!rst_n) ap_gnt_holds_while_req)
  else $error("%0t FAIL: gnt dropped while req was still high", $realtime);

// Assumption: req is a clean input
// ASSUMPTION: req does not toggle spuriously
// TYPE: Input Constraint
// JUSTIFICATION: Protocol requires stable req during arbitration
// RISK LEVEL: Low
property mp_req_stable;
  @(posedge clk) disable iff (!rst_n)
  $rose(req) |-> req [*1:20];
endproperty
mp_req_stable: assume property (mp_req_stable);

// Mandatory coverage — UR-5
cp_req_raised:     cover property (@(posedge clk) $rose(req));
cp_req_to_gnt_full: cover property (@(posedge clk) $rose(req) ##[1:5] gnt);
```

The SVA encoding has already done more than translate — it has **forced three decisions** that the English did not make:

1. The cycle bound is now **exactly 1–5**, not "within 5" (off-by-one resolved per RQM0 Category 3)
2. The property has been **decomposed** into two atomic assertions (RQM0 Category 5, LNG28 enforcement)
3. The `req` behavior has been **captured as an assumption with mandatory coverage** (UR-5)

Stage 4: Back-Translation — The Second Formal Lens

The most powerful step in this methodology is one that is rarely performed: **translate the SVA back into formal English.** The back-translated requirement is not the same document that was written at Stage 1. It is more precise, because it was shaped by an encoding that admits no ambiguity.

Back-translated from SVA above:

> **REQ-001 [LIVENESS]:** Upon the rising edge of `req`, the output signal `gnt` shall assert (transition to logic HIGH) within a minimum of one (1) and a maximum of five (5) clock cycles. Failure to assert within this window constitutes a protocol violation.
>
> **REQ-002 [SAFETY]:** Once `gnt` is asserted while `req` is simultaneously HIGH, `gnt` shall remain asserted on every subsequent clock cycle for which `req` remains HIGH. A premature deassertion of `gnt` while `req` is HIGH constitutes a hold violation.
>
> **ENV-001 [ASSUME]:** The input signal `req`, once asserted by the environment, shall remain asserted for a minimum of one (1) clock cycle and a maximum of twenty (20) clock cycles. The environment guarantees this protocol constraint.

This back-translated document is now suitable for review by product management, compliance teams, and customers — but it was *derived* from the formal verification artifact, not the other way around. The flow has run in both directions, and each direction has caught something the other missed.

4. The RAG Enhancement: Why Rules Matter

Retrieval-Augmented Generation solves a fundamental problem with general-purpose AI applied to formal verification: **general-purpose AI does not know your rules.** It knows SVA syntax from training data. It does not know that `until` is banned in your flow, that `##[0:$]` causes state explosion in your formal tool, or that your naming convention requires `ap_` for assertions and `mp_` for assumptions.

The RAG-SVA ruleset (80 rules across seven categories: RQM, STG, LNG, DOC, FV, SIM, ITG) functions as a long-term memory of every hard-won convention your verification team has accumulated. When the AI translates English to SVA, it retrieves:

- **RQM0** — to classify ambiguities before generating any code
- **STG1** — to enforce atomic decomposition (one implication per property)
- **LNG35** — to enforce goto operator syntax (`sig->N intersect ##[min:max]`)
- **FV8/FV9** — to generate fairness and livelock assumptions for liveness properties
- **UR-1 through UR-5** — to prevent unreachable assumptions that vacuously satisfy formal tools

Without RAG, the AI applies general knowledge, which is occasionally correct and frequently compliant with some other organization's conventions. With RAG, the AI applies *your* rules, consistently, every time.

The GIGO Principle in Formal Verification

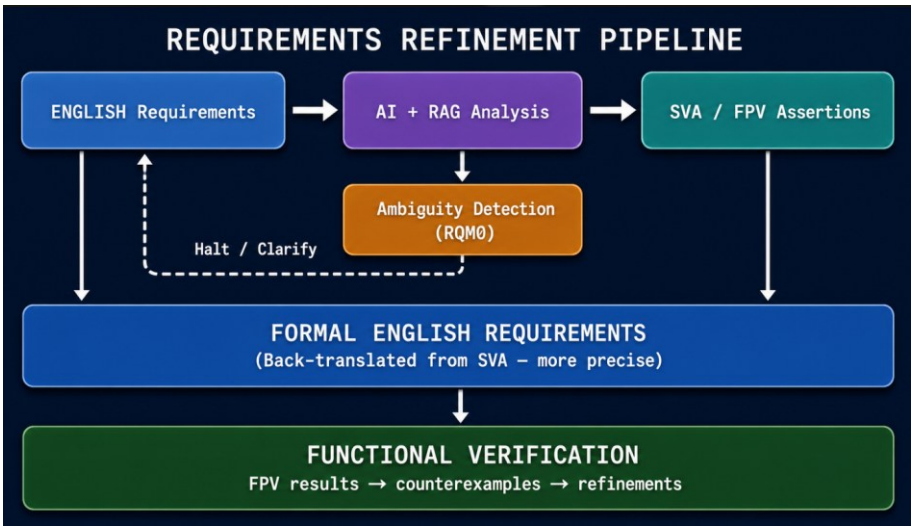
The RAG_RULES documentation states this directly:

> *"When using AI for technical tasks like generating RTL, testbenches, and assertions, the Garbage In, Garbage Out (GIGO) principle directly applies. The quality of the AI-generated output is entirely dependent on the quality of the input."*

Requirements are the input. The entire downstream chain — SVA, testbench, coverage model, formal proof — is only as good as what was fed in at the beginning. The RAG framework is the discipline that prevents garbage from entering the pipeline undetected.

5. The Closed-Loop Refinement Pipeline

The methodology proposed in this paper is not a linear flow. It is a closed loop with four reinforcing feedback **`paths:`**



Each traversal of this loop tightens the requirements:

- **English → SVA:** Forces resolution of temporal, signal, and logical ambiguities
- **SVA → FV:** Reveals vacuous properties (coverage gaps), unreachable assumptions, and liveness issues
- **FV → Formal English:** Captures the refined, verified specification as a traceable document
- **Formal English → Review:** Enables human experts to validate the semantics before committing to silicon

6. Multiple Points of View: Why Angles Matter

A requirement written in one language and reviewed in that same language is reviewed by one lens. The same limitation applies to an author reviewing their own writing — they see what they intended to write, not what they wrote.

Consider the metaphor offered in the original concept: *"It's a beautiful day. It's not hot. It's not cold."* These three statements describe the same observation from different angles. No single statement is complete. Together, they triangulate to a precise description that none individually achieves.

The same triangulation applies to requirements:

Representation	What It Reveals	What It Misses
English requirement	Intent, stakeholder context	Temporal precision, corner cases
SVA assertion	Formal logic, cycle-exact timing	Human readability, design rationale
Timing diagram	Visual signal relationships	Multi-cycle edge cases
Architecture diagram	Block-level data flow	Temporal ordering
Back-translated English	AI-verified formal spec	Original design intent
FV counterexample	Actual bug or gap in spec	Whether the spec was correct

No single representation is sufficient. The methodology derives its power from requiring **all of them**, and from using the AI + RAG framework to translate between them automatically.