It’s the Source, Stupid!

Four Entry Points, One Gated Engine

AI-Assisted Verification Notes — RAG-SVA Methodology Series

Ben Cohen — Cohen Publishing (systemverilog.us) · Dr. Kais Chibani

LICENSE: Personal/academic use only. Commercial use requires a Commercial License.

Figure 1 — The four entry points (clarify, generate, upgrade, reconstruct) converge on the same gated engine before any property is trusted; the gates check the property against the rules, not against which path produced it.

An AI can generate a SystemVerilog Assertion in seconds. Whether that assertion means anything depends entirely on what it was generated from — and in practice that source is rarely a single, clean, natural-language (NL) specification. It is just as often an ambiguous paragraph that needs clarifying before it can be trusted, an existing SVA or PSL suite that needs upgrading against a new requirement, or nothing more than legacy RTL and old simulation data standing in for a specification that was never written down properly. The RAG-SVA methodology — the rule corpus behind the Cohen & Chibani book *Knowledge-Driven AI for SystemVerilog Assertions* [14] — is built to help with all four of these, not only the from-scratch case. This paper describes what makes the source material hard in each case, where conversion errors originate, and the gated methodology that has to sit between any of these four inputs and a property an engineer can trust.

1. Why ASIC Specifications Are Hard

A specification's difficulty is not incidental — it is structural, and it comes from two directions at once. The first is scale and dynamics: the sheer size and complexity of the problem the design solves grows the specification faster than any single author can hold precisely in mind. Adding to that complexity are changes in the requirements as the design evolves, and as details about the requirements, interfaces, and IPs get clarified or modified along the way. The second is reuse: a new design is rarely built from a blank page. It must absorb existing, already-verified blocks and merge them with new requirements and new technology — and a technology change is rarely cosmetic. A higher target clock rate, for instance, can force a re-evaluation of existing IP, a redesign for timing closure, or additional pipeline stages, and each of those ripples back into the specification itself, whether or not the specification's author intended it to.

The consequence is a quiet contradiction every verification engineer eventually notices: a specification is supposed to describe behavior independent of implementation, and yet, in practice, it is frequently tied to the technology that will implement it. Two layers are usually present in the same document, and they need different treatment:

Technology-independent layer

- Performance, actions, and goals — what the design must do, expressed through its inputs and outputs, without reference to how it is built.

Technology-tied layer (implementation)

- Partitioning and IP reuse — the primary structural elements of the design. Because complexity makes reuse a necessity rather than a preference, this layer typically includes three distinct kinds of source material:
- Verified IP with configuration — e.g., a PCI bus, a CPU core, a memory controller — proven blocks whose behavior is already established and whose remaining risk is configuration, not correctness.
- A prior design requiring updates — legacy English specification, existing RTL, existing simulation data, a new set of requirements, and, if available, legacy PSL or SVA code to build from.
- New English requirements for a new partition — including the definition of its interfaces to the surrounding environment.

The OVL arbiter and **`grnt→req`** comparisons [1, 3] make this concrete: the same RTL, verified with and without a structured rule corpus behind the AI, produces measurably different assertion quality — the difference is not the AI's raw capability but whether it was told, in a form it could act on, which layer of the specification it was reading, and from which of the source types below.

2. Four Entry Points — RAG-SVA Is Not Only From-Scratch Generation

"Spec-to-property" suggests a single pipeline: English in, SVA out. In practice, RAG-SVA is asked to help at four distinct entry points, each with a different source material and a different risk profile. The gated methodology in Section 4 is the same across all four — what changes is what the gates are checking.

A. Clarifying ambiguous English

Often the very first service the framework provides is not generation at all: it is turning a vague requirement into a precise one. A halt-don't-guess pass over the English — flagging vague timing words, undefined signals, unstated cycle boundaries — produces a rewritten requirement that a human can review before any property exists. This is frequently the highest-value step, because every downstream error traced in Section 3 originates here.

B. Creating SVA from a clear natural-language spec (the best case)

When the English is already precise — or has been made precise by step A — generation from natural language is the cleanest and most reliable of the four paths. The specification states intent directly; the methodology's job is to classify signals, choose the right temporal operator, and refuse to fill any gap the English didn't actually close.

C. Rewriting or upgrading existing SVA/PSL against new requirements

A large share of real verification work is not new code but maintenance: an existing PSL or SVA suite must absorb a new requirement, a widened bus, an added mode. Here the source material is code, not prose, and the task is a merge — preserving what the old assertions got right, identifying what the new requirement changes, and re-running the full gate sequence on the result rather than assuming the untouched portions are still correct by inheritance.

D. Reconstructing SVA from old information — spec updates, legacy RTL, code, simulation data

The hardest and least discussed case: no usable specification exists at all — only a legacy design (RTL, testbench code, and historical simulation results), or an existing spec carrying multiple rounds of updates that may have injected contradictory requirements. Here the AI is not translating a specification; it is helping reconstruct one, inferring likely intent from implementation and observed behavior and presenting that inference back to the engineer as a candidate specification to confirm or correct, before any property is trusted. This path carries the highest risk of the four, precisely because there is no independent source to check the AI's inference against — which is exactly why the gates in Section 4 cannot be skipped or abbreviated here.

Each entry point needs a different generation posture — clarify, generate, merge, or reconstruct — and the Gen-SVA framework [10] treats this explicitly as a source-agnostic generation problem: the AI's job differs by provenance, and the methodology has to tell it which of the four it is looking at before it writes anything.

3. Sources of Error in the Source Material

Whichever of the four entry points is in play, the AI will faithfully translate whatever ambiguity or error already exists in its source into syntactically clean but incorrect assertions. The recurring patterns:

- Imprecise signal semantics — a signal's name suggests one behavior, but its actual timing, polarity, signature, or width is never pinned down.
- Configuration errors — a reused IP is assumed to be in one configuration when the integration actually uses another (bus width, endianness, mode select).
- Misunderstandings between author and reader — informal language (“eventually,” “shortly after,” “as needed”) that reads as precise to the author but is genuinely ambiguous downstream.
- Multi-clock domain interactions — behavior that depends on the relative phase or frequency of two clocks, often left implicit.
- Contradictory requirements — successive spec updates that each made local sense but together demand incompatible behavior; the AI will encode whichever it sees last unless the contradiction is surfaced and resolved.

For entry points C and D, two further patterns are common: stale assumptions — an old SVA suite assumes a configuration or protocol version the new requirement has quietly changed — and implementation-as-intent — legacy RTL encodes what the design happens to do, which is not always what it was supposed to do, and an AI reconstructing a spec from that RTL can mistake a bug for a requirement. The parking-lot controller study [2] shows the risk at the system level, where an incomplete environment description produced formally valid but behaviorally wrong checkers until the gaps were closed.

4. The Common Engine: One Gated Methodology, Four Entry Points

Regardless of which of the four paths supplied the source material, the same sequence of gates — developed at length in the companion paper "*It's the Methodology, Stupid!*" [14] — has to run before a property is trusted. It is the reason RAG-SVA generalizes across clarification, generation, upgrade, and reconstruction: the gates check the property against the rules, not against how the property came to exist.

Gate 1 — Sanity & Consistency

Halt-don't-guess on ambiguity (RQM0); keyword-driven hold-vs-window checks; banned operators (no `|=>`, no unbounded `##[0:$]`, no `until/within/first_match`); disciplined naming (`p_/ap_/mp_/cp_`). Applied identically whether the ambiguity came from fresh English (path B), a stale assumption in old code (path C), or an inferred guess during reconstruction (path D).

Gate 2 — Intention Boundary

Every signal is classified assume (environment) or obligation (DUT) before any implication is written. For path C, this gate also re-classifies every signal in the legacy suite — a signal correctly treated as an assumption under the old requirement may need re-classifying if the new requirement changes what the environment is responsible for.

Gate 3 — Anchored Obligations

Self-consistency checks on outputs are anchored on the output's own edge, not smuggled into an environment assumption. This matters most in path D, where an inferred obligation is easy to mis-anchor without an explicit spec to check it against.

Gate 4 — Can It Fail?

Reachability and vacuity covers, fairness assumptions, and livelock anchors accompany every property. For path D in particular, this gate is the primary defense against a reconstructed property that merely restates what the RTL already does — a property that can never fail is worthless regardless of which of the four paths produced it.

5. Preparing Source Material for Each Entry Point

Four requirements recur, adapted per entry point:

- Requirements must be unambiguous before generation begins — a gate, not a best-effort pass, whether the English is new (B) or the product of a clarification pass (A).
- If the block is a proven, reused IP, its configuration must be explicitly specified — not inferred — before any property is generated, and re-confirmed whenever path C touches that block.
- Signal signatures from a proven IP become assumptions, not obligations — accepted as given, freeing the assertion budget for the genuinely new logic around the IP.
- Style conventions — one-hot encoding on state or select signals, naming discipline, decomposition strategy — must be declared once and applied identically whether the AI is generating new properties (B), merging into old ones (C), or inferring them from RTL (D).

The two legacy-conversion papers [6, 7] and the SV/SVA-to-VHDL/PSL paper [8] are, in this framing, worked examples of path C and D together: converting an existing, working artifact into FV-ready SVA means the

"specification" is the legacy code and its accompanying English, and the same ambiguity screens from Section 3 apply before the AI is trusted to carry semantics across the translation. The FV abstraction techniques paper [12] extends this into cases where the full design is too large to verify directly and the specification itself must be deliberately abstracted — a decision that, done carelessly, reintroduces exactly the ambiguity this paper argues against.

6. Bidirectional Translation: Requirements ⇄ AI with RAG ⇄ Verification

Every path in Section 2 has been described as a forward translation: some source material goes in, and SVA properties, formal English, diagrams, or C++ come out. That is only half of what an AI with the RAG-SVA rule corpus behind it can do. The same rules that govern the forward direction — signal classification, operator discipline, the four gates in Section 4 — apply symmetrically in reverse: from an updated verification artifact back to an updated specification or document.

Figure 2 — AI with RAG sits between requirements and verification and translates in both directions: English, diagrams, PSL/SVA, and C++ on one side; English specs, SVA, diagrams, and C++ on the other.

Forward: requirements in any format → verification artifacts

The rule corpus does not require a single canonical input format. English prose, a diagram, an existing PSL/SVA suite, or a C++ reference model can each serve as the source material described in Section 2's four entry points, and the AI, guided by the same gates, produces whichever verification artifact is needed: SVA properties, a cleaned-up formal English specification, a diagram of the property's timing, or a C++ checker. Which format comes in and which comes out are independent choices — the methodology sits between them, not tied to either.

Reverse: updated verification artifacts → updated specs and documents

The same translation runs backward. When a design changes fast — an SVA suite is updated first, a C++ model is patched ahead of the English, or a diagram is redrawn to reflect a new timing bound — the AI can read that updated artifact and regenerate the corresponding English specification and documentation, rather than leaving the prose spec to drift silently behind the code it was supposed to describe. This is the direct countermeasure to the spec provenance and staleness risk raised later in this paper: the moment the

verification side changes, the specification side can be regenerated to match, instead of waiting for the drift to be discovered the hard way, during the next integration.

Read this way, entry points C and D from Section 2 are not really separate operations from the forward path — they are the reverse direction of the same bridge. Path C's "merge" and path D's "reconstruct" both start from an artifact that changed or existed before the specification did, and both ask the AI to bring the specification back into alignment with it. The four gates in Section 4 apply exactly as before: whichever direction the translation runs, the gates check the result against the rules, not against which direction it came from.

Additional Concepts Worth Folding Into the Methodology

- Spec provenance and staleness — when reusing an "old specification," how do you know it still matches the RTL it is paired with? Path C and D both inherit whatever drift has accumulated.
- Confidence and traceability tagging — each generated property could carry a tag back to the exact source sentence, code line, or IP datasheet section it came from, mapping onto the DOC9 [15] justification blocks used elsewhere in the corpus — especially valuable for path D, where the "source" is inferred rather than stated.
- The silently-fill-the-gap risk — the single largest AI-specific risk across all four paths: an AI asked to generate a property from an incomplete source may produce something plausible-looking rather than halting, unless the methodology explicitly forbids it.
- Cross-IP assumption conflicts — when two proven IPs are integrated, their individual "signals become assumptions" treatment can conflict at the interface, where IP A assumes what IP B is supposed to obligate.
- Versioned configuration — IP configuration is not static across a project's lifetime; a configuration change mid-project should trigger re-validation of every assumption derived from the old configuration — a direct concern for path C.
- Human-in-the-loop calibration — the methodology should state up front what class of decision the AI may make autonomously versus what must always halt to a human, with path D requiring the tightest calibration of the four. The AI-scribe comparison [13] is a useful outside reference: medicine has already worked through what a scribe may infer versus what it must defer to the clinician on.
- Frame drift across long interactions — in a long multi-turn session an AI can quietly lose track of earlier constraints; a rule corpus that is reloaded and re-checked each pass defends against this, as developed in the frame-drift paper [16].
- A path for the engineer, not just the AI — how a verification engineer's own workflow changes once source screening and AI-assisted generation are both in place, across all four entry points [5].

Closing

Spec-to-property generation through AI is genuinely useful — and its usefulness is broader than a single from-scratch pipeline. RAG-SVA earns its keep across all four entry points: clarifying English before it becomes a bad property, generating cleanly from a good spec, upgrading existing SVA/PSL against new requirements without silently trusting the untouched parts, and reconstructing a workable specification from legacy RTL and

simulation data when nothing better exists. What holds all four together is the same gated methodology [14] — sanity, intention boundary, anchored obligations, can-it-fail — applied without exception regardless of where the source material came from. Where RAG-SVA sits against the wider field of LLM-based assertion generators is taken up separately in the landscape comparison [17]. The risk is not that AI generates bad SVA. The risk is that AI generates plausible SVA from a bad or incomplete source — and plausible is far harder to catch in review than obviously broken. Fix the source first, whichever of the four it is.

References

[1] *OVL Arbiter, Claude with vs. without SVA-RAG for Formal Verification* — https://shorturl.at/MpA6a

[2] *AI-Assisted FV of a SV Parking Lot Controller* — https://shorturl.at/KcFgG

[3] *grnt -> req, Claude LLM vs Claude LLM + RAG* — https://shorturl.at/uLzV0

[4] *It's the Requirements, STUPID!* — https://shorturl.at/ZJhiY

[5] *Post: Path for Engineers for Effective Solutions* — https://shorturl.at/vqjnO

[6] *Converting Legacy PSL-SIM to SVA-FV-Ready Code* — https://shorturl.at/JK03x

[7] *Converting Legacy VHDL to SVA-FV-Ready Code + Specs* — https://shorturl.at/ftICz

[8] *Converting SV/SVA to VHDL/PSL* — https://shorturl.at/cw57E

[9] *RAG-SVA: A Paradigm Shift in Formal Verification* — https://systemverilog.us/Paper_June2426.pdf

[10] *Gen-SVA: Source-Agnostic Generation of Formal-Ready SVA* — https://systemverilog.us/Gen-SVA_Paper_Cohen_KAIS.pdf

[11] *Degeneracy, Environment vs. Obligation, and Design Intent: A req/ack Case Study* — http://systemverilog.us/req_ack_degeneracy_paper.pdf

[12] *FV Abstraction Techniques in RAG-SVA* — http://systemverilog.us/fv_abstraction_paper.pdf

[13] *What if AI Scribe for Doctors Were Used for AI Scribe for Chip Design and Verification?* — http://systemverilog.us/AI_scribe_medical_vs_chip_comparison.pdf

[14] *Cohen & Chibani, Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules (2026); companion paper "It's the Methodology, Stupid! — Meaningful SVA: Write for Formal, Run Anywhere" develops the four-gate engine referenced in Section 4*

[15] *DOC9: Assumption Documentation Rule — governs the required documentation, classification, and metadata tracking for formal assumptions (assume property). Every environmental constraint must include a structured header stating its human-readable intent, formal justification, and risk level; high-risk assumptions require an explicit HUMAN REVIEW REQUIRED tag.*

[16] *Frame Drift in Multi-Turn LLM Interaction, and Why RAG-SVA Rule Corpora Already Defend Against It* — https://systemverilog.us/

[17] *RAG-SVA in the Landscape of LLM-Based Assertion Generation* — https://systemverilog.us/

[18] *How to Use the "not" Operator for Formal Verification* — https://systemverilog.us/

[19] *Deriving a Formal req/ack Assertion Under the RAG-SVA Methodology* — https://systemverilog.us/

[20] *And Now Is the Time: How AI with RAG-SVA Transforms Requirements, Assertions, Formal Verification, and Testbench Generation* — https://systemverilog.us/

Full paper series and links available at systemverilog.us (under Papers).

LICENSE: Personal/academic use only. Commercial use of the RAG-SVA framework and rule corpus requires a Commercial License — contact Cohen Publishing (Ben_cohen@systemverilog.us).