And Now Is the Time

How AI with RAG-SVA Transforms Requirements, Assertions, Formal Verification, and Testbench Generation

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

Every engineer who has been in digital design long enough carries a personal list of tools they once could not work without — and later set aside when something better arrived. This paper is organized around that list, because it is the honest way to describe where AI-assisted verification now sits: not as a novelty, but as the next entry in a long line of tools that each, in their moment, became indispensable. It draws on a talk given at the meetup on Open-Source EDA Driving AI Adoption in Chip Design, organized by Srinivasan Venkataramanan, and on direct experience building and using the RAG-SVA methodology — the rule corpus behind Cohen & Chibani's *Knowledge-Driven AI for SystemVerilog Assertions* [1]. The argument is simple: AI, properly directed by a rule corpus, delivers benefits across the entire verification flow — clarifying requirements, generating formal-ready SVA, producing input for formal verification tools, and generating testbench environments — not merely at the single step of writing an assertion.

1. There Was a Time

The case for a new tool is easiest to see against the history of the old ones. Each item below was, at some point, something a working designer (*including me*) would not start a design without:

- There was a time I would not do a design without a Karnaugh map.
- There was a time I would not do a design without a drawing template, template decals, or a graphics station.
- There was a time I would not do a design without Verilog or VHDL plus a simulator.
- There was a time I would not do a design without a synthesis tool.
- There was a time I would not do a design without PSL plus a simulator.
- There was a time I would not do a design without SVA plus a simulator.
- There was a time I would not do a design without UVM.

And now is the time I would not do a design without AI plus RAG-SVA — and would not do a design without formal verification. Each tool on that list did not replace the engineer; it removed a category of manual, error-prone labor and let the engineer spend attention where judgment actually mattered. AI with a rule corpus is the newest entry, and it earns its place the same way the others did.

2. What AI Is — and What It Needs

It helps to be precise about what AI is, because the precision explains both its power and its limits. In this author's view, an AI language model is essentially a vast collection of linked volumes, data, and code. It is very good at assembling, composing, summarizing, eliminating noise, and researching information. From that foundation it can compose text and create code — SVA or otherwise. But it cannot do any of this well without direction.

Direction means: what do you want, what are your goals, and which rules do you want applied? The analogy to English composition is exact. Ask an AI to write a paragraph and it must know whether you want formal business prose or a friendly personal tone — and if you want your own style, it must first learn your style and your rules of writing. Translating English into SVA is no different. An AI equipped with a capable language model, a working knowledge of the IEEE 1800 standard, and exposure to a large body of SVA examples can indeed translate English into SVA. But that translation will not conform to your style and guidelines unless those guidelines are supplied.

This is the crux for formal verification specifically. Writing SVA for FV requires specific rules so that the code produced is FV-ready for use by formal tools. Your rules may also demand well-documented code, and code that is more complete and better-defined than the loose requirement — pinning down the signature of signals, and refusing to emit an assertion that contradicts itself and is therefore vacuously, always true (the classic totality trap, e.g., `sig || !sig`). Direction is not a nicety here; it is the difference between a plausible-looking assertion and one a formal tool can actually use.

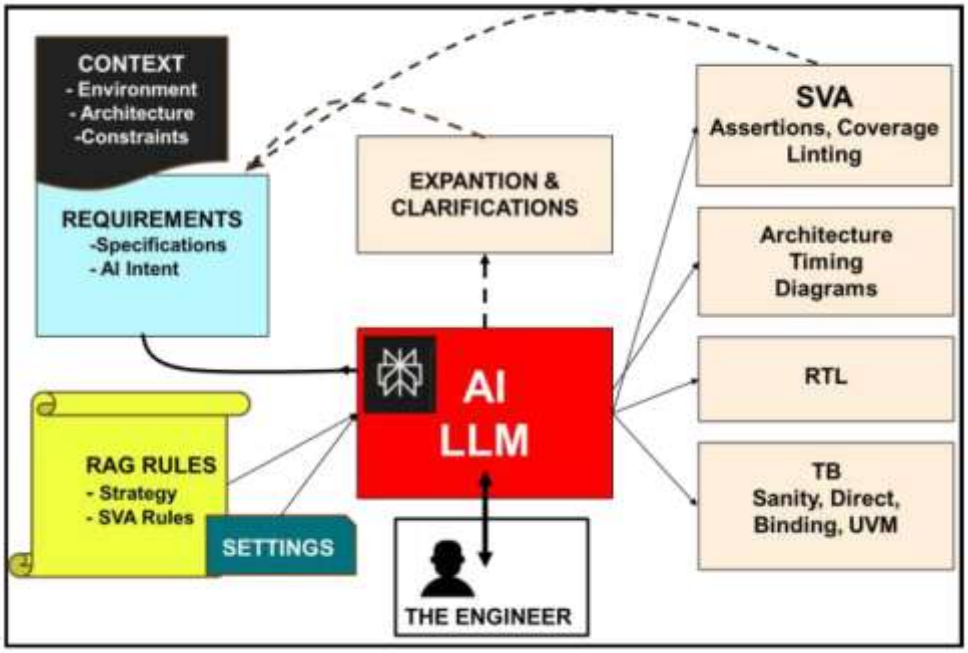

Figure 1 — What AI creation of SVA and a verification environment needs: LLMs for SVA, decomposition and analysis, the user's goals (FV / SIM / settings), and the RAG-SVA rule layer (assume/obligation boundaries, coverage, unreachability).

3. The Back-and-Forth: Requirements ⇄ AI ⇄ Verification

The single most valuable pattern that emerges from working this way is not one-directional translation but navigation — moving back and forth between the requirements and the verification artifacts, with the AI as the bridge. This back-and-forth provides several distinct advantages:

- Better understanding of the requirements — the act of forcing a vague requirement through translation exposes exactly the ambiguities that would otherwise survive into the design.
- Better generation of SVA code — a requirement that has been clarified generates cleaner, more formal-ready assertions.

- Better redefinition of the requirements — once a requirement has been translated into SVA, diagrams, or C++, the translation itself reveals how the requirement should be restated.

A concrete and reusable technique follows from this. You can instruct the AI: *"write requirements based on the original requirements and the generated SVA and diagrams, and write them in a way that you, the AI, can understand, so that you could create good SVA per my rules."* The result is a reusable specification document — one written to be re-consumed by the AI on the next iteration, and therefore far easier to modify when the design changes. The requirement stops being a static, drifting artifact and becomes a living one that stays aligned with the verification code it produced.

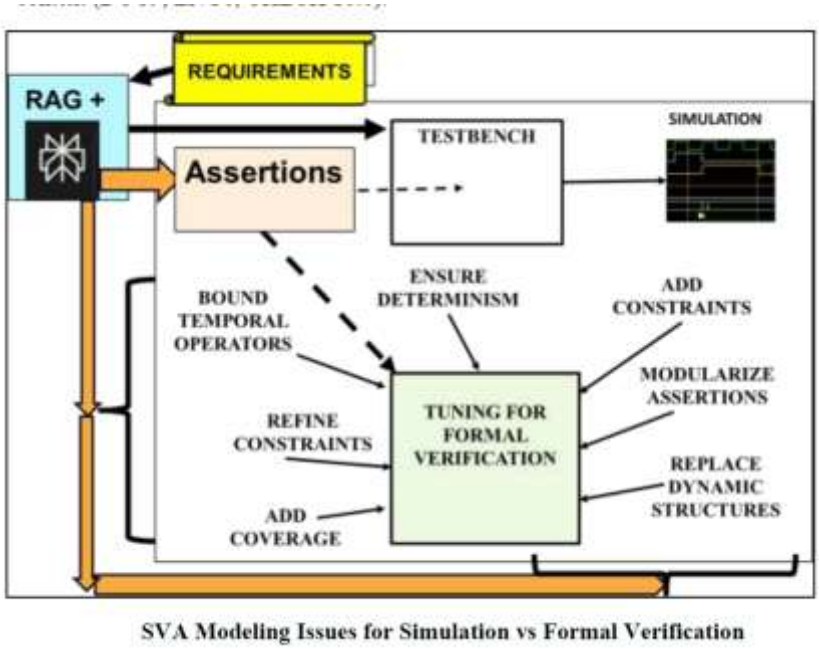

Figure 2 — The back-and-forth navigation between requirements and verification. AI sits in the middle and translates in both directions, so each pass improves the requirement, the SVA, and the shared understanding.

4. What AI Creation of SVA and a Verification Environment Needs

Turning the general idea of "direction" into something operational means enumerating what the AI actually consumes to produce good, FV-ready output. In the RAG-SVA approach these inputs are explicit:

LLMs for SVA, decomposition, and code generation

The underlying model supplies the raw capability — language understanding, knowledge of 1800, pattern familiarity from many examples — and the methodology supplies the decomposition strategy: how a compound requirement is broken into atomic, separately verifiable properties, analyzed, and turned into code.

The user's goals and settings

The methodology must know whether the target is formal verification or simulation, and it must honor settings — decomposition levels (*atomic vs. pipeline*), endpoint usage, documentation templates. These are not cosmetic: a property decomposed for FV convergence looks different from one written for a simulation testbench, and the same requirement legitimately produces different code depending on the declared target.

The RAG-SVA rules

This is the layer that makes output FV-ready rather than merely plausible. It carries the assume/obligation boundary discipline (*environment signals constrain, DUT signals are obligated — never confused*), mandatory coverage (*antecedent reachability and full-sequence execution*), and unreachability prevention (*UR rules that keep assumptions satisfiable rather than silently vacuous*). It is also where the guardrails against the totality trap and the banned-operator list live.

Context and requirements

Finally, the AI needs the surrounding context: the environment (*inputs, ranges, constraints such as a pipeline chosen for speed, or an architecture such as master/slave*), and the requirements themselves — what the design must do given that environment. Context and requirements are separable, and stating them separately is part of what keeps the generated assertions honest.

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.

Figure 3 — The full input stack the AI consumes to generate FV-ready SVA and a verification environment: context and environment, requirements, settings, and the RAG-SVA rule corpus, feeding decomposition, analysis, and code generation.

5. The Issues AI Must Handle — Especially for Formal

Enthusiasm for the tool should not obscure the fact that AI must handle several genuine issues, and they are sharpest in formal verification. Naming them is part of using the tool responsibly.

Self-contradiction and the always-true trap

A requirement that quietly contradicts itself produces an assertion that is vacuously, always true — and an always-true assertion checks nothing while appearing to pass. The totality pattern `sig || !sig` is the obvious case, but real ones hide inside longer expressions where a guard and a condition overlap so that the property can never actually be exercised. A rule corpus that forbids these patterns, and that halts on ambiguity rather than guessing, is what prevents a green pass from being mistaken for a real proof.

Signal signature and completeness

The loose English requirement rarely pins down the full signature of a signal — its polarity, its timing, its width, whether it is a pulse or a level. The AI, directed by rules, must either infer these from stated context or halt

and ask, rather than silently choosing a plausible default. Completeness of the requirement is not the AI's to invent; it is the AI's to surface and the engineer's to confirm.

FV-readiness versus mere legality

Code that compiles is not the same as code a formal tool can converge on. Bounded ranges, correct strong/weak classification, satisfiable assumptions, and reachable cover points are all requirements that the simulation world tolerates skipping and the formal world does not. The value of the rule corpus is precisely that it encodes this difference so the AI produces FV-ready output by construction, not by luck.

6. Beyond Assertions: Formal Tool Input and Testbench Generation

It would understate the benefit to describe AI-with-RAG as an assertion generator alone. The same directed capability extends across the verification environment. Because the methodology already classifies signals as environment assumptions or design obligations, it is positioned to generate the assumption sets and constraints a formal tool needs to converge — not just the assertions it checks. The coverage and unreachability machinery that keeps assertions honest is the same machinery a testbench needs to know what to stimulate and what to consider reachable.

In practice this means one directed, rule-governed process can contribute across the flow: clarified and reusable requirements; formal-ready SVA properties; the assume/constraint input that formal verification tools require; coverage models; and the scaffolding of a testbench environment. The unifying idea is that all of these are downstream of the same two things — a well-directed AI and a rule corpus that tells it what "good" means for this design, this target, and this engineer's standards.

7. The RAG-SVA Paper Series

The methodology described here has been applied and stress-tested across a range of AI-and-verification topics, each documented in a standalone paper. Collected below — available under **Papers** at *systemverilog.us* — they show the same rule corpus operating on different problems: controlled experiments, legacy conversion, methodology argument, and specific operator hazards.

Controlled experiments — AI with vs. without the rules

- OVL Arbiter: Claude with vs. without SVA-RAG for Formal Verification — a side-by-side showing that the same model, on the same arbiter RTL, produces measurably better FV-ready assertions when the RAG-SVA corpus is behind it than when it is not.
- grnt → req: Claude LLM vs. Claude LLM + RAG — a focused before/after on a single handshake direction, isolating the corpus's effect on assertion quality.
- AI-Assisted FV of a SystemVerilog Parking Lot Controller — a system-level study where an incomplete environment description produced formally valid but behaviorally wrong checkers until the rule-driven gaps were closed.

Legacy and cross-language conversion

- Converting Legacy PSL-SIM to SVA-FV-Ready Code — taking an existing simulation-oriented PSL suite and re-deriving it as formal-ready SVA under the corpus.

- Converting Legacy VHDL to SVA-FV-Ready Code + Specs — reconstructing both assertions and a usable specification from legacy VHDL, an instance of the reconstruct entry point.
- Converting SV/SVA to VHDL/PSL — the reverse-direction translation, demonstrating the corpus is not tied to a single source or target language.

Methodology and paradigm

- RAG-SVA: A Paradigm Shift in Formal Verification — the foundational argument for rule-governed, retrieval-augmented SVA generation.
- Gen-SVA: Source-Agnostic Generation of Formal-Ready SVA — the framing of spec-to-property generation as a source-agnostic problem, where the AI's posture differs by provenance.
- It's the Methodology, Stupid! — Meaningful SVA: Write for Formal, Run Anywhere — the four-gate engine argument: what makes an assertion trustworthy is the methodology gating it, not the operator used.
- FV Abstraction Techniques in RAG-SVA — handling designs too large to verify directly by deliberately abstracting the specification without reintroducing ambiguity.

Specific hazards and case studies

- Degeneracy, Environment vs. Obligation, and Design Intent — a req/ack Case Study — how a guard/done coincidence produces an assertion satisfied trivially at time zero, and how the intention boundary prevents it.
- How to Use the "not" Operator for Formal Verification — the placement and grammar of **`not():`** legal in a consequent, illegal as an antecedent, and the plausible-but-wrong forms in between.
- Deriving a Formal req/ack Assertion Under the RAG-SVA Methodology — a step-by-step worked derivation of a hold-until-ack plus bounded-window handshake, from classification through final code.
- It's the Requirements, STUPID! — the argument that verification quality is bounded by requirement quality, and that clarifying the requirement is the highest-value first step.
- What if AI Scribe for Doctors Were Used for AI Scribe for Chip Design and Verification? — a cross-domain comparison drawing on how medicine calibrates what an AI scribe may infer versus what it must defer to the human.

8. Closing

AI baffled me — in the best sense. It is a genuinely great tool, and, directed by a rule corpus that encodes what FV-ready and well-documented mean, it delivers across the whole verification flow rather than at a single step. That is why it now belongs on the list of tools a design should not be done without, next to the simulator, the synthesizer, and formal verification itself. **And now is the time I would not do a design without AI + RAG-SVA, and would not do a design without formal verification.**

References

[1] *Cohen & Chibani, Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules, Cohen Publishing, 2026 — available at https://systemverilog.us/*

[2] *Meetup: Open-Source EDA Driving AI Adoption in Chip Design (organized by Srinivasan Venkataramanan)*

[3] *Companion papers: "It's the Methodology, Stupid!" and "It's the Source, Stupid! — Four Entry Points, One Gated Engine"*

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