RAG-SVA in the Landscape of LLM-Based Assertion Generation

Constrain-at-Generation vs. Generate-then-Repair

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

A wave of LLM-based frameworks for generating SystemVerilog Assertions (SVAs) has appeared over the past two years, several from major EDA and semiconductor groups. They share a common architecture that differs in a fundamental way from the RAG-SVA methodology of Cohen & Chibani [1]. This note maps that landscape, names the closest peer frameworks, and states plainly where RAG-SVA is differentiated, where it is at a disadvantage, and what would strengthen its position. All external frameworks are cited with links in the references.

1. The Field, in One Sentence Each

AssertLLM [2] was among the first to process complete specification files and generate SVAs for each architectural signal. **AssertionForge [3]** (NVIDIA) constructs a Knowledge Graph (KG) from both specification and RTL, fusing them into a unified representation for multi-resolution context synthesis. **LISA [4]** breaks the natural language (NL) spec into a chain-of-thought verification plan, generates SVAs per plan item, formally verifies them, and refines in a RAG-assisted feedback loop. **"Knowledge Graphs, the Missing Link" [5]** — the newest and most advanced — builds a KG from structured intermediate representations of spec, RTL, and formal-tool feedback (syntax diagnostics, counterexamples, coverage), then runs a multi-agent workflow driving three refinement loops. **Spec2Assertion [6]** uses progressive regularization and chain-of-thought for pre-RTL generation. Surveys [7] additionally catalog ChIRAAG, LAAG-RV, AssertGen, SANGAM, DeepAssert, and VERT.

2. Sidebar: What Is a Knowledge Graph (KG)?

Several of the frameworks above rely on a Knowledge Graph, so it is worth a plain-language definition before the comparison. A knowledge graph stores information as a network of things and the relationships between them, rather than as flat text. It has three parts: nodes (the entities — a signal, a module, a requirement, a port); edges (the relationships — drives, resets, responds-to, belongs-to); and triples, the storage unit, written as (subject → relationship → object).

A sentence buried in a spec — "*the ack signal is driven by the arbiter and must respond within 3 cycles of req*" — becomes a set of structured, queryable facts:

```
(ack)   —[driven_by]→    (arbiter)
(ack)   —[responds_to]→  (req)
(ack)   —[within]→       (3 cycles)
(req)   —[is_a]→         (input_port)
```

The value is that a machine can traverse and query this structure. It can ask "*what drives ack?*" or "*which requirements mention req?*" and get exact answers by walking the edges, instead of relying on an LLM to re-read the whole spec and notice. The contrast with plain retrieval is the key point: ordinary RAG retrieves chunks of text that look relevant; a KG retrieves specific facts and their connections. That is why the KG-based frameworks argue they give better grounding — there is less chance the model invents a signal name or misreads a relationship, because the relationships are stored explicitly. In the assertion-generation setting, these frameworks build a KG that links the specification and the RTL into one connected structure, so that

when the model generates an assertion it can pull the exact signal name, direction, width, and hierarchy by querying the graph rather than guessing; the newest work also stores formal-tool feedback — syntax diagnostics, counterexamples, coverage — as queryable nodes.

3. The Common Architecture — and the Divergence

Despite their variety, the leading frameworks converge on one architecture: generate-then-refine, grounded in RTL, closed with formal-tool feedback. The pattern is: extract structure from spec and RTL, generate SVA, run the formal tool, feed failures (syntax errors, counterexamples, coverage gaps) back, and regenerate. Their own reported remaining weakness is semantic correctness for properties needing deeper temporal reasoning or global invariants [5].

RAG-SVA takes a different position on four axes. The contrast is not that one is right and the others wrong — it is that they optimize for different things.

Axis	KG / RAG academic frameworks	Cohen & Chibani **RAG-SVA**
Input modalities	Primarily specification text plus RTL; some add waveform diagrams	English, timing diagrams, RTL, PSL, SVA, and C++ — singly or fused together
Use of RTL / diagrams	RTL fused into the intent model to fill spec gaps (risk: implementation-as-intent)	RTL and diagrams read for signal facts and precise timing; the requirement stays authoritative for intent
Correctness strategy	Generate, then repair via tool-feedback loops (CEX, coverage)	Constrain at generation via a rule corpus; do not emit the error in the first place
Ambiguity handling	Resolve with more context (KG, RTL fusion)	Halt on ambiguity; require human confirmation (RQM0)
Works pre-RTL (requirements only)	Largely no — the KG-grounded, closed-loop frameworks need RTL (and often a runnable design) to build the graph and drive feedback	Yes — English or a timing diagram alone is a complete input; RTL is optional
Ingests existing PSL / SVA	Not a first-class input — prior SVA appears at most as prompt examples	Yes — a primary entry point: upgrade, merge against new requirements, or convert to FV-ready form
Rule enforcement	Implicit in prompts / schema; judged by the tool after the fact	Explicit, named, auditable rules (CHECK ambiguities, tautologies, intention boundary, banned operators, unreachability) enforced before the tool runs

4. Input Flexibility: More Than English

A point worth stating plainly, because it is easy to understate: RAG-SVA is not an English-only pipeline. It accepts, and takes advantage of, a range of input modalities — as broad as, and in the case of timing diagrams broader than, the KG-based frameworks. It can consume any of the following, singly or in combination:

- English requirements — the baseline intent statement.
- Timing diagrams and waveforms — often the most precise statement of a temporal requirement, showing exact cycle offsets, hold windows, and edge relationships that prose blurs. A waveform showing req rising and ack following two-to-three cycles later, with req held until ack, encodes the exact [2:3] bound and the hold that English would leave vague.
- Existing RTL — read for signal facts: names, widths, polarities, port directions, clock and reset structure. This is the same raw grounding benefit the KG frameworks extract from RTL.
- Legacy PSL / SVA — an existing assertion suite to upgrade, merge against a new requirement, or convert to FV-ready form.
- C++ or reference models — behavioral intent expressed as executable code.

The important part is that these combine. RAG-SVA can take old RTL plus a timing diagram plus an English requirement plus a legacy PSL/SVA suite together, and cross-use them: the RTL supplies the signal signature, the diagram supplies the exact cycle relationships, the English supplies the intent, and the legacy assertions supply what was already verified. The discipline is not about refusing inputs — it is about which input is authoritative for what. RTL and diagrams are trusted for facts and timing; the requirement remains authoritative for intent, which is precisely what avoids the implementation-as-intent hazard the KG frameworks cite as their hardest open problem.

A short example: three sources, one assertion

Suppose the following three inputs arrive together for a req/ack handshake:

```
// (1) RTL  – supplies signal FACTS (names, widths, directions, clock):
module arbiter (input logic clk, input logic req, output logic ack, ...);

// (2) Timing diagram – supplies exact TIMING:
//     req  __/"""\____        req rises at T1, held through ack
//     ack  ________/"\__       ack pulses at T3  (= req_rise + 2)
//     window: ack legal at +2 or +3 cycles after req rise

// (3) English – supplies INTENT:
//     "ack must respond 2 to 3 cycles after req rises; req holds until ack."
```

RAG-SVA fuses the three: the RTL confirms ack is a 1-bit output on this clock (fact), the diagram fixes the window at exactly [2:3] and shows the hold (timing), and the English confirms the obligation direction and the until-ack hold (intent). The result is a pair of FV-ready properties — the requirement authoritative for what must be true, the RTL and diagram authoritative for the signal signature and the exact bound:

```
// ack obligation – window fixed by the diagram, intent by the English:
ap_req_to_ack_window: assert property (@(posedge clk) disable iff (!rst_n)
  $rose(req) |-> ##[2:3] ack);

// req hold – an environment assumption (req is an input per the RTL):
mp_req_holds_until_ack: assume property (@(posedge clk) disable iff (!rst_n)
  (req && !ack) |-> ##1 req);
```

No single source produced the assertion alone. The RTL told RAG-SVA that req is an input (so its hold is an assumption, not an assertion) and that ack is the output obligation; the diagram fixed the exact [2:3] window that the English stated only loosely; the English fixed the intent and the until-ack hold. That is multi-source fusion in the service of a rule-governed result — richer input than an English-only reading, without surrendering authority over intent to the implementation.

5. Where RAG-SVA Is Genuinely Differentiated

- FV-ready by construction, not by repair. The academic frameworks lean on the formal tool as the correctness oracle inside a loop. RAG-SVA encodes FV-readiness — bounded operators, strong/weak discipline, satisfiable assumptions, unreachability defense — as generation-time rules, so the output is formal-ready before a tool ever runs.
- Halt-don't-guess on ambiguity. Every framework above cites specification ambiguity and incompleteness as the core problem, and every one tries to resolve it with more context. RAG-SVA

treats genuine ambiguity as a stop condition: name the candidate readings, get a human decision, then generate. It is a distinct — and arguably safer — answer to the same problem they all identify.

- Intention-boundary discipline. The assume-versus-obligation classification (environment constrains, design is obligated) is a first-class, enforced rule in RAG-SVA. None of the peer frameworks foreground this separation the way RAG-SVA does, yet it is exactly what prevents an assertion from silently blaming the design for the environment's behavior.
- Auditability. Because the rules are named and applied at generation time, every generated property traces to a specific rule. The closed-loop frameworks judge correctness by tool outcome, which is powerful but leaves the why of a given property implicit.

6. Where RAG-SVA Is at a Disadvantage (Stated Honestly)

- Benchmarks vs. case studies. AssertionForge reports assertions-generated / syntactically-correct / proven counts on four to five designs [3]; the KG "Missing Link" work evaluates seven open-source designs with GPT-5.2 and Cadence JasperGold [5]. RAG-SVA's published evidence is qualitative worked examples. Closing this gap means numbers on shared, open benchmarks.
- Visibility. The peer frameworks are on arXiv and in IEEE/ACM proceedings, several with NVIDIA or EU-funded backing; RAG-SVA's papers live on *systemverilog.us* and are not indexed against the generic search terms. A single indexed publication would change discoverability substantially.
- No persistent cross-run KG artifact. The newest KG frameworks build a queryable graph that persists across runs, accumulating tool feedback (counterexamples, coverage) as reusable structured data. RAG-SVA ingests and fuses its inputs per task but does not, by default, maintain that persistent graph database across sessions — a capability worth considering as an addition, not a deficiency in the generation itself.

8. The Circularity Problem: Deriving a Check From the Thing It Checks

There is a foundational objection to any approach that grounds assertion generation in RTL, and it deserves to be stated plainly because it shapes how these frameworks can and cannot be combined with RAG-SVA. An assertion's purpose is to check whether the RTL is correct. If the assertion is derived from the RTL, the check reduces to "the RTL does what the RTL does" — which is true no matter how buggy the RTL is.

The failure is concrete. Suppose the requirement is "*ack must arrive within 3 cycles*," but the RTL has a bug and actually takes five. An assertion inferred from that RTL becomes `req |-> ##[1:5] ack` — and it passes, because it encoded the bug as if it were the specification. The check certifies the defect. A golden reference cannot be derived from the artifact it is meant to judge, any more than an exam can be graded using the student's own answers as the answer key. This is the implementation-as-intent hazard, and it is most dangerous precisely for behavioral and timing requirements — the same class the KG frameworks concede is their hardest open problem.

The fact / intent distinction is the whole game

The RTL-grounded frameworks are not naive about this, and fairness requires stating their defense. It rests on separating two uses of RTL. Using RTL for facts — signal names, widths, port directions, clock and reset structure — is entirely safe: a timing bug does not corrupt the fact that ack is a one-bit output on clk. Using RTL for intent — inferring what the assertion should require from what the RTL does — is the circular part.

The better frameworks keep the specification in the loop alongside the RTL and lean on an external oracle (the spec, or a human reviewing counterexamples) to decide intent. The weakness is not that they ignore the problem; it is that they do not make the fact-versus-intent separation a first-class, enforced rule, so the circularity leaks back in exactly where it is most dangerous.

RAG-SVA makes that separation an enforced rule rather than an implicit hope. RTL and timing diagrams are authoritative for facts and for timing-as-specified; the requirement is authoritative for intent; and when they conflict, the requirement wins, by rule, with genuine ambiguity halted to a human. RAG-SVA can therefore read the same RTL without inheriting its bugs as intent, because the corpus forbids RTL from defining correctness. Stated as one line: the RTL-grounded frameworks use RTL to help decide what is correct; RAG-SVA uses RTL only to help express a correctness the requirement has already defined. The first can certify a bug; the second cannot, by construction.

9. Not Competition but Composition: A RAG-SVA-First Pipeline

The natural conclusion is not that one approach wins, but that they compose — and that the order of composition is not arbitrary. Each covers the other's structural gap. RAG-SVA is strongest at the front of the flow: turning requirements, timing diagrams, and legacy assertions into FV-ready, intent-authoritative SVA, before or without RTL. The KG and closed-loop frameworks are strongest at the back: once RTL exists, grinding properties against the design and the formal tool's feedback to surface deep temporal corner cases and coverage holes that static rules cannot see.

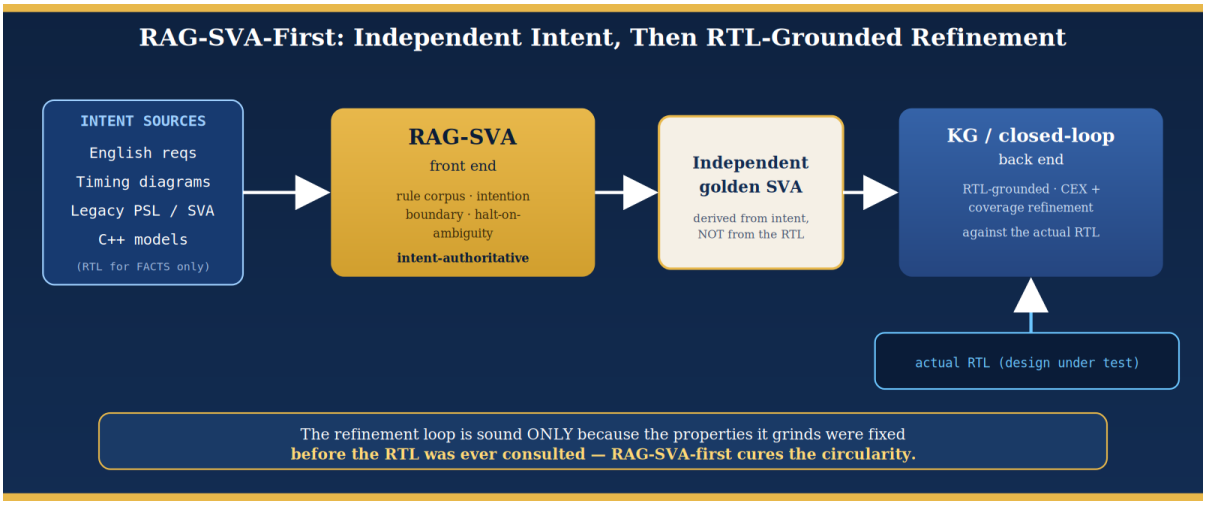

Figure 1 — RAG-SVA-first pipeline: intent-authoritative properties are fixed from requirements and diagrams (RTL used for facts only) before any RTL-grounded refinement loop runs. The loop is sound precisely because the properties it refines were derived independently of the design.

But the ordering matters for a reason deeper than convenience. A counterexample-driven refinement loop is only sound if the properties it refines were derived independently of the RTL in the first place. If the seed properties already came from the RTL, the loop is polishing a check that certifies the design against itself — the circularity of Section 8, now automated. RAG-SVA supplies exactly the independence the loop needs: properties whose intent came from the requirement, not the implementation. In that light RAG-SVA-first is not merely a tidy pipeline ordering; it is the cure for the RTL-grounding circularity. The disciplined, intent-authoritative front end is what rescues the RTL-grounded loop from certifying its own design's bugs.

The combined flow, then, is: RAG-SVA establishes intent-authoritative properties first — from requirements and diagrams, independent of RTL — producing an auditable golden reference genuinely separate from the design. The RTL-grounded loop then does what it does well: checking that independently-derived reference against the actual RTL, using counterexamples and coverage to find where the implementation violates the intent. RAG-SVA gains persistent tool-loop refinement it does not natively provide; the loop gains the independent intent and pre-RTL reach it structurally lacks. The result is stronger than either alone, and it is sound precisely because the intent was fixed before the RTL was ever consulted.

10. The Sharpest Contrast

The KG "Missing Link" framework [5] is the clearest foil for RAG-SVA, precisely because it is the most advanced expression of the opposite philosophy: maximally RTL-grounded, maximally tool-loop-dependent, correctness emergent from iterative repair. RAG-SVA is spec-first, halt-first, correctness enforced at generation. A direct, named comparison — *constrain-at-generation versus generate-then-repair* — would concede their automation and benchmark rigor while staking RAG-SVA's claim on FV-first construction, halt-on-ambiguity safety, intention-boundary discipline, and rule auditability. That is the position where RAG-SVA is not merely another entry in the list but a genuine alternative to the entire generate-then-repair paradigm.

References

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

[2] Fang et al., AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs, ASP-DAC 2025. https://arxiv.org/abs/2411.14436

[3] Bai, Bany Hamad, Suhaib, Ren (NVIDIA), AssertionForge: Enhancing Formal Verification Assertion Generation with Structured Representation of Specifications and RTL, arXiv 2503.19174, 2025. https://arxiv.org/abs/2503.19174

[4] Paul et al., LISA: LLM Informed SystemVerilog Assertion Generation with RAG and Chain-of-Thought, IEEE ISVLSI 2025. https://www.computer.org/csdl/proceedings-article/isvlsi/2025/11130226/29yER9EzsRO

[5] Knowledge Graphs, the Missing Link in Agentic AI-based Formal Verification, arXiv 2605.06434, 2026. https://arxiv.org/abs/2605.06434

[6] Wu et al., Spec2Assertion: Automatic Pre-RTL Assertion Generation using LLMs with Progressive Regularization, arXiv 2505.07995, 2025. https://arxiv.org/abs/2505.07995

[7] SystemVerilog Assertions (SVA) — survey of LLM-based generation frameworks (ChIRAAG, LAAG-RV, AssertGen, SANGAM, DeepAssert, VERT). https://www.emergentmind.com/topics/systemverilog-assertions-sva

[8] Pulavarthi et al., Are LLMs Ready for Practical Adoption for Assertion Generation?, DATE 2025. https://arxiv.org/abs/2503.10897

Note: external frameworks are cited for comparison and positioning. Summaries are based on each work's own abstract and published description; consult the linked sources for full methodology and results.