Can You Trust AI-Generated Assertions? Catching SVA That Passes but Does Not Match the Spec

A CY62256N SRAM case study with RAG-SVA and the return path

Ben Cohen · Kais Chibani · systemverilog.us · October 31, 2026

Ben_Cohen@SystemVerilog.us

Abstract

An earlier paper[i] described a process that turns a vendor datasheet into an SVA checker, an RTL model and a testbench, then reads the checker back into a line-item requirements specification (the return path). A reader asked the natural follow-up: how do you catch an assertion that passes but does not match the timing in the datasheet? This paper answers that question. It argues that passing is not evidence of correctness, identifies seven factors that set the confidence one can place in AI-generated assertions, and shows, on the CY62256N 32K × 8 asynchronous SRAM, how the return path exposed real mismatches that every simulation had passed.

1. The question

"Interesting that the return path checks the generated SVA against the datasheet. How did you catch cases where the AI wrote an assertion that passed but did not match the SRAM timing spec?"

What is really being asked is the confidence level, or quality, of the assertions. An assertion can pass without being right. It passes against the model, or even in a formal proof, yet what it encodes is not what the datasheet says. That happens in four typical ways:

Failure mode	What happens	Why passing hides it
Too loose	Read data is checked 100 ns after the access starts when the datasheet promises 70 ns	A slow part still passes
Too strict, or over-constrained	An assumption forbids legal environment behavior	The proof never explores that behavior
Vacuous	The assertion's trigger can never occur	It can never fail
Off by one cycle	A nanosecond limit was rounded to cycles the wrong way	The error is inside the property itself

Why "it passed" is not the answer. In this case study the model and the checker came from the same reading of the datasheet. A shared misunderstanding passes in both. Passing shows the SVA and the model agree with each other; it does not show that either one agrees with the datasheet. Catching these errors needs a comparison made outside the code, against the original requirement, and that is what the return path provides.

The short answer

The checker is read back into plain-language requirements written from the SVA code alone ("the SRAM shall drive DQ on the 5th cycle of a read window" rather than `s_rd_window |-> dq_oe`). Each line

names the datasheet parameter it claims to implement, and a reviewer compares the two side by side, in datasheet terms rather than SVA terms. On the CY62256N that comparison found checks that were looser than the datasheet, checks that were stricter, and requirements that were silently missing, all in a checker whose simulations passed. Sections 3 to 9 explain how each part of the process contributes to that result.

2. Seven factors that set the confidence

The quality of the final assertions depends on the whole chain, not on one step:

1. **The requirements:** how they were obtained and how their ambiguities were resolved.
2. **The first generation:** the speed and accuracy of generating the first SVA with RAG-SVA.
3. **The return path:** how the requirements were re-created from the SVA, and how the original requirements were consulted only for what was missing.
4. **The AI model:** its ability to analyze the requirements and the limits of what it can be trusted with.
5. **The return-path document:** its speed, accuracy and format.
6. **The second generation:** the speed and accuracy of re-creating SVA from the return-path requirements.
7. **The human:** the engineer who makes the decisions and signs off.

3. Factor 1: the requirements

Requirements come from many sources, which together define what the DUT must do when exposed to the stimulus driven by its environment. They arrive in many formats, and each format has its own weaknesses.

3.1 Requirements in English

English carries ambiguity, terms that mean different things to different readers, and assumptions left unstated. Consider one sentence:

Upon a new request (req), the ready condition (rdy) shall occur in 3 cycles, and the acknowledge signal (ack) shall be asserted within 2 cycles.

Under the RAG-SVA rules, five questions must be answered before any code is written:

Ambiguity	Possible readings	SVA consequence
"rdy in 3 cycles"	Exactly 3 cycles, or within 3 cycles	`##3 rdy` versus `##[1:3] rdy`
"ack within 2 cycles"	Counted from `req` or from `rdy`; same cycle (0) allowed or not	`##[0:2]` versus `##[1:2]`, anchored on `req` or on `rdy`
"A new request"	The rising edge of `req`, or any cycle in which `req` is high; behavior if another `req` arrives before the handshake ends	`$rose(req)` versus `req`; overlapping transactions allowed or forbidden
Signal signatures	Does `req` stay high until `ack`? Are `rdy` and `ack` one-cycle pulses or levels?	Extra properties on `req` hold and on pulse width
Intention	Is each signal driven by the environment (an input, assumed) or by the DUT (an output, asserted)?	Which properties are `mp_` assumptions and which are `ap_` assertions

The questions interact. If `ack` were counted from `req`, it would have to arrive within 2 cycles, before `rdy` at cycle 3, which is unlikely to be intended; the gate raises that as a consistency question. The intention question matters most for formal verification: if `req` and `rdy` are inputs and `ack` is an output, the requirement mixes an environment constraint with a DUT obligation, and RAG-SVA splits it rather than writing an assumption over an output. Two readings, both legal SVA, show how far apart the answers can be:

```
// Reading A: rdy exactly 3 cycles after the req edge; ack 1-2 cycles after rdy
// req, rdy = inputs (environment); ack = output (DUT)
mp_req_rdy: assume property (@(posedge clk) disable iff (!rstn)
  $rose(req) |-> ##3 rdy);
ap_rdy_ack: assert property (@(posedge clk) disable iff (!rstn)
  $rose(req) ##3 rdy |-> ##[1:2] ack);

// Reading B: rdy within 1-3 cycles; ack 0-2 cycles after rdy
mp_req_rdy: assume property (@(posedge clk) disable iff (!rstn)
  $rose(req) |-> ##[1:3] rdy);
ap_rdy_ack: assert property (@(posedge clk) disable iff (!rstn)
  $rose(req) ##[1:3] rdy |-> ##[0:2] ack);
```

Either checker would pass against a design built to the same reading, and fail against one built to the other. Only the person who owns the requirement can say which is right.

3.2 Timing diagrams, tables and spreadsheets

Datasheets and specifications also express requirements as timing diagrams, parameter tables and spreadsheets. These are more precise than English but have their own problems: a waveform may show one example rather than the rule, a table may give a minimum without saying what it is measured from, and two sources may contradict each other. The CY62256N datasheet, for example, is asynchronous: it has no clock, no reset, and no statement of which side is the device and which is its environment, all of which a formal checker needs.

4. Factor 2: generating the SVA with RAG-SVA

Claude is very good at analyzing complex problems like a puzzle: connecting the objects it is given and working out the relationships between them. RAG-SVA supplies the rules Claude must read and apply before it addresses the task: assertion style, decomposition, labeling, and which signals may appear in assumptions.

With RAG-SVA, Claude stops and asks for clarity on specific points of the requirements, each with proposed alternatives. The engineer picks one of the proposals or supplies an alternative, and the loop repeats until the requirements are clear. At that point the engineer can ask for any of several outputs:

- all assertion statements, packaged as a checker;
- a rewrite of the specification in a chosen format: one an AI can use to generate new assertions, slides, plain simple English, or line items with references to the original requirements;
- a record of every issue that was detected and how it was resolved.

For the CY62256N, the wall-clock timing was as follows (October 20, 2026, Pacific time):

Time	Event	Elapsed
13:48	Datasheet uploaded with the request for an FV checker	—
13:49 to 14:07	Clarification questions raised and answered (ten decisions)	Engineer's answers from 13:56
14:10	Checker delivered, compiled clean	22 min from upload; 14 min from the first answers
14:12 to 14:26	0x0005 = 0xFE request; checker revised; RAM model and testbench (directed and weighted-random) generated, elaborated and simulated	14 min
Total	Datasheet to checker, model and testbench	36 min, including the engineer's interaction time

The engineer's effort went into decisions about the device (which side is the DUT, how to sample an asynchronous part, what to do with a 5 ns parameter under a 20 ns clock) rather than into writing assertions.

5. Factor 3: the return path

The requirements were re-created (the return path) from `cy62256n_sram_checker.sv` alone, which already embodies the clarified requirements, without consulting the PDF. Each requirement states what the SVA actually requires: its windows, cycle counts and conditions.

The checker's own header lists what it deliberately excluded, with the engineer's confirmation: tOHA, tLZOE, tLZCE and tLZWE are 5 ns minimums, shorter than one 20 ns sample, so the sampling clock cannot observe them. Statements written from the checker alone would simply omit them. So once the statements were written, the original datasheet, the only other source available, was consulted for two purposes: to attach the datasheet parameter each statement claims to implement, and to list every datasheet item the checker does not cover. The datasheet is the cross-check, not the source.

Those omissions became explicit GAP line items (REQ-GAP-001 to 006) rather than silence, so a reviewer sees what is not checked as clearly as what is.

The verification requirements document used in Section 9 goes one step further: written at a 5 ns sample period, it brings back the timing items that the 20 ns checker could not observe.

6. How the return path catches passing-but-wrong assertions

Set side by side, the read-back requirements and the datasheet showed where the checker deviated, in a checker whose simulations all passed:

Finding	Datasheet (-70)	What the checker does	Consequence
Looser than spec	tAA, tACE 70 ns max	Data checked on the 5th 20 ns sample of the read window	A part as slow as 80 to 100 ns would pass
Looser than spec	tDOE 35 ns; tHZOE, tHZCE, tHZWE 25 ns max	Checked after 3 samples	Effective limit 40 to 60 ns
Stricter than spec	tPWE 50 ns min	Write must last 3 samples (60 ns)	Some legal write pulses are excluded from the proof
Silently missing	tOHA, tLZOE, tLZCE, tLZWE (5 ns)	Not checked	Now REQ-GAP-001
Silently missing	tRC; read cycles shorter than the window	Not checked	Now REQ-GAP-002
Silently missing	Bus contention (Note 19)	Not visible after the bus split	Now REQ-GAP-004

None of these is a coding error. Each follows from a decision (the 20 ns sampling clock) that was reasonable when it was made. The value of the return path is that it turns each consequence of that decision into a line a reviewer must accept or reject explicitly.

Supporting safeguards

- **Vacuity:** every assertion has a trigger cover (its condition can occur) and a full cover (condition and result together). A vacuous assertion shows up as an unhit cover.
- **Assertions that never fire:** three deliberately broken models were each caught by exactly the assertion that targets their bug.
- **A second, independent checker:** regenerating SVA from the requirements document produced a checker at 5 ns resolution that restored tWC and the low-Z checks (Section 9). Where two checkers from two sources disagree, the difference points to one requirement ID to resolve.

Neither covers nor mutants replace the comparison: an assertion can be non-vacuous, catch a seeded bug, and still encode the wrong number.

7. Factor 4: the AI model

The model's strength is analysis: reading a vendor datasheet, relating its tables, waveforms and notes, and finding what is missing or inconsistent. Its weakness is that it can be confidently wrong, and it cannot judge what the requirement's owner intended. RAG-SVA narrows the space in which it can be wrong: fixed style, forbidden operators, mandatory covers, and gates that stop for the engineer's decision instead of guessing. One limit remains by construction: when the same model writes the checker, the model and the read-back, a single misreading can pass through all three. The return path addresses that by putting every claim in front of a human, next to its datasheet reference.

8. Factor 5: the return-path document

Format. Each requirement is one line item: a stable ID, one atomic "shall" statement, a type (ENV, DEV, REACH, GAP, SCN), the datasheet reference, the SVA label that implements it, and a verification status. In that form the specification can be reviewed line by line and imported into commercial planning and traceability tools.

Accuracy. Because it is written from the code, it is accurate about the code by construction; its accuracy against the datasheet is exactly what the review in Section 6 measures.

Speed. Creating the return-path document took minutes: less than 8.

9. Factor 6: re-creating SVA from the return-path requirements

A second checker, `cy62256n70_sva.sv`, was generated from `CY62256N-70_Verification_Requirements` (v1.0) alone. Every property is named after the requirement it implements (ENV-01 to ENV-05, REQ-01 to REQ-06, COV-01 to COV-10), and its header carries the PORT-INTENT table. At 5 ns resolution it checks tWC and the low-Z times that the first checker dropped, and adds a dedicated OE-controlled read. It elaborates with no diagnostics in slang; it has not yet been simulated.

Speed. Generating `cy62256n70_sva.sv` also took minutes: less than 10.

10. Factor 7: the human

The engineer remains the owner of the requirement. In this flow the engineer: answered the clarification questions; chose how to sample an asynchronous part and accepted the consequences; decided that the 0x0005 = 0xFE request belongs in a separate scenario plan, not the sign-off plan; and reviewed the read-back requirements against the datasheet. The AI writes the assertions and the read-back; the human decides whether each line says what the chip must do, and signs off.

11. Limits

- The checkers have not yet been run in a commercial simulator or formal tool. The read-back catches wrong intent; the tools must still confirm the SVA executes as intended.
- The model, the first checker and the read-back come from one reading of the datasheet. The review reduces that risk; it does not remove it.
- Cover hit counts have not yet been measured in an SVA-capable tool.

12. Conclusion

A passing assertion proves agreement, not correctness. The way to catch an AI-generated assertion that passes but misses the spec is to compare it with the spec in the spec's own language. The return path makes that comparison practical: it turns the checker into line items, each tied to a datasheet parameter, so a reviewer can see in minutes where the checks are looser, stricter or missing. On the CY62256N it found all three kinds, in a checker that had passed every simulation.

What matters most is that the engineer's effort goes into decisions about the device rather than into writing assertions.

RAG-SVA is distributed with the book *Knowledge-Driven AI for SystemVerilog Assertions*. Commercial use requires a commercial license; the license is free for educational use.

13. Related papers

The papers below, all available at https://systemverilog.us/#papers, develop the ideas this paper relies on.

Requirements and ambiguity

- **It's the Requirements, STUPID!**: why assertion quality is bounded by requirement quality. https://systemverilog.us/ItsTheRequirementsStupid.pdf
- **From Ambiguous Requirement to Verified Assertions**: a requirement template that removes ambiguity before code is written. https://systemverilog.us/Requirement_Template_Paper.pdf
- **It's the Source, Stupid!**: how the source of a requirement shapes the assertions generated from it. https://systemverilog.us/Its_the_Source_Stupid.pdf

Environment versus obligation (the req/rdy/ack example)

- **Degeneracy, Environment vs. Obligation, and Design Intent: A req/ack Case Study**. http://systemverilog.us/req_ack_degeneracy_paper.pdf
- **Deriving a Formal req/ack Assertion Under the RAG-SVA Methodology**. http://systemverilog.us/req_ack_hold_window_reasoning.pdf

The RAG-SVA methodology

- **RAG-SVA: A Paradigm Shift in Formal Verification**. https://systemverilog.us/Paper_June2426.pdf
- **It's the Methodology, Stupid! Meaningful SVA: Write for Formal, Run Anywhere**. https://systemverilog.us/Its_the_Methodology_Stupid.pdf
- **Gen-SVA: Source-Agnostic Generation of Formal-Ready SVA**: generating SVA from requirements in any format. https://systemverilog.us/Gen-SVA_Paper_Cohen_KAIS.pdf
- **Determinism in SVA for Formal Verification: One Property, One Path, One Outcome**. https://systemverilog.us/Determinism_SVA_FV_Paper.pdf
- **FV Abstraction Techniques in RAG-SVA**: including the symbolic-address technique used for the SRAM data-integrity check. http://systemverilog.us/fv_abstraction_paper.pdf

Trusting AI output

- **Frame Drift in Multi-Turn LLM Interaction, and Why RAG-SVA Rule Corpora Already Defend Against It**. https://systemverilog.us/Frame_Drift_and_RAG_SVA_Discipline.pdf
- **RAG-SVA in the Landscape of LLM-Based Assertion Generation**. https://systemverilog.us/V4_RAG-SVA_Landscape_Comparison.pdf
- **OVL Arbiter: Claude With vs. Without SVA-RAG for Formal Verification**: the same design checked two ways. https://systemverilog.us/ovl_arbiter_ai_rag2.gz

Testbenches and the full index

- **RAG-SVA Testbench Styles: One Checker, Directed, Random, and UVM**. https://systemverilog.us/RAG-SVA_Testbench_Styles.pdf
- **Mapping the RAG-SVA Body of Work (Revision 2)**: a classified index of all the papers, with links. https://systemverilog.us/Mapping_the_RAG-SVA_Body_of_Work.pdf

---

[i] **From Datasheet to Traceable Requirements**: **The CY62256N SRAM Example**
One AI-assisted process: $Requirements \rightarrow Claude\ Opus$ 5.5 + $RAG\text{-}SVA \rightarrow Verified\ SVA\ checker, RTL\ model\ and\ testbench$
$\rightarrow read\text{-}back\ spec \rightarrow commercial\ tools$
https://systemverilog.us/V5_Vendor_Datasheet_to_Traceable_Requirements_The_CY62256N_SRAM.pdf