

The RAG-SVA Paradigm Shift in Formal Verificationⁱ

Ben Cohen & Dr. Kais Chibani

1. Introduction: The Unique and Focused Mandate of RAG-SVA

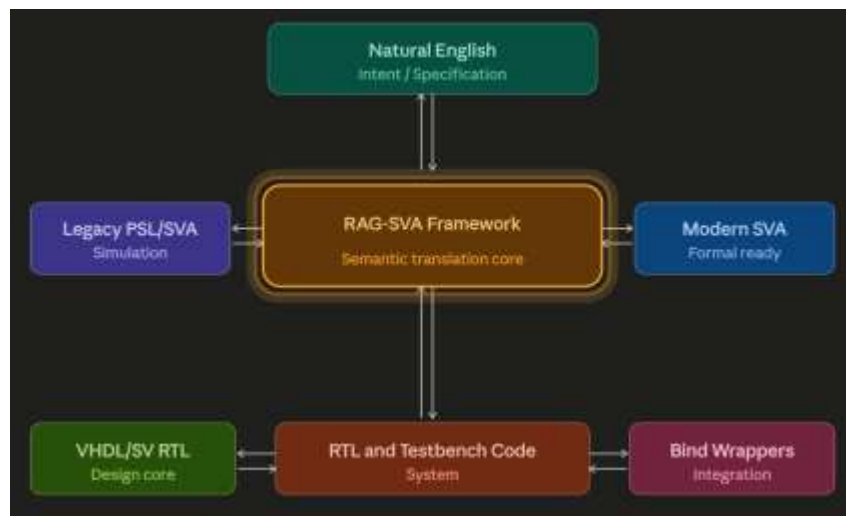
In traditional hardware verification, Large Language Models (LLMs) are frequently applied as simple text-completion engines. When prompted to generate SystemVerilog Assertions (SVA), an unguided AI acts primarily on autocomplete mechanics—retrieving generic syntactic patterns from its training weights, filling specification ambiguities silently, and generating code without structural guardrails. This velocity-biased approach introduces critical design risks: it leaves syntax vulnerabilities unchecked and fails to distinguish between design inputs and outputs.

To resolve this vulnerability, we introduced a unique, highly specialized framework: **Retrieval-Augmented Generation for SystemVerilog Assertions (RAG-SVA)**ⁱⁱ. Rather than treating AI as an autonomous, unguided coder, the **RAG-SVA framework** establishes a deterministic engineering environment. By leveraging a strict multi-tier guardrail system—spanning from input specification parsing to post-generation token scans—it forces the language model to act as a rigorous specification auditor and precise compliance generator.

The defining value of the **RAG-SVA framework** lies in its absolute versatility across multiple architectural configurations and verification scenarios. Whether processing legacy code modernization or driving new microarchitectural specifications, the framework successfully ensures mathematical stability, tool safety, and functional accuracy.

2. Multi-Directional Universality: The Representational Translation Loop

A foundational tenet of the *Cohen-Chibani* methodology is that single representational mediums are inherently blind to specific classes of engineering defects. A requirement reviewed solely in natural English lacks cycle-exact timing clarity; conversely, a raw SVA property isolates timing but hides human intent and high-level design rationale. The **RAG-SVA framework** addresses this limitation by functioning as a multi-directional translator across core engineering representations:



English ↔ SVA ↔ PSL ↔ RTL ↔ VHDL

By establishing formal relationships between these formats, **RAG-SVA** enables highly flexible automation workflows:

- **Natural English to Formal-Safe Logic:** Ingests unconstrained microarchitectural intent, isolates domain ambiguities, and compiles mathematically stable SVA properties.
- **Cross-Language Translation (PSL \leftrightarrow SVA):** Translates legacy Property Specification Language (PSL) models into modern SystemVerilog semantics without sacrificing verification intent.
- **Hardware to Checker Alignment (RTL/VHDL \leftrightarrow SVA):** Extracts port definitions and signal structural relationships directly from RTL source blocks (whether written in SystemVerilog or VHDL) to build accurate testbench infrastructures and target assertions.
- **Back-Translation for Human Audit:** Automatically converts generated formal code back into structured "Formal English" requirements, allowing human stakeholders to audit the precise mathematical representation before finalizing silicon designs.

3. New Requirements Evaluation and Clarification

When standard AI models ingest an initial human prompt, their reinforcement learning loops bias them toward immediate code execution. Gaps in the specification are bridged by silent assumptions, which often mask serious design bugs.

The **RAG-SVA framework** eliminates this behavior via the **RQM0 Ambiguity Quality Gate**. Code generation is completely blocked until the requirements are systematically audited against six distinct categories of ambiguity:

- **Signal & Width Omissions:** Pinpointing generic array parameters (e.g., tracking multi-bit request/grant vectors) and enforcing concrete parameter configurations ($N=2$) before writing properties.
- **Temporal Bounds & Waveform Windows:** Flagging unconstrained or vague timing instructions (e.g., "*a grant must occur after a request*") and requiring the explicit assignment of minimum and maximum latency values.
- **Operational Modes & Tie-Breakers:** Identifying multi-mode options within protocol definitions (such as Round-Robin vs. FIFO arbitration rules) and forcing an explicit selection.
- **Asynchronous Gating Behavior:** Clarifying clock and reset interactions, such as confirming whether an enable port acts via clock-gating or reset-gating mechanics.

Once cleared, the **RAG-SVA framework** mandates the creation of a comprehensive **PORT-INTENT Table** (RQM2). This table classifies every signal based on its true structural driving source, rather than basic wire direction. Environment-driven inputs are explicitly mapped to assume conditions, while design-driven outputs and tracked states are bounded strictly as obligation targets.

4. Legacy Conversion & Modernization

A massive bottleneck in modern design cycles is the presence of legacy assertion suites written decades ago for simulation-centric execution loops (e.g., PSL code from 2004 or legacy SVA from 2009). These older properties frequently contain constructs that cause formal verification engines to stall, time out, or pass vacuously.

The **RAG-SVA framework** automates this conversion process in a systematic, rule-enforced manner. It parses legacy syntax and translates it according to modern, tool-safe formal guidelines:

Eliminating Tool-Crashing Patterns

Legacy simulation code frequently relies on unbounded delays and repetitions (such as `##[1:$]` or `[*0:$]`) or specialized operators (like `throughout` and `until`) to check continuous conditions across long simulation runsⁱⁱⁱ. In formal tools, these unconstrained bounds cause state-space explosion. The **RAG-SVA framework** replaces these patterns with modern, formal-safe equivalents:

- Legacy `until` constructs are translated into deterministic structural windows:
`signal[*1:N] intersect event[->1].`
- Simulation-centric unbounded delay windows are converted into finite, parameter-driven bounds.
- The non-overlapping implication operator (`|=>`) is fully banned to prevent cycle-interpretation errors, standardizing the property library strictly on the overlapping format (`|-> ##1`).

5. Formal Verification Ready (FV-Ready) SVA

An assertion that compiles without syntax errors can still be fundamentally flawed for formal verification. Writing "FV-Ready" SVA requires careful management of the mathematical properties of the formal tool's proof space. The *Cohen-Chibani* methodology ensures this by enforcing explicit structural boundaries through the **RAG-SVA framework**.

Protecting the Intention Boundary

The most severe mistake an unguided LLM or inexperienced engineer can make is placing an obligation signal (a DUT output) inside an environment assumption's consequent. For example:

```
// ❌ CRITICAL OVER-CONSTRAINT VIOLATION (Banned by RAG-SVA Framework)
assume_gnt_pulse: assume property (gnt |-> ##1 !gnt);
```

Because `gnt` is driven by the DUT, forcing this condition inside an `assume` statement instructs the formal engine to constrain the design's outputs. The tool prunes all states where the design would have violated this pulse rule, reporting a false green **PROVEN** status while masking active hardware bugs. The **RAG-SVA framework** enforces strict lookup structures to ensure design outputs are strictly verified using assertions (`ap_`).

Liveness and Bounded Convergence (STG9 / FV5)

Pure liveness claims (e.g., *"Every request must eventually receive a grant"*) can pass vacuously in formal verification if the tool models an infinite trace where the event never occurs, or if it stalls on an unbounded sequence. The **RAG-SVA framework** addresses this via a dual approach:

1. **Form A (Bounded Convergence):** Converts unbounded liveness language into a bounded format using a finite parameter (`MAX`). This technique enables the formal solver to resolve the proof within a finite window while leveraging `strong()` semantics to catch trace-termination errors.
2. **Form B (Unbounded Infrastructure):** When pure liveness is required, the framework enforces the inclusion of the `strong()` operator, which allows the property to be falsified by an infinite loop counterexample.

(See Section 8 for a complete example)

6. Mathematical Soundness: Fairness and Anti-Livelock Infrastructure

When executing unbounded or complex temporal proofs, a formal engine will actively search for any environment state that can falsify an assertion or cause a vacuous proof pass. If an input signal is allowed to stay in a blocking state indefinitely, the formal tool will exploit that freedom, resulting in a false negative or a vacuous proof. The *Cohen-Chibani* framework structurally resolves this by generating specialized companion constraints:

FV8 Activation & Blocking Fairness (mp_fair_*)

For every liveness property, the **RAG-SVA framework** executes a scan across the entire PORT-INTENT table to locate any input signal that could block design progress. It then generates a dedicated fairness constraint to prevent the tool from locking that signal:

```
mp_rdy_fair: assume property (@(posedge clk) s_eventually rdy);
```

This constraint prevents the formal tool from holding `rdy=0` indefinitely, which would otherwise stall the design and block valid state transactions.

FV9 Anti-Livelock Constraints (mp_liveLock_*)

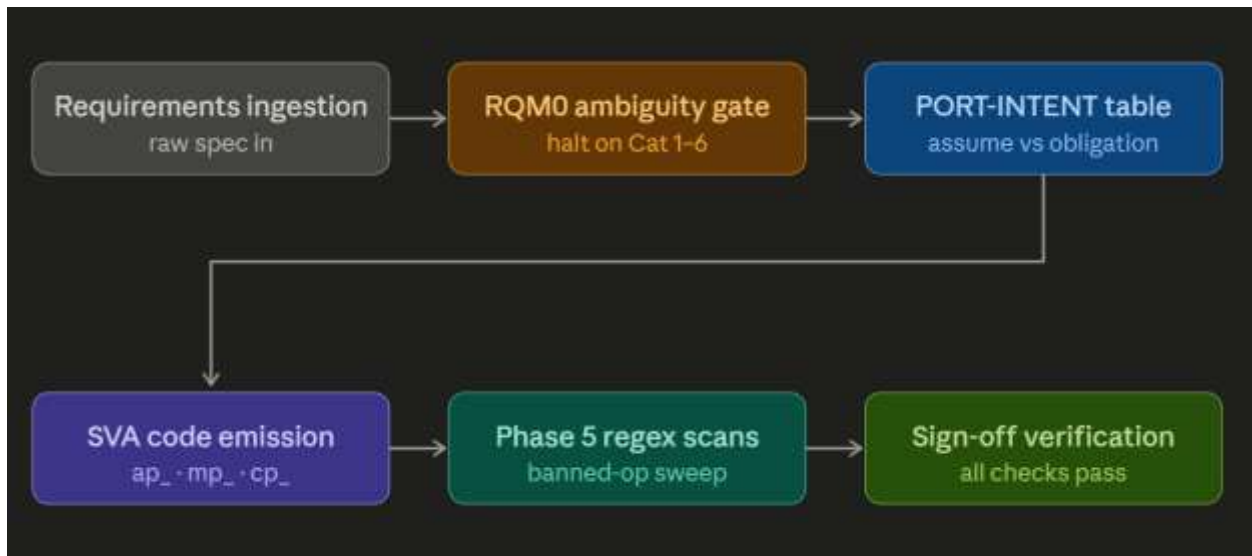
If an environment input can toggle continuously, it can repeatedly restart a property's antecedent, preventing the in-flight transaction from ever reaching completion. The **RAG-SVA framework** builds targeted anti-livelock assumptions to prevent this behavior:

```
mp_req_no_livelock: assume property (@(posedge clk) s_eventually always !new_req);
```

This forces the formal solver to eventually stabilize environmental triggers, giving the design under test the cycles it needs to complete its internal operations and deliver an output.

7. Quality Gates and Verification Sign-Off Documentation

A key strength of the **RAG-SVA framework** is its strict post-generation verification and documentation pipeline, which ensures that no property is left un-vetted before tool execution.



Complete UR-5 Coverage Pairing Compliance

An assumption that is too restrictive can silently blind a formal tool by cutting off valid exploration paths. To prevent this form of vacuity, the **RAG-SVA framework** enforces the **UR-5 Pairing Rule** (*Unreachability Prevention Rules UR-5 — mandatory coverage pairing*). Every generated environment assumption (mp_) must be paired with a corresponding hit coverage property (cp_*_hit):

```
cp_rdy_fair_hit: cover property (@(posedge clk) rdy);
```

If a formal tool run reveals that an assumption's companion coverage point is unreachable, it indicates that the proof space has been over-constrained or that the signal has been classified incorrectly.

Mandatory DOC4 / DOC9 Traceability Headers

To establish clear traceability from natural language specifications to compiled hardware verification checkers, the **RAG-SVA framework** mandates that every generated environment assumption features a comprehensive DOC9 structural header block:

```
/* ASSUMPTION: rdy must eventually assert on every explored trace.  
TYPE:          Fairness/Blocking-Signal (FV-FA-1)  
JUSTIFICATION: rdy=0 stalls the DUT. Tool holding rdy=0 forever creates a false negative.  
RISK LEVEL:   Low */
```

This documentation layer ensures that all verification assumptions are clearly justified, reviewable by human engineers, and fully traceable back to the source specification.

8. Worked Example: From Requirement to Verified SVA

Actual session with Using Claude Sonnet 4.6:

[User] Every req must eventually receive an ack req is an input, ack an output. rst_n is active low, there is a posedge clk req is a pulse

[Claude] ⚠ Thread is long – re-anchoring rules now.

⚠ Thread is long – re-anchoring rules now. GATE_MODE_2 active. MAX=16.

CHECK 0 – continue?

Phase 1 – RQM0 Quality Gate

Requirement text: "Every req must eventually receive an ack. req is a pulse."

Keyword scan:

- "eventually" → LIVENESS (FV-SW-2) → FV-FA + FV-LL required
- "every req" → activation trigger = \$rose(req)
- "must" with bound? → MAX=16 confirmed in Pre_instructions_for_FV.md → FORM A bounded liveness
- "req is a pulse" → PULSE-BASED (CHECK 12/STG12) → add \$rose(req) |-> ##1 !req assumption
- "ack" → output, obligation, response
- "req" → input, assume, trigger
- rst_n active-low → disable iff (!rst_n)
- clk posedge

Category 1-6 scan:

- Cat-1: signals clear
- Cat-2: "eventually" → resolved by MAX=16
- Cat-3: no boundary ambiguity
- Cat-4: no contradiction
- Cat-5: single implication
- Cat-6: no syntax ambiguity

Phase 1 – continue?

Phase 2 – PORT-INTENT TABLE

PORT	DIRECTION	TYPE	NAME	INTENTION	ACTIVITY
PORT	input	logic	clk	N/A	clock
PORT	input	logic	rst_n	N/A	reset
PORT	input	logic	req	assume	trigger
PORT	output	logic	ack	obligation	response

INTENTION boundary check:

- Antecedent: \$rose(req) → assume ✓
- Consequent: ack → obligation ✓
- CASE A → ap_label ✓
- No mixed consequent ✓
- No obligation in antecedent ✓

CHECK 18 – Vector index completeness:

All signals are scalar logic (no [N:0]) → CHECK 18 does not apply ✓

§4.0 Three-Axis Classification:

Axis Result

Axis 1 LIVENESS – keyword "eventually" (FV-SW-2)

Axis 2 BOUNDED (MAX=16, no \$) → strong() NOT required in FV; recommended for simulation

Axis 3 BOUNDED ##[1:16] → finite N → resolves within MAX cycles

Class: LIVENESS-B (bounded liveness, intent-driven by "eventually")

→ FV-FA + FV-LL companion analysis required despite bounded form.

§4.7 – 7-question self-check:

# Question	Result
1 All signals in Port-Intent table?	✓
2 Label prefix consistent with §4.1?	✓ ap_
3 Class identified by keyword scan, not code?	✓ "eventually" → LIVENESS
4 Bounded (no \$) → strong() absent / acknowledged redundant?	✓
5 All covers accounted for?	✓ trigger + full + UR-5 pairs
6 FV-FA-0 run on full Port-Intent table?	✓ – see below
7 FV-LL analysis run on all cycling assume inputs?	✓ – see below

FV-FA-0 scan – full Port-Intent table (assume signals only):

Signal	Can block consequent?	False-negative strategy	Action
req	YES – held LOW: antecedent never fires → vacuous pass	Activation blocking	L3-B: s_eventually req

FV-LL analysis – req is pulse-based:

- "req is a pulse" → pulse protocol (not hold-until-completion)
- FV-LL-6 decision: pulse protocol → use bare signal s_eventually always !req
- No auxiliary new_req needed (FV-LL-6: pulse path)

Phase 2 ✓ – PORT-INTENT shown above – proceeding to Phase 3

Phase 3 – Syntax Validation (CHECK 1-18)

Pre-generation scan on planned patterns:

Check	Pattern planned	Result
CHECK 2.5	No => – using -> ##1	✓
CHECK 4	Window: ##[1:16] ✓; Pulse hold: req[*1:1] → ##1 !req ✓	✓
CHECK 5	Clock on statement only	✓
CHECK 6	No naked intersect ##	✓
CHECK 7	No until, ##[0:\$], firstmatch, within	✓
CHECK 10 UR-1	All mp_ have ->	✓
CHECK 10 UR-2	All mp_ use implication	✓
CHECK 10 UR-3	\$rose(req) used as antecedent trigger only	✓
CHECK 10 UR-5	Every mp_ paired with cp_*_hit	✓
CHECK 11	No ##[0:\$] or degenerate patterns	✓
CHECK 12	req is pulse-based (user stated) → ##1 !req assumption legal	✓
CHECK 13.5	All properties use p_ prefix	✓
CHECK 15	DOC9 on every mp_* (FV mode YES)	✓ planned
CHECK 16.1	Single trigger path, single outcome	✓
CHECK 16.2	All bounds finite (MAX=16)	✓
FV-SW-1	No \$ in consequent → strong() not required	✓
FV-SW-2	Liveness confirmed by keyword "eventually"	✓
LNG-HIO	No intersect pairing → not applicable	✓

Phase 3 ✓ – continue?

Phases 4, 5, 6 running silently...

Write the FV checker SVA file

Write the FV checker SVA file

[User]: continue

[Claude]: Phase 7 ✓ – done.

Ran 2 commands, read a file

Ran 2 commands, read a file

Phase 7 ✓ – done.

Summary – 5 properties, 5 covers, all Phase 5 scans PASS:

ID	Label	Class	Rule basis
P1	ap_req_eventually_ack	LIVENESS-B	\$rose(req) -> ##[1:16] ack – bounded, FV-SW-1 no \$ → strong() not required
P2	mp_req_pulse	SAFETY/Input Constraint	\$rose(req) -> ##1 !req – CHECK 12 pulse path
P3	mp_req_fair	FAIRNESS/Activation	s_eventually req – FV-FA-1 L3-B, prevents vacuous pass
P4	mp_req_no_livelock	DEADLOCK	s_eventually always !req – FV-LL-3, pulse protocol → bare signal (FV-LL-6)

ID	Label	Class	Rule basis
C1-C5	cp_*	COVERAGE	Items 9+10 for ap_; UR-5 pairs for all mp_

Fairness gate active –

waiting for your selection: A/B. No additional properties will be generated until you choose.

A) Yes – generate all required fairness constraints (s_eventually for livelock prevention)

B) No – skip fairness constraints

[User] Yes

Fairness constraints are already embedded in the file as P3 (mp_req_fair) and P4 (mp_req_no_livelock), generated in the same pass per the Pre_instructions_for_FV.md single-pass rule.

No additional properties needed – the FV-FA-0 scan found only one assume signal (req), which is covered by both assumptions.

Fairness gate: closed ✓

CHECK FV-FA-3 verification:

No pure Safety-class property set – P1 is LIVENESS-B → fairness/livelock assumptions are permitted ✓

CHECK FV-FA-4:

cp_req_fair_hit covers req (non-blocking state) – reachability diagnostic in place ✓

CHECK FV-FA-5:

One fairness assumption per signal per failure mode – no duplicates ✓

CHECK FV-LL-3:

mp_req_no_livelock present ✓

CHECK FV-LL-5/LL-6:

Pulse protocol confirmed → bare !req form used → no new_req auxiliary needed ✓

File arbiter_req_ack_fv.sv is complete and closed. Ready for next requirement.

```
// =====
// FILE      : arbiter_req_ack_fv.sv
// LICENSE    : Personal/academic use only. Commercial use requires a Commercial License.
// DESCRIPTION : FV checker – req must eventually receive ack (LIVENESS-B, bounded)
// GENERATED  : Per RAG_RULES + DoThisFirst.md v2.0 + Pre_instructions_for_FV.md
// FV MODE    : YES | DECOMP: ATOMIC | MAX: 16 | VERBOSITY: !C
// -----
// PROPERTY INVENTORY
// -----
// ID | Label | Class | Type
// ---|-----|-----|-----
// P1 | ap_req_eventually_ack | LIVENESS-B | assert – bounded response
// P2 | mp_req_pulse | SAFETY | assume – req 1-cycle pulse
// P3 | mp_req_fair | FAIRNESS | assume – activation fairness
// P4 | mp_req_no_livelock | DEADLOCK | assume – livelock prevention
// C1 | cp_req_eventually_ack_trig | COVERAGE | cover – Item 9 antecedent
// C2 | cp_req_eventually_ack_full | COVERAGE | cover – Item 10 full sequence
// C3 | cp_req_pulse_hit | UR-5 PAIR | cover – mp_req_pulse
// C4 | cp_req_fair_hit | UR-5 PAIR | cover – mp_req_fair (FV-FA-4)
// C5 | cp_req_no_livelock_hit | UR-5 PAIR | cover – mp_req_no_livelock
```

```

// -----
// UR SCAN TABLE
// -----
// mp_req_pulse      | UR-1: PASS (has |->) | UR-2: PASS | UR-3: PASS ($rose antecedent)
//                   | UR-4: N/A           | UR-5: PASS (cp_req_pulse_hit)
// mp_req_fair       | UR-1: EXC (global s_eventually invariant - no |-> needed)
//                   | UR-2: EXC           | UR-3: N/A  | UR-4: N/A
//                   | UR-5: PASS (cp_req_fair_hit)
// mp_req_no_livelock | UR-1: EXC (global s_eventually always invariant)
//                   | UR-2: EXC           | UR-3: N/A  | UR-4: N/A
//                   | UR-5: PASS (cp_req_no_livelock_hit)
// =====
`ifndef SYNTHESIS
module req_ack_fv_checker (
    input logic clk,
    input logic rst_n,
    input logic req,
    input logic ack
);
// =====
// PARAMETERS
// =====
localparam int MAX = 16; // user-confirmed bound (Pre_instructions_for_FV.md)

// =====
// §4.0 CLASSIFICATION RECORD
// Axis 1 : LIVENESS - keyword "eventually" (FV-SW-2)
// Axis 2 : BOUNDED - finite MAX=16, no $ in consequent
//                   strong() NOT required in FV (resolves within MAX cycles)
//                   strong() RECOMMENDED in simulation (end-of-trace detection)
// Axis 3 : BOUNDED - ##[1:MAX] with finite N
// FV-FA : REQUIRED - intent-driven by "eventually" (FV-SW-2)
// FV-LL : REQUIRED - req is pulse -> bare signal form (FV-LL-6 pulse path)
// =====

// =====
// P1 - ASSERTION | CLASS: LIVENESS-B | §4.1 CASE A (assume->obligation)
// Requirement : Every req must eventually receive an ack (MAX=16 cycles)
// Antecedent : $rose(req) INTENTION=assume ✓
// Consequent : ##[1:MAX] ack INTENTION=obligation ✓
// strong() : omitted in FV (bounded); present in simulation form below
// Note: With Claude Opus 4.8 rerun of code:
// FV-SW-1 : bounded, no $ in consequent -> strong() not required in FV

// =====
property p_req_eventually_ack;
    $rose(req) |-> ##[1:MAX] ack;
endproperty

ap_req_eventually_ack: assert property (
    @(posedge clk) disable iff (!rst_n)
    p_req_eventually_ack
    else $error("[FAIL] ap_req_eventually_ack: ack not seen within %0d cycles of
    $rose(req) at time %0t", MAX, $time);

```

```

// =====
// P2 – ASSUMPTION | CLASS: SAFETY | mp_ (assume→assume, CASE B)
// req is a 1-cycle pulse (user-stated: CHECK 12 / STG12 pulse path)
// DOC9 header below
// UR-1: PASS (has |->) | UR-2: PASS | UR-3: PASS | UR-5: cp_req_pulse_hit
// =====
/* ASSUMPTION : req is a 1-cycle pulse – deasserts one cycle after rising.
TYPE : Input Constraint
JUSTIFICATION: User stated "req is a pulse" – CHECK 12 pulse path applied.
Prevents tool from holding req high across multiple cycles,
which would distort the activation pattern.
RISK LEVEL : Low */
property p_req_pulse;
    $rose(req) |-> ##1 !req;
endproperty

mp_req_pulse: assume property (
    @(posedge clk) disable iff (!rst_n)
    p_req_pulse);

// =====
// P3 – ASSUMPTION | CLASS: FAIRNESS | FV-FA-1 / L3-B Activation
// FV-FA-0 scan result: req held LOW forever → antecedent never fires →
// ap_req_eventually_ack passes vacuously.
// Form: s_eventually req (L3-B activation fairness – signal LOW is blocking)
// UR-1: EXC (global invariant) | UR-2: EXC | UR-5: cp_req_fair_hit
// =====
/* ASSUMPTION : req must eventually assert on every explored trace.
TYPE : Fairness/Activation
JUSTIFICATION: Tool can hold req=0 forever. ap_req_eventually_ack then
passes vacuously – antecedent $rose(req) never fires.
s_eventually req prohibits the "hold req=0 forever" strategy.
RISK LEVEL : Low */
mp_req_fair: assume property (
    @(posedge clk) disable iff (!rst_n)
    s_eventually req);

// =====
// P4 – ASSUMPTION | CLASS: DEADLOCK | FV-LL-3 / L4
// FV-LL-6 decision: req is PULSE protocol → bare signal form is safe.
// "s_eventually always !req" is consistent with pulse behavior:
// req already deasserts after 1 cycle (mp_req_pulse), so repeated
// req pulses cycling indefinitely defeat liveness.
// No auxiliary new_req needed – pulse path per FV-LL-6.
// UR-1: EXC (global invariant) | UR-2: EXC | UR-5: cp_req_no_livelock_hit
// =====
/*
ASSUMPTION : At some future point req will permanently stop asserting,
allowing the in-flight transaction to complete.
TYPE : Fairness/Blocking-Signal
JUSTIFICATION: req is a pulse. Repeated pulses can restart the ack
obligation indefinitely. s_eventually always !req forces
the tool to explore traces where req eventually quiescens,
permitting ack to settle within MAX cycles.
Pulse protocol confirmed → bare !req form (FV-LL-6).
RISK LEVEL : Low
*/

```

```

mp_req_no_livelock: assume property (
    @(posedge clk) disable iff (!rst_n)
    s_eventually always !req);

// =====
// COVERAGE
// =====

// C1 – Item 9: Antecedent reachability for ap_req_eventually_ack
cp_req_eventually_ack_trig: cover property (
    @(posedge clk) disable iff (!rst_n)
    $rose(req));

// C2 – Item 10: Full sequence execution for ap_req_eventually_ack
cp_req_eventually_ack_full: cover property (
    @(posedge clk) disable iff (!rst_n)
    $rose(req) ##[1:MAX] ack);

// C3 – UR-5 pair for mp_req_pulse
cp_req_pulse_hit: cover property (
    @(posedge clk) disable iff (!rst_n)
    $rose(req) ##1 !req);

// C4 – UR-5 / FV-FA-4 pair for mp_req_fair
//     Covers non-blocking state of req (req=1)
//     If unreachable: another assumption over-constrains req – investigate
cp_req_fair_hit: cover property (
    @(posedge clk) disable iff (!rst_n)
    req);

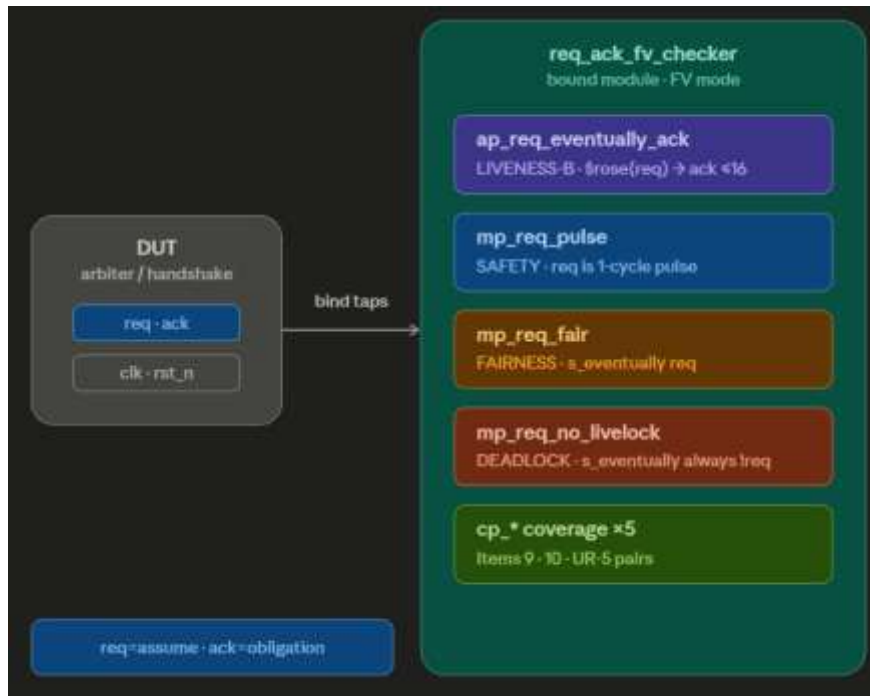
// C5 – UR-5 pair for mp_req_no_livelock
//     Covers !req (livelock-free state reachable)
cp_req_no_livelock_hit: cover property (
    @(posedge clk) disable iff (!rst_n)
    !req);

endmodule : req_ack_fv_checker

// =====
// BIND STATEMENT
// Replace <dut_module_name> with the actual DUT module name.
// =====
// bind <dut_module_name> req_ack_fv_checker u_req_ack_fv_checker (
//     .clk (clk),
//     .rst_n (rst_n),
//     .req (req),
//     .ack (ack)
// );

`endif // SYNTHESIS
// =====
// END OF FILE : arbiter_req_ack_fv.sv
// =====

```

The diagram shows the structure: the DUT on the left exposes its four interface signals (req, ack, clk, rst_n); the bind construct taps them into the checker module on the right, which holds the four property classes plus the coverage group.

The key architectural point is that the checker only observes – every arrow runs DUT → checker. None of the property blocks drive anything back. req enters as assume (environment constraint), ack enters as obligation (DUT must satisfy), and the four classes partition the verification job:

- ap_req_eventually_ack – the one assertion (the DUT's obligation)
- mp_req_pulse, mp_req_fair, mp_req_no_livelock – three assumptions that constrain the environment so the proof converges without false negatives
- cp_* x5 – coverage proving the assumptions are satisfiable and the assertion is exercised

Each box is clickable if you want the detail on any one. Want me to also draw the internal signal flow – how \$rose(req), the pulse constraint, and the fairness/livelock assumptions interact on a timeline inside the checker?

[User - click on] Explain ap_req_eventually_ack

p_req_eventually_ack is the one true assertion in the file – the DUT's obligation. Everything else (the three mp_ assumptions, the five cp_ covers) exists to make *this* property provable and meaningful.

Here's the property:

```
systemverilog
property p_req_eventually_ack;
  $rose(req) |-> ##[1:MAX] ack;          // MAX = 16
endproperty
```

```
ap_req_eventually_ack: assert property (
  @(posedge clk) disable iff (!rst_n)
  p_req_eventually_ack)
  else $error("[FAIL] ap_req_eventually_ack: ack not seen within %0d cycles of $rose(req) at
time %0t", MAX, $time);
```

What it says in plain terms: every time req rises, ack must appear somewhere in the next 1 to 16 cycles. If 16 cycles pass after a \$rose(req) with no ack, the assertion fails and prints the cycle count and timestamp.

Reading the pieces:

- \$rose(req) – the antecedent (trigger). req is an assume signal (input), and an edge is a legal trigger. It fires the cycle req transitions 0→1.
- |-> – overlapping implication. The consequent begins checking from the same cycle the trigger matches. Your RAG set bans |=>; the explicit |-> ##1 form (or |-> ##[1:N] here) is the required style.
- ##[1:MAX] ack – the consequent (obligation). ack is an obligation signal (output). The window is 1 to 16 cycles – a timing window, so ##[1:N] is correct (not [*N], which would be a hold-duration).
- disable iff (!rst_n) – the whole check is suppressed while reset is asserted (active-low).

Why it's bounded and not s_eventually: the requirement said "eventually," which by your FV-SW-2 rule is **intent-based LIVENESS**. But liveness in an assert consequent can't use s_eventually or `##[1:]` without `strong()` – both are banned in your ruleset. So the canonical move (\$4.0 / FV-SW-1, the convergence technique) **converts the unbounded "eventually" into a bounded `##[1:16]` with a user-supplied MAX. Because the bound is finite – no ` ` in the consequent – strong() is not required in FV;** the tool resolves it within 16 cycles regardless.

This is the subtle part worth holding onto: the code *looks* like a plain bounded safety check, but it's classified LIVENESS-B. That classification is what forces the companion assumptions to exist. A bounded ##[1:MAX] derived from "eventually" still needs the full FV-FA + FV-LL set, because the bound solves tool termination but not the false-negative problem:

Without the companion...	The tool could...	Caught by
mp_req_fair	hold req=0 forever → trigger never fires → vacuous pass	C1, C4
mp_req_no_livelock	pulse req endlessly → restart the obligation, never settle	C5
mp_req_pulse	hold req high across cycles → distort the activation pattern	C3

One honest caveat on the bound itself: ##[1:16] is stronger than true "eventually." It demands ack within exactly 16 cycles, not merely someday. If the real DUT can legitimately take 17+ cycles under some legal input sequence, this assertion will report a failure that isn't a real bug – it's the bound being too tight. The fix in that case isn't to loosen the assertion blindly; it's to confirm with the designer what the genuine worst-case latency is and set MAX from that. Right now MAX=16 comes from your Pre_instructions_for_FV.md, so it's a given for this session – but it's the number to challenge if you see spurious failures.

The \$error deliberately prints both %0d (the MAX bound) and %0t (the timestamp) so a failing run tells you immediately how long it waited and when – required format per your output rules for every ap_.

This output is formal-tool portable, requires no restructuring, and includes all mandatory documentation, coverage witnesses, and clocking constructs enforced by the seven rule families.

9. Configuration

Purpose: The configuration tells the framework *how* to generate — which decomposition strategy to apply, whether to enforce formal-verification strictness (DOC9, fairness, bounded operators), how verbose to be, and what latency bounds and naming conventions to use — so that every run is deterministic and reproducible rather than left to the model's discretion.

Currently Active Configuration

Parameter	Value	Source
FV MODE	YES	Pre_instructions_for_FV.md
DOC9	Required (follows FV=YES) — <i>traceability layer: a documentation header accompanying every assumption</i>	settingsCombined
DECOMP	ATOMIC — <i>keeps each ap_/mp_ focused on a single behavior</i>	Pre_instructions_for_FV.md
Stages	Decompose by protocol stage: <i>request → crossing → response → completion</i>	settingsCombined
MAX (latency bound)	16	Pre_instructions_for_FV.md
N (vector width)	2	Pre_instructions_for_FV.md
COV ITEMS	Antecedent reachability & full-sequence execution	Pre_instructions_for_FV.md
VERBOSITY	!C (code only) — <i>suppresses explanatory prose, tables, and analysis</i>	Pre_instructions_for_FV.md
GATE MODE	GATE_MODE_2 (abbreviated) — <i>pause points: CHECK 0 → Phase 1 → Phase 3 → Phase 7</i>	session selection
Prefixes	ap_ / mp_ / cp_ / p_	RAG_RULES (overrides file's a_/am_)
Bounded operators	Enforced	settingsCombined
Single clock / assertion	Yes	settingsCombined
disable iff	posedge, active-low !rst_n	settingsCombined

10. Conclusion: The Value Shift

The RAG-SVA framework (*Cohen–Chibani*) transforms a general-purpose LLM from an autocomplete engine into a rule-governed formal verification compiler. Its disciplined pipeline, quality gates, PORT-INTENT signal classification, and companion constraints eliminate the structural failure modes that unguided AI reliably introduces: vacuous proofs, over-constrained assumptions, and liveness misclassification. The result is a deterministic, auditable, and tool-portable SVA generation methodology ready for formal sign-off. Specifically:

Deterministic, auditable generation

- Same requirement + same rule corpus → functionally identical SVA regardless of which LLM executes the prompt; correctness is corpus-resident, not model-resident
- Every generated property traces back to a named rule and a source requirement.

Specification quality gates — catch bugs before code exists

- RQMO halts generation until the spec clears six structured ambiguity categories: signal-width, temporal bounds, operational modes, asynchronous gating, tie-breaker resolution, and contradiction checks.

- Any specification gap, contradiction, or out-of-scope construct triggers a mandatory clarification request before a single line of SVA is written.

Intention-boundary protection

- PORT-INTENT table classifies every signal by its true driving source, not just wire direction.
- Inputs → assume; outputs and tracked states → obligation.
- Prevents the most critical FV error: placing a DUT output inside an assume statement, which over-constrains the proof space and produces a false-PROVEN status while masking real hardware bugs

Formal-safe operator discipline

- Bans tool-crashing / vacuity-prone constructs: `until`, `within`, `firstmatch`, `##[1:$]` without `strong()`, `##[0:$]`, and `|=>`.
- Standardizes on overlapping implication (`|-> ##1`) and finite bounded ranges (`##[1:N]`)
- Converts legacy PSL/SVA patterns into modern formal-ready equivalents

Liveness handled correctly

- Converts unbounded “eventually” into bounded Form A (`##[1:MAX]`) for convergence, or Form B with mandatory `strong()` for true unbounded liveness.
- Prevents vacuous pass caused by antecedents that never fire enforced through mandatory UR-5 coverage witnesses and FV-FA fairness constraints

Mathematical-soundness infrastructure (companion constraints)

- FV8 fairness (`mp_fair_*`) — stops the tool from holding an input in a blocking state forever.
- FV9 anti-livelock (`mp_*_no_livelock`) — stops continuously toggling inputs from restarting an obligation so it never completes.
- Generated automatically by scanning the full PORT-INTENT table, not just signals named in the assertion.

Anti-vacuity and traceability sign-off

- UR-5 pairing: every assumption gets a `cp_*_hit` cover; an unreachable cover flags over-constraint or misclassification.
- Mandatory coverage for antecedent reachability and full-sequence execution on every assertion.
- DOC9 headers on every assumption (intent, type, justification, risk level) for human audit.
- Post-generation regex scans sweep for banned operators before tool execution.

Multi-directional translation

- **English ↔ SVA ↔ PSL ↔ RTL/VHDL**: ingests unconstrained intent, modernizes legacy checkers, extracts assertions from RTL port maps, and reverse-translates generated SVA into structured English requirements for human sign-off.

Workflow control and consistency

- 7-phase validation workflow with blocking gates (CHECK 0 → Phase 7) and code blackout on any failure.
- Mandatory `p_ / ap_ / mp_ / cp_` naming so every artifact is self-identifying.
- Standardized timing diagrams, signal tables, and requirement documents as deliverables.

CAVEAT

- **Human must be in the loop**
 - RAG-SVA guarantees rule-compliance and structural soundness.
 - It does not guarantee specification correctness.
 - A maximum-latency bound of sixteen cycles is only as tight as the latency contract it is meant to encode.

- The framework makes that engineering judgment explicit and traceable, but the judgment itself remains the engineer's responsibility.

ⁱ <https://systemverilog.us/> Includes

BOOKS:

- Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules 2026 (*with downloadable RAG-SVA rules*)
- Fast-Tracking SVA through Exposure: Core Usage, Concepts, AI Integration 2025
- SystemVerilog Assertions Handbook Revised 4th edition 2023
- SystemVerilog Assertions Handbook, 4th Edition in Chinese
- *See site for more books*

PAPERS:

- [OVL Arbiter, Claude with vs. without SVA-RAG for Formal Verification](#)
- [AI-Assisted FV of a SV Parking Lot Controller](#)
- [grnt -> req, Claude LLM vs Claude LLM + RAG](#)
- [Paper: It's the Requirements, STUPID!](#)
- [Post: Path for engr for effective solutions](#)
- [PAPER: Converting Legacy PSL-SIM to SVA-FV-Ready Code](#)
- [PAPER: Converting Legacy VHDL to SVA-FV-Ready Code + SPECS](#)
- [PAPER: Converting SV/SVA to VHDL/PSL](#)
- [PAPER: RAG-SVA Paradigm Shift in FV](#) (*this paper*)
- *See site for more papers*

ⁱⁱ From Cohen& Chibani book and RAG-SVA rules “**Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules**” 2026

ⁱⁱⁱ See Cohen’s papers:

“Understanding SVA Degeneracy “

“Understanding the within Operator”

“(intersect) vs (throughout, until, until_with, within)”