It's the Methodology, Stupid!

Meaningful SVA: Write for Formal, Run Anywhere

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

Ask ten engineers why an assertion suite failed to catch a bug, and nine will point at syntax: a wrong operator, a missed edge, an off-by-one delay. The tenth — the one worth listening to — will point at methodology. The uncomfortable truth of SystemVerilog Assertions is that the language is rich enough to express almost anything, including things that are vacuous, unreachable, degenerate, or quietly meaningless. Syntax tells you whether a property compiles. Methodology tells you whether it means anything. This paper argues a simple **thesis**: **write every assertion as if a formal verification (FV) tool will read it, even if it will only ever run in simulation**. The discipline that FV forces — *separating environment from obligation, proving reachability, guarding against vacuity and lock conditions* — is precisely the discipline that makes assertions meaningful anywhere. The examples below are drawn from a working rule corpus (*the RAG-SVA framework*) that encodes this methodology as machine-checkable gates.

1. Sanity and Consistency Before a Single Line of Code

A methodology begins before code. The first gate is the requirement itself: if the specification is ambiguous — an unnamed signal, timing language that could mean a hold or a window, a cycle count that could be inclusive or exclusive, two implications tangled in one sentence — the correct output is a question, not a best-guess assertion. A halt-don't-guess discipline (*rule RQM0 in the corpus, with six named ambiguity categories*) costs one round-trip with the specifier and saves a property that silently checks the wrong thing.

The second gate is linguistic consistency. English timing words map to SVA operators in ways that are easy to get backwards. "Holds for N cycles" is a repetition, signal[*N]; "within N cycles" is a window, ##[1:N] signal. Swapping them produces properties that compile cleanly and verify the wrong behavior. The corpus enforces this as a keyword-driven check (CHECK 4): scan the requirement for hold/window vocabulary and reject any mismatch between the words and the operator.

The third gate is an operator discipline. Certain legal SVA constructs are banned outright because their failure modes outweigh their convenience: unbounded `##[0:$]` (*no formal convergence, no simulation endpoint*), `until` and `first_match` (*ambiguous or deprecated endpoints*), `within` (hides the timing relationship), and `|=>` (*a hidden cycle that reviewers routinely miss — replaced by the explicit* `|-> ##1`). Bounded ranges, the goto operator `[->1]`, and `intersect` are preferred because their semantics are exact and their state spaces finite. Two subtler consistency traps get their own checks: the hold-intersect offset (when an assertion window ##[A:B] is paired with an assumption hold, the hold bounds must be [A+1:B+1], because overlapping implication consumes the trigger cycle), and guard/done degeneracy (a liveness `form strong(GUARD[*0:MAX] ##0 DONE`) that is illegal and irrelevant). Finally, naming is not cosmetics: `p_` for properties, `ap_/mp_/cp_` for `assert/assume/cover` statements makes every directive's role visible in a log file at three in the morning.

2. Separate the Environment from the Obligations

The single most consequential methodological decision is classifying every signal by intention before writing anything. An input driven by the environment is an assumption signal: it may appear in antecedents and in assume properties, because the testbench or the neighboring block — not the DUT — is responsible for its

behavior. An output driven by the DUT is an obligation signal: it belongs in assertion consequents, because the DUT must deliver it. A port-intent table listing every signal's direction, intention, and activity is produced and reviewed before any property, and it becomes the contract against which every property is mechanically checked: an input token in an assertion consequent is a halt, not a warning.

This boundary does real work. Consider the classic handshake: "every assertion of **req** must be followed by **ack** within 16 cycles; req holds until and with the occurrence of **ack**." The naive rendering mixes both duties into one assertion. The methodology splits it: the hold behavior of **req** is an environment promise, so it becomes an assumption; the delivery of **ack** is the DUT's obligation, so it becomes an assertion:

```
// Environment: req holds through-and-including the ack cycle
mp_req_holds_until_ack: assume property (@(posedge clk) disable iff (!rst_n)
  (req && !ack) |-> ##1 req);   // ack: release trigger only

// Obligation: the DUT must acknowledge within the bound
ap_req_to_ack: assert property (@(posedge clk) disable iff (!rst_n)
  $rose(req) |-> strong(##[1:16] ack));
```

Note the disciplined exception: **ack**— an obligation signal — appears in the assumption's antecedent. That is permitted only because it acts purely as a release trigger, gating when the hold ends, and is never constrained by the assumption. The corpus names this exception (F1-A), requires explicit sign-off, and requires the inline comment you see above. Everything else about the boundary is absolute: constraining a DUT output in an assumption masks bugs; obligating an environment input in an assertion blames the DUT for the testbench's behavior.

3. Anchor Each Obligation to Its Own Signal

Some obligations are not about responding to the environment at all — they are self-consistency requirements on an output. "**ack** is a one-cycle pulse" is a property of **ack** alone. The methodology anchors such checks on the obligation signal's own edge:

```
// Pure edge trigger on an obligation signal (documented exception)
ap_ack_pulse: assert property (@(posedge clk) disable iff (!rst_n)
  $rose(ack) |-> ##1 !ack);
```

At first glance this violates the boundary of Section 1 — an obligation signal in an antecedent. The corpus treats it as a second named exception (the pure-edge-trigger case) with its own confirmation gate, because the antecedent here is not an environment stimulus; it is the obligation's own occurrence, used to anchor the check of its shape. The distinction matters: anchoring on **$rose(ack)** checks every pulse the DUT actually produces, exactly when it produces one, with no dependence on how the environment behaved to provoke it. The obligation is checked at its source.

4. The Environment Becomes Clear — and Meaningful

A side effect of the separation is that the environment stops being folklore. Every assumption is an executable, reviewable statement of what the surrounding world promises, and the methodology requires each one to carry a documentation block stating its intent, its type (*input constraint, fairness, state reduction, temporal*), its justification traced to the specification or a physical reason, and its risk level, with high-risk assumptions flagged for human review. The assumption set becomes the interface specification — readable by the block's next integrator, checkable by the formal tool, and monitorable in simulation. When a design moves to a new

testbench, the assumptions travel with it and immediately report any environment that breaks the contract. An undocumented constraint buried in a testbench class is a rumor; an **mp_** property with a justification block is a specification.

5. Reachability, Lock Checks, and Vacuity

A property that passes tells you nothing unless you also know it could have failed. The methodology attacks the three classic silent-pass modes directly. Reachability: every assumption must be exercisable, so each **mp_** carries a paired cover (**cp_*_hit**) proving its antecedent condition actually occurs; an unreachable assumption either over-constrains the proof or constrains nothing. Structural rules reinforce this — assumptions use implications rather than standalone sequences (a standalone sequence in an assume must hold every cycle, collapsing the state space to unreachability).

Vacuity: every assertion carries two covers — one proving the antecedent fires (the trigger is reachable) and one proving the full sequence completes end-to-end (the *req-to-ack* path is genuinely exercised). A liveness assertion also needs generation-time vacuity defense: if the tool may simply hold **req** low forever, the assertion passes without ever being tested, so a fairness assumption (**s_eventually req**) forces activation.

Lock checks: deadlock and livelock are where hold-style protocols hide their worst behavior. With **req** held until **ack**, a formal tool can launch new transactions forever so that no individual transaction completes — a livelock trace that is legal under every safety property. The counter-measure is an anchor signal that isolates genuinely new activity from legal holding, and a strong-form convergence assumption over it:

```
logic new_req; // Automatically generated by RAG
assign new_req = $rose(req) || (req && $past(ack, 1));

mp_livelock_req: assume property (@(posedge clk) disable iff (!rst_n)
  s_eventually always (!new_req));   // eventually, no new transactions
cp_new_req_hit: cover property (@(posedge clk) disable iff (!rst_n) new_req);
```

The diagnostic cover on the anchor closes the loop: if **new_req** is unreachable, the anchor itself is wrong. Finally, liveness obligations use **strong()** with bounded windows — **strong(##[1:MAX] ack**) — so that a trace that runs out of time fails rather than passing by exhaustion. Bounded strong liveness converges in formal tools and fails honestly at end-of-test in simulation; unbounded liveness without **strong()** is the canonical vacuous pass.

6. Simulation-Only? Write FV-Style Anyway

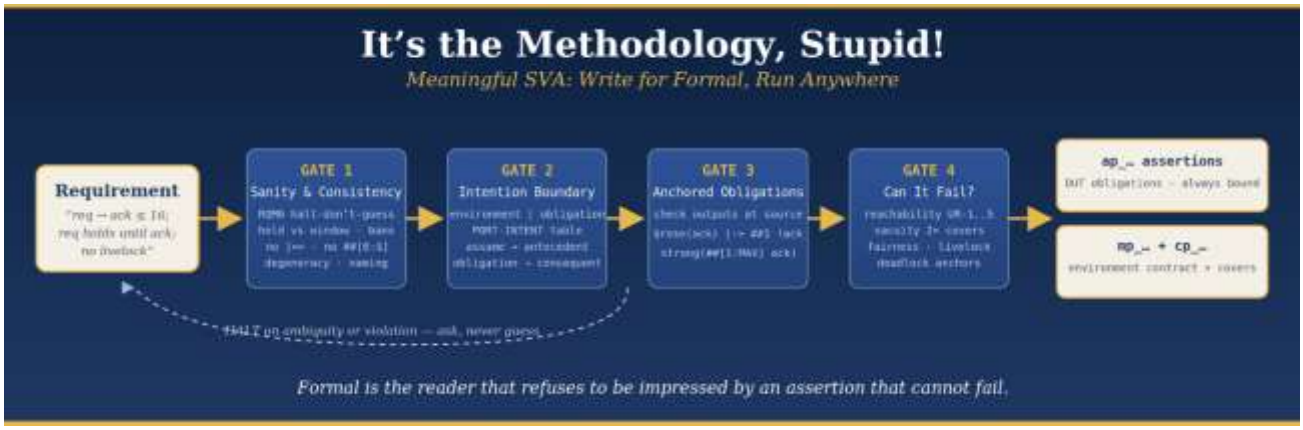
Suppose formal is not in the flow at all. The FV-style suite is still the better simulation suite. The assumptions do not disappear in simulation — simulators treat **assume** essentially as **assert** — so the environment contract from Section 4 becomes a live checker of the testbench itself, catching stimulus bugs that would otherwise masquerade as design failures. The covers from Section 5 report vacuity and reachability in every regression, answering the question no passing assertion answers on its own: did this check ever actually run? And the bounded, explicit-delay, banned-operator style produces properties whose simulation cost is predictable, with no unbounded threads accumulating across a long run.

If simulation speed is a genuine concern, the methodology offers a clean structural answer rather than a stylistic compromise: generate two separate checkers, bound independently. One module carries the assertions only — the obligations, the part you always run. The second carries the assumptions and coverage

— the environment contract and the reachability/vacuity instrumentation. Both bind to the same DUT instance with identical port maps. Nightly regressions bind both; a speed-critical soak run binds only the assertion checker; a testbench bring-up session might bind only the environment checker. Because the split follows the intention boundary of Section 1, it falls out mechanically — an AI generator working under the rule corpus can emit both files from the same validated property set in a single pass. The separation of environment from obligation, adopted for verification rigor, turns out to be exactly the modularity that performance tuning wants.

Closing

None of the individual rules above is exotic. What makes them a methodology is that they are enforced as gates, in order, before code exists: settings and requirement sanity first, intention boundaries second, syntax discipline third, and reachability, vacuity, and lock analysis as first-class deliverables alongside the assertions themselves. **Write for formal, even when you run in simulation** — not because formal is the destination, but because formal is the reader that refuses to be impressed by an assertion that cannot fail. **It's not the syntax. It's the methodology.**



References and Companion Work

This paper is a companion piece to the RAG-SVA framework developed in full in the authors' forthcoming book [1]: the complete rule corpus, the intention-boundary and property-type classification system, the phase-gated generation workflow with halt-don't-guess discipline, and the formal-verification companion analyses (fairness, livelock, vacuity) illustrated here. The rule identifiers cited in this paper (RQM0, CHECK 4, F1-A, UR-1 through UR-5, LNG-HIO, LNG37b) refer to that corpus. Assertion style, operator semantics, and AI-assisted usage background can be found in [2] and [3]; IEEE Std 1800 defines the strong/weak sequence semantics on which Section 4 relies.

[1] B. Cohen and K. Chibani, Knowledge-Driven AI for SystemVerilog Assertions: A Verification Framework of Methodologies, Processes, and Coding Rules, VhdlCohen Publishing, 2026.

[2] B. Cohen, Fast-Tracking SVA through Exposure: Core Usage, Concepts, AI Integration, VhdlCohen Publishing, 2025.

[3] B. Cohen, SystemVerilog Assertions Handbook, Revised 4th Edition, VhdlCohen Publishing, 2023.