How to Use the "not" Operator for Formal Verification

A Case Study in Ambiguous English, Legal SVA, and the Halt That Catches Both

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

---

The **`not()`** operator is one of the smallest pieces of syntax in SystemVerilog Assertions, and one of the most consistently misused. It looks like ordinary logical negation, and in casual conversation about a requirement it gets talked about as if it were — "*not X happens*," "*it is not the case that Y*." But **`not()`** is not a boolean operator. It is a property operator, with a specific grammatical role, and exactly one place in an implication where it is legal to appear. This paper is structured as a case study: a single English requirement, plausibly written, translated three different ways — one broken, one legal but structurally opaque, one correct — with each translation illustrating a different real-world failure mode. Along the way it shows how a structured rule corpus — the RAG-SVA methodology behind *Cohen & Chibani's Knowledge-Driven AI for SystemVerilog Assertions* [1] — catches the failure modes automatically, halts generation until a human resolves the ambiguity, and applies the same discipline at partition scale and as a static linter over legacy code.

1. What not() Actually Is

In the SystemVerilog Assertion grammar, **`not()`** is defined as a property operator: **`property_expr ::= not property_expr | ...`**. Given any argument, **`not()`** always produces a property, never a sequence. The **`|->`** operator has an asymmetric signature: its antecedent must be a `sequence_expr`; its consequent may be a full `property_expr`. So **`not()`** is legal in the consequent — because the consequent accepts properties — and illegal in the antecedent, because the antecedent only accepts sequences, and `not()` never produces one.

Evaluation adds a second trap. A concurrent assertion is checked at every clock tick. **`not(seq)`** at a given cycle means "the sequence does not begin a match starting here" — useful inside a consequent, or as a standalone property. But wrapping an entire implication in **`not()`** invokes propositional logic rather than sequence semantics: `A |-> B` is logically equivalent to `!A OR B`, so **`not(A |-> B)`** reduces to **`A and not(B)`**. Asserted at every cycle, that demands `A` be true on every cycle, forever — almost never the intended meaning.

2. Implies Operators, Briefly

SVA has two "implies" operators.

- `antecedent |-> consequent` is overlapped implication: the antecedent is a sequence, the consequent is a property, and the consequent begins evaluation on the same cycle the antecedent's match completes. Because **`not()`** produces a property, the following is legal: `sequence_1 |-> not(sequence_2)`.
- `prop1 implies prop2` is property-level implication: `prop1` is the antecedent, `prop2` is the consequent, both are properties, and both begin evaluation at the same cycle — neither waits for the other. Because both operands are properties, either side may be a **`not()`**-wrapped sequence: `not(sequence_1) implies not(sequence_2)` is legal.

For a plain boolean antecedent like `$rose(req)` the two operators behave identically; for a multi-cycle sequence antecedent they diverge, because `|->` starts the consequent at the antecedent's match completion while `implies` starts both operands simultaneously.

3. Case Study: One English Requirement, Three Translations

Here is the requirement, as it would plausibly land on a verification engineer's desk. It is not a strawman — it is exactly the phrasing that gets written down when a specifier means to negate an expected behavior, and it is exactly the phrasing that most easily lures a bad translation.

> *"It is not the case that: whenever req rises, ack pulses (rises for exactly one cycle) within 1 to 3 cycles."*

The interpretation of this requirement is genuinely ambiguous. The same English can be read at least three different ways:

- No "new-req-then-ack-within-1-to-3" under any circumstances
- If a new req arrives, then no ack pulse in cycles 1 to 3 that follow.
- If there is no ack pulse in the next 1 to 3 cycles, then no new req shall be issued in that window.

These readings do not all describe the same behavior. The first two share a common intent — whenever req rises, an ack pulse must not occur in the following 1 to 3 cycle window — and it is that intent the case study proceeds with. The point of naming the alternatives explicitly is that a good translator has to notice them, and the user has to understand which interpretation the generated SVA actually encodes. Three translators, faced with this English, produce three different SVA properties. Only two are usable, and only one is the preferred form.

Translator 1 — the literal syntactic mapper (produces Form 1, broken)

A translator that maps English phrase-by-phrase into SVA sees "it is not the case that A implies B" and writes it as literal negation of an implication:

```
not( $rose(req) |-> (##[1:3] $rose(ack) ##1 !ack) );
```

By the reduction in Section 1, this collapses to `$rose(req) and not(##[1:3] $rose(ack) ##1 !ack)` at every cycle. Asserted at every clock tick, this demands that `req` be rising on every single cycle, forever — *impossible for a rising-edge event* — and it fails on the first cycle `req` is not rising, regardless of what `ack` ever does.

There is a second, subtler problem. When the antecedent is false (i.e., `$rose(req) == 0`), the wrapped property is vacuously true, and negating it produces a hard false. IEEE 1800 does not define a symmetric notion of "false vacuity" — *vacuous truth is a well-defined semantic escape hatch, but vacuous falsehood is not* — so most tools interpret the negated case as a real failure rather than a benign non-firing. The property is not just wrong; it is unsatisfiable per cycle on essentially every trace. And it compiles cleanly — which is the danger. A code review that reads the English and reads the SVA in isolation would not necessarily notice; both look plausible on their own.

Translator 2 — the sequence-level rewriter (produces Form 2, legal but opaque)

A translator that recognizes the trap in Form 1 but does not enforce an explicit trigger/obligation split reads the English differently — as a forbidden pattern statement — and writes it as a negation of the full sequence:

```
not( $rose(req) ##[1:3] $rose(ack) ##1 !ack );
```

This is legal. The whole argument to `not()` is a genuine sequence (req-rise, then within 1 to 3 cycles ack-rise, then next cycle ack-fall), and `not(seq)` at every cycle means "this sequence does not begin matching here." Because `$rose(req)` is a plain boolean, at cycles where `req` is not rising the sequence trivially fails to match,

so the `not()` is trivially satisfied — no per-cycle unsatisfiability. Cycle-for-cycle, this is behaviorally equivalent to Form 3 below. But the SVA has erased something the English requirement had: the distinction between the trigger (`req`, an input the environment drives) and the obligation (`ack`, an output the design drives). Both are folded into one undifferentiated sequence with no `|->` between them. Correct answer, wrong shape — an auditability failure, not a logic bug.

Translator 3 — the rule-guided translator (produces Form 3, correct)

A translator gated on the RAG rules reads the English, halts before writing, and confirms the intended reading — a forbidden pattern following a trigger — with the human. Then it writes:

```
$rose(req) |-> not (##[1:3] $rose(ack) ##1 !ack);
```

The antecedent is a plain sequence (`$rose(req)`). The consequent is a property (`not(...)`), which is exactly where `not()` belongs — the grammar rules of Section 1 are satisfied. The trigger and the obligation appear on opposite sides of `|->`, so the auditability rule is satisfied. The result is unambiguous, both to a compiler and to a reviewer scanning the property text against a port-intent table: whenever `req` rises, the forbidden ack-pulse pattern must not occur in the following window.

Translator	Produces	Compiles?	Correct logic?	Failure class
1 literal syntactic mapper	`not(A |-> B)`	Yes	No	Wrong logic (unsatisfiable per cycle)
2 sequence-level rewriter	`not(A ##[1:3] B ##1 C)`	Yes	Yes	Structurally opaque (no trigger/obligation split)
3 rule-guided translator	`A |-> not(##[1:3] B ##1 C)`	Yes	Yes	— (correct)

4. How the RAG Rules Help Produce Good Assertions

The case study above is not a story about one lucky halt — it is a story about several specific mechanisms in the corpus, each catching a different failure mode.

The not() |-> placement rule

The corpus encodes the placement rule directly: `!` or `~` applied to a sequence must be rewritten as `not()`; `not()` before `|->` is a halt condition; `##N` or `##[n:m]` adjacent to `not()` must be rewritten with an explicit continuation (i.e., `sequence_1 #_# prop_1`). This catches the specific placement mistake in Form 1 before it is written.

Intention boundaries

Separately from grammar, the corpus requires every signal to be classified assume (environment) or obligation (design) before any property is written, and requires the antecedent/consequent split to reflect that classification. This catches Form 2 — the legal-but-opaque form — as an audit failure: not because the SVA does the wrong thing, but because the SVA hides the trigger/obligation distinction the requirement actually stated.

Liveness canonical forms

Where `not()` interacts with bounded and unbounded ranges, the corpus's canonical liveness forms (bounded form with `strong()`, unbounded form with mandatory `strong()`) prevent a second class of `not()`-adjacent mistake: a `not()`'d sequence containing an unbounded `##[1:$]` that passes vacuously because nothing forces the tool to explore a long enough trace to falsify it.

5. The Halt: Stopping Until a Human Confirms

A rule corpus that only caught the grammar violation would be a linter. What distinguishes the RAG-SVA methodology in this case study is that the underlying English — "it is not the case that, whenever req rises, ack pulses within 1 to 3 cycles" — is genuinely ambiguous. It admits at least three distinct SVA translations: a forbidden pattern (the intended reading, and Form 3), a documentation note that no such timing relationship is being claimed (which would generate no assert at all, only a cover), and a conditional exception with an unnamed exception signal (a template with a placeholder, generated only after the exception is specified). Each produces different, entirely legal SVA.

The corpus's response is not to guess the most likely reading — it is to halt, present the candidate readings explicitly, and wait for the human to pick one before any `not()`-bearing property is generated. That is why Translator 3 in the case study did not simply write Form 3 on its own — it wrote it after a halt that named the alternatives out loud and got a confirmation.

6. Scaling Beyond One Line: Partitions, Not Just Properties

The case study used a single requirement. In practice the same discipline runs across a partition's worth of requirements — dozens or hundreds of properties covering a bus interface, an arbiter, or a full IP block — rather than a check invoked by hand each time:

- Check scans — the not()-placement pattern is a mechanical scan applied to every property in a file, not a per-line manual re-check.
- Requirement-split two-stage workflow — when a partition's worth of requirements arrives together, generation is split into a Stage 1 analysis-only pass (classification, ambiguity flags, not()-placement review) confirmed by the human before any Stage 2 code is written for the whole batch.
- Consistency across the corpus — fifty properties in one partition get the same operator preferences, the same naming, and the same not() placement rule enforced identically on property one and property fifty. A human reviewer's attention degrades over a long file; a scan does not.

7. The Rules as a Linter, Not Only a Generator

Every section above has been framed as generation-time discipline — catching mistakes before code is written. The same rule set runs just as well in the other direction: as a static linter over SVA that already exists, written by a human, by a different AI, or inherited from a legacy codebase with no rule corpus behind it at all. The pattern-match for `not()` misplacement does not care whether the code was just generated or was checked in five years ago — it is a scan, and scans apply equally to new and old code. This is the same operation described in the legacy-conversion literature [2, 3, 4] under a different name.

Closing

The requirement in the case study — "it is not the case that whenever req rises, ack pulses within 1 to 3 cycles" — sounds like an ordinary specification sentence. Its three translations, one broken and two legal, illustrate the whole argument of the RAG-SVA methodology: what compiles is not the same as what checks the right thing; what checks the right thing is not the same as what is auditable; and both problems can be caught, at

generation time or as a linter over existing code, by rules that halt when the input is ambiguous rather than proceeding on a plausible guess. `not()` is a small operator with an exact rule. What makes it worth a paper of its own is how often the wrong placements look right.

References

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

[2] *Converting Legacy PSL-SIM to SVA-FV-Ready Code*

[3] *Converting Legacy VHDL to SVA-FV-Ready Code + Specs*

[4] *Converting SV/SVA to VHDL/PSL*

[5] *Gen-SVA: Source-Agnostic Generation of Formal-Ready SVA*

[6] *It's the Methodology, Stupid! — Meaningful SVA: Write for Formal, Run Anywhere*