Determinism in SystemVerilog Assertions for Formal Verification

Ben Cohen, Kais Chibani
Cohen Publishing — systemverilog.us

1. What Determinism Means for a Property

In formal verification, a property is `deterministic` when, for a given design state and a given set of inputs, there is exactly one legal outcome the checker evaluates — no branching choice in how the pass/fail result is reached. A `nondeterministic` model, by contrast, allows more than one legal next state or outcome for the same inputs; formal tools can model and explore that kind of choice deliberately, using `rand` (free) variables to represent unconstrained environment behavior. What determinism rules out is different: a single property whose own antecedent or consequent silently embeds more than one independent triggering path or more than one independent required outcome, joined with `or`.

2. Why It Matters Specifically for Formal Tools

A simulator only ever walks one path through a compound, or-joined property, and the surrounding waveform usually lets an engineer infer which path fired. A formal engine has no such context: it explores the full state space symbolically, so an or-joined antecedent or consequent forces the tool to fork the proof obligation across every combination of the intermixed paths. Two costs follow. First, the state space the tool must search grows with every extra path folded into one property, which is exactly the kind of blow-up formal engines are most sensitive to. Second, and worse for debugging, a counterexample on a compound property does not say which of the intermixed protocol stages actually failed — the human still has to reverse-engineer that from the trace. Determinism is therefore a proof-quality requirement for FV, not a style preference for readability.

3. Ranges Are Not the Same as Nondeterminism

Timing ranges (`##[1:5]`) and repetition ranges (`[*2:4]`) look flexible, but they are not nondeterministic in the strict sense: SystemVerilog's assertion scheduling semantics are themselves deterministic, so at every clock the checker enumerates exactly the same well-defined set of candidate matches, every run, per the IEEE 1800 scheduling algorithm. What actually breaks determinism is mixing independent triggering conditions or independent outcomes with `or`, or leaving a window unbounded (`##[0:$]`, `[*1:$]`). Both explode the state space and make a first-match-style pick ambiguous, which is why a formal-facing rule set also bans constructs like `first_match(...)` and unbounded delay/repeat windows alongside the determinism check itself — they are three symptoms of the same underlying problem.

4. A Small Example

Consider a property meant to cover two unrelated protocol stages — a fetch sequence and a DMA sequence — written as one compound assertion:

```
// NONDETERMINISTIC — two paths folded into one property
($rose(fetch) ##[1:5] mem_rd) or ($rose(dma) ##[1:3] tri_bus)
  |-> ##1 (MODE == GET_FETCH_TAG) or (##2 MODE == GET_MEM_TAG);
```

This single property has two different triggering sequences in its antecedent and two alternative, differently-timed outcomes in its consequent. A formal engine must consider all four combinations of trigger and outcome as one proof obligation, and a failure gives no indication of which protocol stage was actually involved. The fix is not to add more logic — it is to recognize that this was always two separate claims, and to give each its own name:

```
// DETERMINISTIC — one triggering path, one outcome, per property
ap_mem_fetch: assert property (@(posedge clk) disable iff (!rstn)
  ($rose(fetch) ##[1:5] mem_rd) |-> ##1 (MODE == GET_FETCH_TAG)
);

ap_dma_bus: assert property (@(posedge clk) disable iff (!rstn)
  ($rose(dma) ##[1:3] tri_bus) |-> ##1 (MODE == GET_MEM_TAG)
);
```

Nothing is lost in the split: both protocol stages are still fully covered. What changes is that each property now has exactly one triggering path and one required outcome, so a formal engine proves (or refutes) each independently, and any counterexample points, by the assertion's own name, at exactly which claim broke.

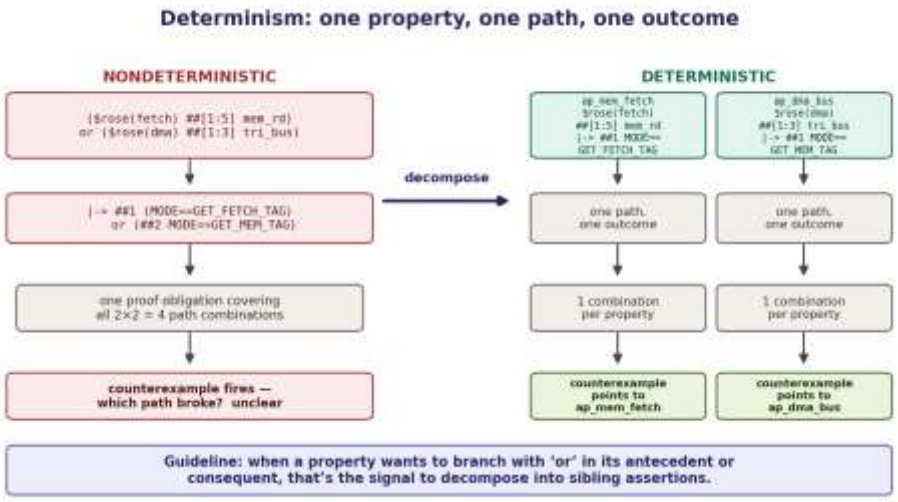

Figure 1. A nondeterministic, or-joined property forces the formal engine to prove all path combinations at once and yields an ambiguous counterexample; decomposing into ap_mem_fetch and ap_dma_bus gives each path its own, independently traceable proof.

5. Guideline

One property, one triggering path, one outcome. When a property wants to branch on `or` in its antecedent or its consequent, that is the signal to decompose into sibling assertions rather than encode the branch inside a single property. This keeps the state space bounded, keeps every formal counterexample traceable to one nameable claim, and — in an AI-assisted or RAG-SVA generation flow — keeps the same rule corpus producing the same deterministic result regardless of which engine or model authored the assertion.

References

Ben Cohen, RAG-SVA rule corpus and FV tuning notes: “Determinism,” “Ranges and Determinism in Assertions,” and “FV-CHECK 16.1: Determinism Validation.”

Cohen, B., Chibani, K. “RAG-SVA: A Paradigm Shift in Formal Verification” — deterministic, auditable generation: same requirement plus same rule corpus yields functionally identical SVA regardless of which model executes the prompt.

systemverilog.us