When Assumptions Contradict: What Formal Verification Does and What It Hides

With Back-Translated Requirements (Line Items)

Ben Cohen, Dr. Kais Chibani

systemverilog.us

November 2026

Abstract

In formal verification (FV), assumptions define the legal environment, and every proof result is only as valid as those assumptions. This paper uses a simple request/acknowledge handshake to show what happens when two assumptions demand opposite values of the same signal in the same cycle. Formal does not choose between them; it removes every trace that reaches the conflict. When the signal that triggers the conflict is a DUT output, the input assumptions end up constraining the design itself, slow-response paths are never explored, and the proof report can read "all proven" while real behavior goes unchecked. We show how to recognize the problem, how to detect it with covers and tool checks, and how to fix it by classifying each requirement correctly as an assumption or an obligation.

1. Assumptions define the legal traces

A formal tool explores every input sequence that satisfies all assumptions (`assume property`) [2], [4]. An assertion fails only if some legal trace violates it, and a cover is reached only if some legal trace satisfies it. Assumptions are therefore not checked against the design; they are taken as true and used to prune the search.

This has a direct consequence: if the assumptions together admit no legal trace, or admit none beyond a certain cycle, formal has nothing to check past that point. Every assertion then passes vacuously and every cover becomes unreachable. Even `assert property (0)` would be "proven" in a fully over-constrained environment.

2. The example requirements

Consider a requester driving `req` and a responder driving `ack`, with two English requirements:

- **R1:** `req` lasts at most 2 cycles; by the third cycle it must drop.
- **R2:** `req` must hold until `ack`, and then drop.

Written directly as assumptions on `req`:

```
// R1: req lasts at most 2 cycles
am_req_max:  assume property (@(posedge clk) $rose(req) |-> ##[1:2] !req);
// R2: req holds until ack
am_req_hold: assume property (@(posedge clk) req && !ack |-> ##1 req);
```

Each assumption is reasonable on its own. Together they are consistent only if ack arrives in cycle 1 or 2. If ack arrives in cycle 3, R1 requires req = 0 in cycle 3 while R2 (since ack was still 0 in cycle 2) requires req = 1 in cycle 3. There is no legal value.

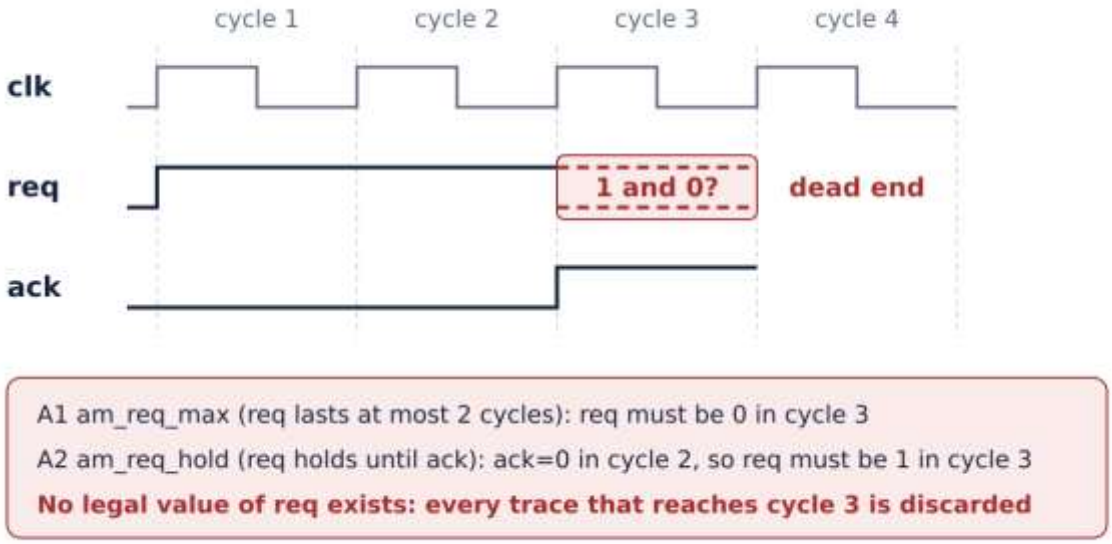


Figure 1. ack in cycle 3 leaves no legal value of req; the trace dead-ends.

3. What formal does with the contradiction

Formal does not resolve the contradiction or report which assumption is "right." Assumptions are constraints, so a state from which no legal next value exists is a **dead end**: any trace that reaches it is discarded. Prefixes before the dead end remain legal, so failures in cycles 1 and 2 can still be found, but anything that would fail in cycle 3 or later is hidden.

How damaging this is depends on who drives ack.

3.1 ack is an environment input

If ack is also free, the solver simply never drives ack in cycle 3 or later. The conflict is avoided silently: no error, no warning in many flows, and the late-ack corner is never explored. The proofs look clean.

3.2 ack is a DUT output

In the realistic case, ack is produced by the design, and the environment cannot avoid the conflict. The result is that the input assumptions end up constraining the DUT's own behavior:

- **Fixed latency of 3 cycles:** every trace in which req rises dies in cycle 3. Formal effectively explores only runs where req never rises (or rises too close to the proof bound to reach cycle 3). All handshake assertions pass vacuously.
- **Variable latency (1, 2 or 3 cycles depending on internal state):** formal keeps only the runs where the DUT happens to acknowledge in cycles 1 or 2. Any bug on the slow-acknowledge path is never seen.

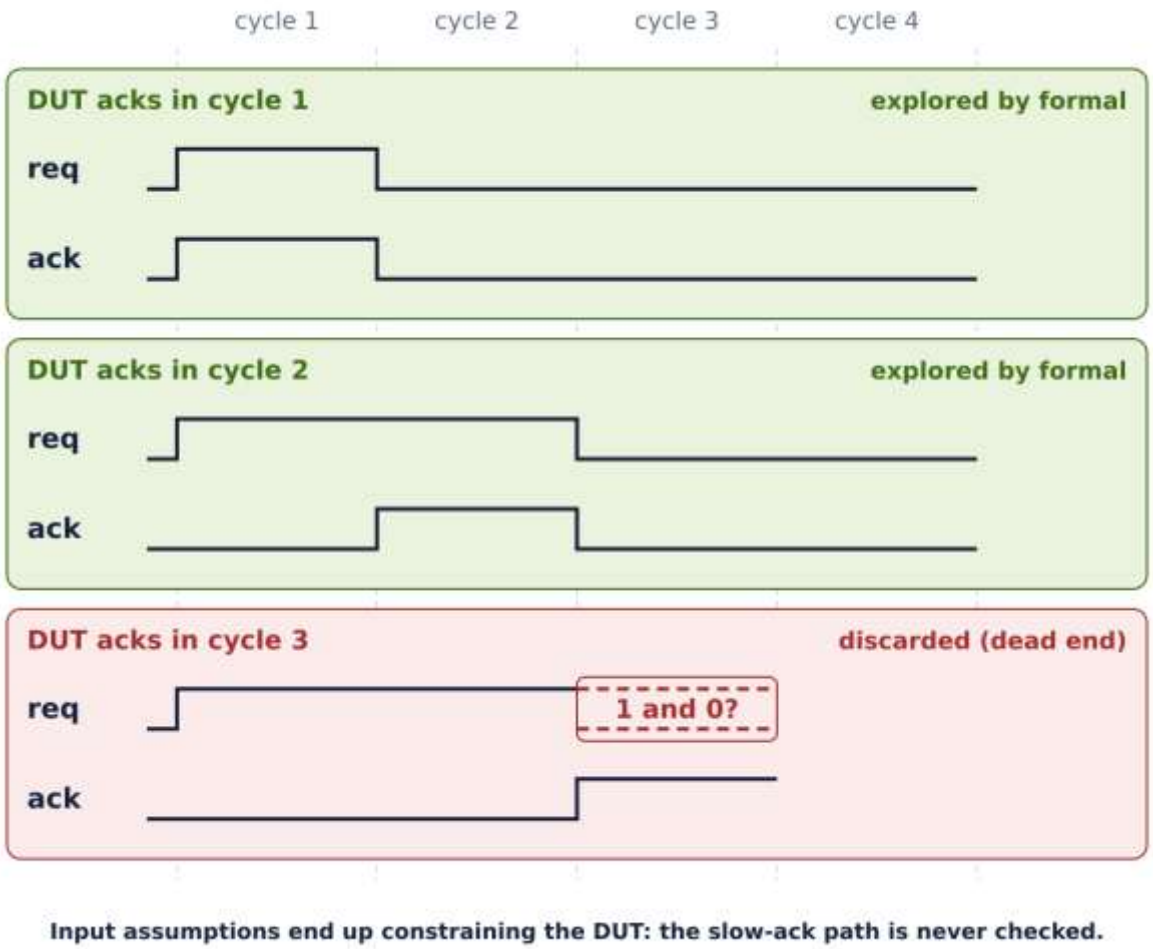


Figure 2. With ack as a DUT output, formal explores early-ack runs and discards ack-in-cycle-3 runs.

This is the dangerous form of over-constraint [2]: it is not a mistake in the RTL but in the verification environment, and it hides real design behavior behind “proven” results.

4. Detecting the problem

- **Tool checks:** formal tools provide checks for conflicting assumptions, over-constraint and dead-end states. Run them on every new set of assumptions.
- **Sanity covers:** write covers for behavior that must be reachable. Here, the late-acknowledge case:

```
cp_late_ack: cover property (@(posedge clk) $rose(req) ##2 ack);  // ack in cycle 3
```

If `cp_late_ack` is unreachable while the design can legitimately produce a 3-cycle acknowledge, the assumptions are too tight.

- **Witness traces:** ask the tool for a trace in which each assertion’s antecedent is true. An assertion with no witness passed vacuously.

- **Requirements review:** translate the assumptions back into plain-English line items and read them together [1]. The R1/R2 conflict is visible on paper before any tool runs. Section 7 applies this back-translation to the corrected SVA as line items L1–L5.

5. The fix: classify each requirement correctly

Because `ack` is a DUT output, "the request completes within 2 cycles" is really a requirement on the responder's timing. It cannot be assumed by the environment; it must be **asserted** as a DUT obligation. Only the rule about how the requester drives `req` stays an assumption:

```
// Environment (requester) rule: assumption
am_req_hold: assume property (@(posedge clk) req && !ack |-> ##1 req);
// DUT (responder) obligation: ack while req is in its 2-cycle window
ap_ack_2cyc: assert property (@(posedge clk) $rose(req) |-> ##[0:1] ack);
// Sanity cover: make sure late acks are still explorable
cp_late_ack: cover  property (@(posedge clk) $rose(req) ##2 ack);
```

Now a design that acknowledges in cycle 3 produces a real counterexample instead of a silent dead end. The counterexample forces a decision: either the DUT is too slow, or the 2-cycle requirement is wrong and must be revised (for example, "`req` may exceed 2 cycles only while waiting for `ack`").

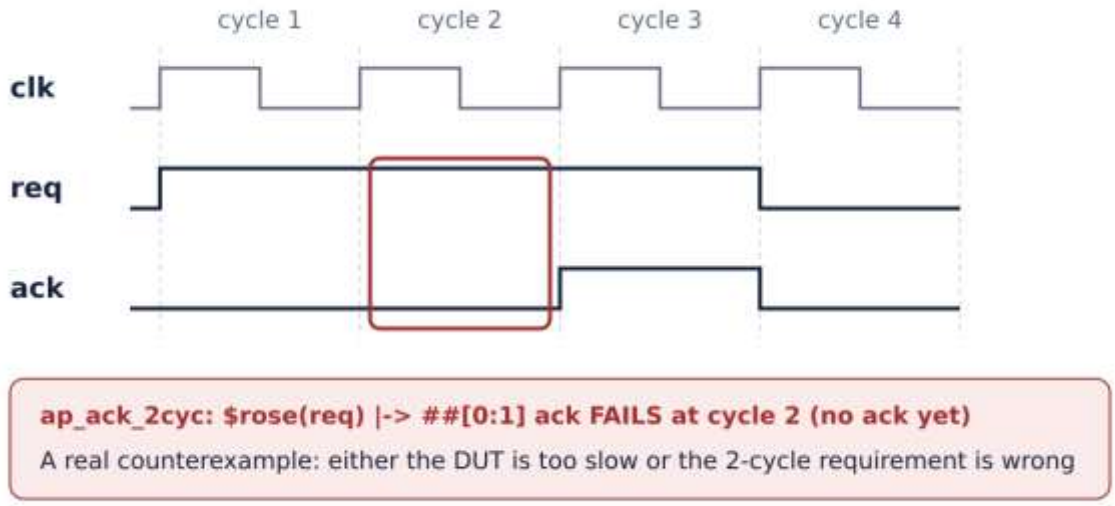


Figure 3. With the timing requirement as an assertion, formal reports a counterexample at cycle 2.

6. Methodology lessons

- A proof is only as good as its assumptions; "all proven" is meaningless without evidence that the assumptions leave the interesting behavior reachable.
- Separate assumptions (environment rules on DUT inputs) from obligations (DUT behavior, including output timing) before writing any SVA. A constraint whose subject is a DUT output belongs in an assertion.
- Check that assumptions are mutually consistent, ideally at the requirements stage, by reading them together as line items.

- Pair every assumption set with covers for the corner cases it must not exclude, and run the tool's over-constraint and dead-end checks.

These steps are part of the RAG-SVA flow [1]: requirements are clarified and classified as assumptions or obligations before SVA is generated, and the generated properties can be back-translated into line items for review. Contradictory or overly tight assumptions are then caught where they are cheapest to fix, rather than hidden behind proof results.

7. Back-translated requirements (line items)

Back-translating the corrected SVA (with `ack` as a DUT output) into plain-English line items is the review step that exposes what the SVA does and does not say.

Table 1. Assume/obligation table for the corrected handshake.

Signal	Driven by	Direction	INTENTION	Properties
`clk`	environment	DUT input	clock	sampling clock
`rstn`	environment	DUT input	reset	`disable iff (!rstn)`
`req`	environment	DUT input	ASSUME	`am_req_hold`, `am_req_drop`
`ack`	DUT	DUT output	OBLIGATION	`ap_ack_2cyc`, `ap_ack_with_req`

Signal roles (Table 1): `req` is a DUT input driven by the environment (assumptions); `ack` is a DUT output (obligations). Clock `clk`, active-low reset `rstn`; every property uses `disable iff (!rstn)`. Per the RAG-SVA rules, `|=>` is not used; next-cycle implication is written `|-> ##1` [1].

L1 [ASSUME] req holds until ack. If `req` is high and `ack` is low in a cycle, `req` is high in the next cycle.

```
property p_req_hold; req && !ack |-> ##1 req; endproperty
am_req_hold: assume property (@(posedge clk) disable iff (!rstn) p_req_hold);
```

L2 [ASSUME] req drops after ack. If `req` and `ack` are both high in a cycle, `req` is low in the next cycle. (Added: the original R2 "then drop" was missing.)

```
property p_req_drop; req && ack |-> ##1 !req; endproperty
am_req_drop: assume property (@(posedge clk) disable iff (!rstn) p_req_drop);
```

L3 [ASSERT] ack within the 2-cycle window. When `req` rises, the DUT asserts `ack` in that cycle or the next one. With L1 and L2 this also bounds `req` to at most 2 cycles, replacing R1.

```
property p_ack_2cyc; $rose(req) |-> ##[0:1] ack; endproperty
ap_ack_2cyc: assert property (@(posedge clk) disable iff (!rstn) p_ack_2cyc);
```

L4 [ASSERT] No ack without req. Whenever `ack` is high, `req` is high. (Added: unrequested `ack` was not covered.)

```
property p_ack_with_req; ack |-> req; endproperty
ap_ack_with_req: assert property (@(posedge clk) disable iff (!rstn) p_ack_with_req);
```

L5 [COVER] Late ack (cycle 3). Show a trace where `ack` arrives 2 cycles after `req` rises. Because L3 forbids this, reaching the cover is a witness of an L3 violation, not a check on environment tightness.

```
sequence s_late_ack; $rose(req) ##2 ack; endsequence
cp_late_ack: cover property (@(posedge clk) disable iff (!rstn) s_late_ack);
```

Gaps found by the back-translation

- **G1:** R2's "then drop" was not expressed; fixed by L2.
- **G2:** R1's 2-cycle limit on `req` is no longer a separate rule; it now follows from L1 + L2 + L3, and the line items say so explicitly.
- **G3:** `ack` without a request was unconstrained; fixed by L4.
- **G4:** the cover changed role once the timing became an assertion (L5).

The complete, commented checker with these properties and the assume/obligation table is listed in **Appendix A**. It compiles cleanly with slang (pyslang 12.2.0, 0 diagnostics); it has not been run in a simulator or formal tool, so the compile is a syntax and elaboration check only.

References

[1] B. Cohen and Dr. K. Chibani, Knowledge-Driven AI for SystemVerilog Assertions, Cohen Publishing (systemverilog.us), 2026.

[2] B. Cohen, S. Venkataramanan, A. Kumari and L. Piper, SystemVerilog Assertions Handbook, 4th ed., VhdlCohen Publishing, 2023.

[3] B. Cohen, Fast-Tracking SVA through Exposure, VhdlCohen Publishing, 2025.

[4] IEEE Std 1800-2023, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE, 2024.

Appendix A. Checker: req_ack_chk.sv

Complete checker for the corrected handshake (Section 7), with the assume/obligation table in its header comments. File: `req_ack_chk.sv`.

```
// req_ack_chk.sv : req/ack handshake checker, ack is a DUT output
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
// Authors: Ben Cohen, Kais Chibani -- systemverilog.us
//
// Companion to the paper "When Assumptions Contradict: What Formal
// Verification Does and What It Hides" (Section 7, Appendix A).
//
// ----------------------------------------------------------------------------
// ASSUME / OBLIGATION TABLE
// ----------------------------------------------------------------------------
// Signal | Driven by   | Direction  | INTENTION   | Properties on it
// -------+-------------+------------+-------------+-------------------------
// clk    | environment | DUT input  | clock       | sampling clock
// rstn   | environment | DUT input  | reset       | disable iff (!rstn)
// req    | environment | DUT input  | ASSUME      | am_req_hold, am_req_drop
// ack    | DUT         | DUT output | OBLIGATION  | ap_ack_2cyc, ap_ack_with_req
// ----------------------------------------------------------------------------
// Item | Kind   | Label           | Requirement (back-translated)
// -----+--------+-----------------+---------------------------------------
// L1   | ASSUME | am_req_hold     | req holds until ack
// L2   | ASSUME | am_req_drop     | req drops the cycle after ack
// L3   | ASSERT | ap_ack_2cyc     | ack in the cycle req rises or the next one
// L4   | ASSERT | ap_ack_with_req | ack is never high without req
// L5   | COVER  | cp_late_ack     | ack 2 cycles after req rises (L3 violation
//      |        |                 | witness)
// ----------------------------------------------------------------------------
// Rules applied (RAG-SVA):
//  * Constraints whose subject is a DUT input are assumptions; constraints
//    whose subject is a DUT output (ack timing) are assertions. Putting the
//    ack timing in an assumption over-constrains the DUT (paper, Section 3.2).
//  * |=> is not used; next-cycle implication is written |-> ##1.
//  * Named properties use the p_ prefix; labels use am_ / ap_ / cp_.
//  * Active-low reset rstn; every directive uses disable iff (!rstn).
// ----------------------------------------------------------------------------

module req_ack_chk (
  input logic clk,   // sampling clock
  input logic rstn,  // active-low reset: all checks disabled while low
  input logic req,   // request, DUT input  (environment drives it)
  input logic ack    // acknowledge, DUT output (DUT drives it)
);

  // ------------------------ Assumptions (environment) ---------------------

  // L1: if req is high and ack has not arrived, req stays high next cycle.
  property p_req_hold;
    req && !ack |-> ##1 req;
  endproperty

  // L2: once ack arrives while req is high, req drops in the next cycle.
  property p_req_drop;
    req && ack |-> ##1 !req;
  endproperty

  am_req_hold: assume property (@(posedge clk) disable iff (!rstn) p_req_hold);
  am_req_drop: assume property (@(posedge clk) disable iff (!rstn) p_req_drop);
```

```
 // ------------------------- Obligations (DUT) ------------------------------

  // L3: when req rises, ack must be high in that cycle or the next one.
  //     With L1 and L2 this also bounds req to at most 2 cycles (old R1).
  property p_ack_2cyc;
    $rose(req) |-> ##[0:1] ack;
  endproperty

  // L4: the DUT never acknowledges without an active request.
  property p_ack_with_req;
    ack |-> req;
  endproperty

  ap_ack_2cyc:     assert property (@(posedge clk) disable iff (!rstn) p_ack_2cyc);
  ap_ack_with_req: assert property (@(posedge clk) disable iff (!rstn) p_ack_with_req);

  // ------------------------- Cover ------------------------------------------

  // L5: ack first arrives 2 cycles after req rises (cycle 3).
  //     L3 forbids this, so a hit is a witness that the DUT can be slow
  //     (an L3 counterexample), not an environment-tightness check.
  sequence s_late_ack;
    $rose(req) ##2 ack;
  endsequence

  cp_late_ack: cover property (@(posedge clk) disable iff (!rstn) s_late_ack);

endmodule : req_ack_chk
```