Constraint Overhead in Formal Verification

A Q&A on helper assumptions, constrained signals, and combined vs. split constraints

---

The exchange below concerns the following passage on formal-verification solver complexity:

> ***A subtlety worth keeping in mind:*** *a very large collection of helper assumptions can itself increase solver complexity. The engine must simultaneously satisfy all of them, and redundant or overlapping constraints add overhead. If adding more helper assumptions degrades performance rather than improving it, pruning the set to the minimal necessary subset often restores convergence.*

QUESTION

You are an expert in chip design — explain and expand on formal verification. Does this overhead apply when only design signals are constrained, or also FV signals? And does that mean writing a constraint such as `$stable(signal) && signal < MAX_VALUE` in one property is better than splitting them?

Expanding the Passage

When a formal tool runs, it isn't just checking assertions — it's solving a constraint-satisfaction problem over *all* active constraints at once. Every `assume` property, every helper assumption, every environmental restriction becomes part of the reachable-state computation the engine must maintain on every cycle it explores. The solver builds and prunes a representation of "which states are legal given everything I've been told must hold." Each constraint added is another condition that representation must simultaneously respect.

That is why a very large collection can *increase* complexity. The cost is not linear-and-free. Two effects drive it:

- **Redundancy** — if constraint B is already implied by constraint A, the engine still carries B, still evaluates it, still ANDs it into the reachable-state logic every cycle. It buys no additional restriction but costs compute.

- **Overlap / interaction** — when two constraints touch the same signals over the same cycles, the engine may have to reason about their conjunction, which can blow up the state representation (in BDD-based engines, variable ordering and node count; in IC3/PDR, the number of proof obligations and lemmas). Overlapping constraints can force the solver to repeatedly re-derive relationships it cannot easily simplify away.

The practical symptom is a proof that was converging suddenly slowing, timing out, or exhausting memory as "helpful" assumptions accumulate. Pruning to the minimal necessary subset works because a smaller, non-redundant constraint set gives the engine a smaller reachable-state problem to maintain.

Does the Overhead Apply to Design Signals Only, or FV Signals Too?

The overhead is fundamentally about constraints on **inputs** and **free / unconstrained** signals — not about which "side" a signal is labeled. The distinction that matters to the engine:

A formal tool drives the design's **primary inputs** (and any free variables) to explore behavior. Constraints (`assume` properties) restrict how those inputs may move. That restriction shapes the reachable state space, and that is where the cost lives.

- **Constraining input / environment signals** (the normal case): directly prunes the input space the solver explores. This is where both the benefit *and* the overhead concentrate. A tight, well-chosen input constraint shrinks the problem; a pile of overlapping input constraints adds conjunction overhead.
- **"Constraining" internal design signals**: you generally don't — and often can't soundly — `assume` on an internal signal the design itself drives. If you do, the engine treats it as an additional invariant it must hold, which can over-constrain the proof (hiding real bugs) *and* add the same solver overhead. This is usually a modeling error, not a tuning knob.
- **Auxiliary / modeling signals introduced for FV** (helper free variables, abstraction "cut" signals, symbolic constants): constraints on these absolutely carry overhead too — arguably more, because they add state variables that did not exist in the design. A symbolic constant with `assume`d bounds, or an abstraction where a signal is freed and then re-constrained, each adds to the same reachable-state computation.

So: the overhead applies to **any constrained signal that participates in the reachable-state computation** — design inputs and FV-modeling signals alike. It is *not* limited to design signals. If anything, FV-specific modeling signals are a common hidden source of the exact "too many

overlapping constraints" slowdown the passage warns about, because engineers add them for convenience without counting their cost. What does *not* add meaningful overhead is a signal the design fully determines and that you never assume anything about — the engine computes it; it is not a degree of freedom.

Combined vs. Split Constraint

For solver performance: no meaningful difference — and the decision should not be made on performance grounds. Whether written as:

```
am_combined: assume property (@(posedge clk) $stable(signal) && signal < MAX_VALUE);
```

or:

```
am_stable: assume property (@(posedge clk) $stable(signal));
am_range:  assume property (@(posedge clk) signal < MAX_VALUE);
```

the engine sees the **same conjunction** either way. Internally, all active assumptions are ANDed together into the environment constraint — two conditions combined with `&&` in one property, or two separate `assume`s, produce the same logical restriction and essentially the same reachable-state computation. The solver does not care about property boundaries; it cares about the total conjoined constraint. Splitting one property into two constraints that are identical in aggregate does **not** add the overhead the passage describes.

The overhead the passage warns about comes from constraints that are **redundant or overlapping in content** — e.g., separately assuming `signal < MAX_VALUE` and `signal <= MAX_VALUE - 1` (redundant), or `signal inside {[0:100]}` and `signal < 200` (overlapping; the second adds nothing). That is a *semantic* redundancy, and it costs the same whether written as one property or ten. Combining vs. splitting is a *syntactic* choice and is performance-neutral.

What should drive the decision (not performance)

- **Debuggability / traceability** — separate assumptions each get their own name, so when the tool reports a vacuous proof or an over-constraint, it points at `am_range` or `am_stable` specifically. A combined `&&` property fails as a unit, losing which conjunct did the damage. This alone usually argues for splitting.

- **Independent coverage and control** — pairing each assumption with a satisfiability cover, or toggling/waiving one constraint without the other, requires them to be separate.
- **Semantic grouping** — if two conditions genuinely express one environmental fact and are meaningless apart, combining reads more honestly. Here, `$stable(signal)` (a transition claim) and `signal < MAX_VALUE` (a range claim) are two independent facts, which again favors splitting.

Recommendation for this example: split them. They are two distinct environmental claims — a stability claim and a range claim — and separate `assume`s give per-constraint diagnostics and per-constraint covers with zero performance penalty. Combine only when conditions are semantically inseparable, and even then, know that the trade is against debuggability, not for speed.

Caveat. This is the general behavior of modern formal engines, but exact internal handling varies by tool (JasperGold, VC Formal, Questa Formal each have their own constraint-processing and simplification). When chasing a specific convergence problem, the tool's own constraint/complexity reports are ground truth over any general rule.

Generated from a working conversation on formal-verification methodology. Provided as-is for reference; verify tool-specific behavior against your formal engine's documentation.