Formal Verification Scope Analysis After a Design Change

Ben Cohen & Kais Chibani, SystemVerilog.us

Question

When a design changes from Design A to Design B, how do you decide which formal-verification properties, assumptions, and harness conditions need to be reconsidered? What formal evidence should be captured before the change, what should be re-examined afterward, how do you determine the affected scope, and do commercial FV tools actually support this kind of design-change impact analysis?

Response *(Claude Sonnet 5)*

Good question, and one that sits right at the boundary between the Cohen-Chibani's RAG-SVA rule corpus (which governs how a property is written) and general FV methodology (which governs when a property needs to be revisited at all). The RAG-SVA reference files don't actually contain a change-impact-analysis section — they're about generating correct SVA for a given design snapshot — so the mechanism below combines the one piece of the framework that's directly relevant (*the Port-Intent / INTENTION classification*) with standard commercial FV practice.

Before the Change: What Evidence You Need in Hand

You can't tell what changed if you don't have a frozen baseline. Four things need to exist before Design B lands.

A **signed-off proof status matrix** — every property/assumption, its status (proven, bounded to depth N, CEX-free but not proven, vacuous, waived), the engine and depth used, and runtime/complexity. Without this, "did the change break anything" degenerates into "let's re-run everything and see."

The **Port-Intent table from RQM2 (§3 of the RAG-SA framework)** — the assume/obligation/N-A tag for every signal that appears in a property. This is the single most useful artifact for change-impact triage, because a huge share of "which assumptions need reconsidering" reduces to "did any signal's INTENTION tag change" (more on this below).

A **documented assumption rationale** — not just the assumption text, but why it's valid (e.g., "assume no simultaneous push+pop when full — guaranteed by the arbiter upstream, verified in block X"). An assumption without a rationale is undebuggable after a change, because you can't tell whether the reason it was valid still holds.

Vacuity/antecedent-coverage results, and a list of exclusions — blackboxed hierarchies, cut points, case splits, waived properties. A property that was quietly vacuous before the change is exactly the kind of thing that either stays vacuous (harmless) or becomes non-vacuous and starts failing (informative) after B — you need the "before" number to know which happened.

After the Change: What to Reconsider, by Change Category

- **Interface/protocol change** (new or removed ports, changed handshake): re-run every signal touched through the Port-Intent classification again. This is the concrete trigger for HALT row 3 / CASE D in §4.1 — if a signal that was "assume" becomes "obligation" (or vice versa) because logic moved across the design boundary, an **mp_** (assumption) needs to become an **ap_** (assertion to prove), or you'll keep "verifying" against a constraint the DUT itself is now supposed to guarantee. That's a silent correctness hole, not a compile error, so it has to be caught by re-walking the table, not by re-running the proof.

- **Internal microarchitecture change, same external contract** (pipeline restructuring, arbiter algorithm swap, FIFO depth/pointer-encoding change): interface-level properties should still hold and are a regression check. But anything with a hardcoded timing window (##[m:n], latency bounds) needs its bound re-derived from the new pipeline depth — per §4.0's boundedness axis, a property that was legitimately Bounded at depth 3 doesn't silently become correct at depth 5 just because the proof still terminates; the bound itself is a design fact that changed.
- **Reset/clock changes**: revisit **disable iff** conditions and any assumption about clock relationship (synchronous vs. now-CDC). These sit at the boundary of "structural" and "semantic" — COI tools will flag them, but only a person/requirement review will catch that a formerly-synchronous assumption is now unsafe.
- **Parameter change** (depth, width, port count): this is exactly why the framework insists on parameterized properties tied to the design's parameters rather than magic numbers — if a property was hand-coded with depth-1 as a literal instead of referencing the parameter, it goes silently stale on a parameter change and won't even show up in a COI diff, because the property's own source didn't change.

Determining the Affected Scope: Structural vs. Semantic

Two different questions get conflated here, and it matters that you keep them separate.

Structural scope ("did the RTL diff touch this property's logic") is what fan-in cone-of-influence (COI) analysis answers. Take the property's COI (the fan-in cone from every signal it references) and intersect it with the RTL diff. Three buckets fall out: (1) diff outside the COI — the property is untouched, replay the cached result; (2) diff inside the COI but the property still holds — re-run the proof (the model changed, so the old result isn't valid as-is, but no rewrite needed); (3) diff inside the COI and it touches the property's antecedent/consequent signals directly — treat as a candidate rewrite, not just a re-run.

Semantic scope ("does this property still say what it's supposed to say") is what COI can't tell you. The FIFO example below is the clean illustration: the overflow property's COI absolutely includes the pointer-compare logic both before and after switching from binary to Gray-code pointers, so a COI tool correctly flags it as affected — but it can't tell you that the comparison operator itself needs to change from arithmetic to Gray-code-aware logic. That's a requirement-level review, driven by the Port-Intent table and the original requirement text, not a structural diff.

Do FV Tools Support This? Concretely, Yes for Scope, No for Semantics

The major commercial formal tools (Cadence JasperGold, Synopsys VC Formal, Siemens Questa Formal/PropCheck) all provide: a per-property COI report you can generate and diff against source control; incremental/cached proof reuse keyed off that COI, so an unaffected property's prior result is replayed rather than re-proven from scratch; and regression dashboards that track pass/fail/vacuity status across runs so a status flip is visible immediately rather than buried in a log. That's genuinely useful — it turns "re-verify everything after every RTL change" into "re-verify the dozen properties whose cone the diff actually touches."

What none of them do is tell you a property's meaning is now wrong while its proof still passes cleanly — the FIFO overflow property with the wrong comparator will happily re-prove if the new pointer-compare logic is simply buggy in a way the tool's engines can't distinguish from "correct new logic." That gap — structural coverage without semantic coverage — is precisely the problem your own RAG-SVA papers are aimed at (the Port-Intent/requirement-traceability step is a deliberate answer to it), so it's worth stating plainly in any seminar or paper context: COI-based incremental formal is a scope-reduction tool, not a change-safety tool.

Two Worked Examples

FIFO, Design A → B

Depth 8 with binary pointers → depth 16 with Gray-code pointers for a new CDC boundary. Baseline: ap_no_overflow, ap_no_underflow, ap_full_empty_consistency all proven at depth 8, mp_no_simultaneous_push_pop_when_full assumed with rationale “guaranteed by upstream arbiter.”
After B: COI diff shows the pointer-compare logic changed → ap_no_overflow/ap_underflow land in bucket 3 (rewrite candidates) because binary-pointer comparison logic doesn't mean the same thing under Gray coding — the comparator needs to be rewritten, not just re-proven. The push/pop assumption is structurally untouched (bucket 1) but should still get a manual look, because the new CDC boundary adds synchronization latency the old rationale never accounted for — a case where structural scope says “skip” and semantic scope says “check anyway.”

Arbiter, Design A → B

Fixed-priority → round-robin (this is the same grnt/req territory as your own paper). Baseline: safety properties like ap_grant_implies_request (grnt[i] |-> req[i]) and a liveness property ap_eventual_grant (s_eventually every asserted req[i] gets grnt[i]) proven, the liveness proof carrying a fairness assumption tied to the priority scheme.
After B: ap_grant_implies_request's COI barely changes and it re-proves as-is (bucket 2). ap_eventual_grant is the interesting one — its COI touches the arbiter's core logic directly (bucket 3), and more importantly the fairness assumption itself is priority-scheme-specific: a fixed-priority fairness assumption (“low-priority requester X eventually gets service because Y stops requesting”) is simply the wrong assumption for round-robin, where fairness follows from the rotation instead. Re-running the old proof with the old assumption against the new arbiter will either vacuously pass or fail for the wrong reason — this is a case where the Port-Intent re-classification (is the priority signal still N/A, or did it become something a property needs to reason about directly?) is what catches the problem, not the COI tool.