Where Do FV Abstraction Techniques Live in a Rule-Governed SVA Methodology?

Cut Points, Black-Boxing, Symbolic Variables, and Scoreboards in the RAG-SVA Framework

Ben Cohen · Dr. Kais Chibani

Cohen Publishing · systemverilog.us

1. The Question

Any rule-governed approach to assertion generation must eventually face the following question:

> *Does a rule-governed SVA generation methodology account for the abstraction techniques that a formal verification engineer must apply in real projects to reach convergence — cut points, black-boxing, scoreboards, symbolic free variables, and the like? These are not optional refinements; on real designs they are frequently the difference between a proof that converges and one that does not.*

The question deserves a direct answer, because it touches on a boundary that any assertion-generation methodology must draw honestly: what belongs to the rule corpus, and what belongs to the formal tool flow?

The short answer is a division of labor. Applying an abstraction — cutting a signal, black-boxing a module, declaring a symbolic variable — is a tool-flow operation, executed in the formal tool's environment, and is largely outside what an assertion rule corpus formalizes. But every one of those abstractions produces side effects that land squarely inside the rule corpus's jurisdiction: new free signals, new assumptions constraining them, new opportunities for over-constraint and vacuity. Abstraction is applied at the tool; it is kept honest in the rules. The remainder of this note works through each named technique with a concrete, rule-compliant example.

2. Black-Boxing: The Interface Contract Becomes Assumptions

When a submodule is black-boxed, the tool removes its internals and its outputs become free (undriven) signals. From the perspective of the enclosing proof, those signals undergo an INTENTION flip: what was a DUT obligation when verifying the submodule standalone is now environment behavior — the proof can no longer derive the signals' values from logic, so their legal behavior must be stated as assumptions.

Consider a design containing a FIFO that is black-boxed for convergence. Its outputs fifo_empty and fifo_rd_valid are now free. The FIFO's interface contract is pinned down with a documented assumption:

```
/*
ASSUMPTION:    Black-boxed FIFO never signals valid read data while empty.
TYPE:          Input Constraint
JUSTIFICATION: FIFO interface contract (proven separately as an assertion
                in the FIFO-standalone FV run -- assume-guarantee pairing).
RISK LEVEL:    Medium -- if stronger than the real FIFO, proofs here are void.
*/
mp_bb_fifo_no_valid_when_empty: assume property (
    @(posedge clk) disable iff (!rst_n)
    fifo_empty |-> !fifo_rd_valid);

cp_bb_fifo_no_valid_when_empty_hit: cover property (   // reachability pair
    @(posedge clk) disable iff (!rst_n) fifo_empty);
```

Three rule-corpus mechanisms are doing real work here. The documentation block makes the assumption's type, justification, and risk explicit and auditable — essential because every black-box assumption is a potential source of false proofs if it is stronger than the real block's behavior. The paired reachability cover is the over-constraint alarm: if fifo_empty becomes unreachable after abstraction, the assumption never engaged, and something upstream has over-constrained the proof. And the JUSTIFICATION field records the assume-guarantee obligation: whatever is assumed at the black-box boundary here must be proven as an assertion when the FIFO is verified standalone. Skipping that pairing converts an abstraction into an unsoundness.

3. Cut Points: State Reduction with a Documented Risk

Definition — cut point: a formal-tool operation that severs a signal from its fan-in (driving logic). The tool discards everything that drives the signal and treats the signal itself as a free primary input — able to take any value at any cycle unless re-constrained by assumptions. The driving logic and all of its state drop out of the proof's cone of influence, which is the point: a wide accumulator, a deep counter, or a complex datapath feeding the cut signal no longer contributes state to the proof. The cost is soundness exposure in one direction only — cutting makes the proof more pessimistic (the free signal can now do things the real logic never would), so a property proven with a cut is still sound; but any assumption added to tame the freed signal can over-constrain and mask real bugs, which is why the example below carries RISK LEVEL: High and a boundary-reachability cover.

A cut point is typically applied to remove a large state contributor (a wide accumulator, a deep counter) from a proof that does not depend on its detailed behavior. The freed signal is then optionally re-constrained to its legal range.

```
/*
ASSUMPTION:    Cut-point signal acc_q constrained to its reachable range.
TYPE:          State Reduction
JUSTIFICATION: acc_q proven <= MAX_ACC in a separate dedicated run;
                cutting it here removes 2^32 states from this proof.
RISK LEVEL:    High -- HUMAN REVIEW REQUIRED (range must match the
                separate proof exactly, or bugs above the range are masked).
*/
mp_cut_acc_range: assume property (
    @(posedge clk) disable iff (!rst_n) acc_q <= MAX_ACC);

cp_cut_acc_range_hit: cover property (
    @(posedge clk) disable iff (!rst_n) acc_q == MAX_ACC);  // boundary reachable
```

The cut itself is one line in a tool script. The rule corpus's contribution is everything around it: the High risk level flags the assumption for human review; the justification ties the assumed range to the separate proof that establishes it; and the boundary cover confirms the extreme of the assumed range is actually exercised. A cut point without this documentation is an invisible soundness dependency; with it, the dependency is explicit and checkable.

4. Symbolic Free Variables: The Heart of the Formal Scoreboard

The most powerful of the techniques raised, and the one with the deepest connection to the rule corpus's property-writing rules. A symbolic (rigid) free variable is an undriven signal constrained only to remain stable; the formal tool then verifies the property for every possible value of that variable simultaneously. This is the classic “track one symbolic transaction” pattern — and, as Section 5 develops, it is the core of what “scoreboard” actually means in a formal environment.

```
logic [7:0] sym_addr;   // undriven -- free variable, tool picks any value

/*
ASSUMPTION:    sym_addr is rigid (constant per trace).
TYPE:          State Reduction
JUSTIFICATION: Standard symbolic-variable idiom; proof covers all 256
                addresses simultaneously without a per-address scoreboard.
RISK LEVEL:    Low
*/
mp_sym_addr_stable: assume property (
    @(posedge clk) ##1 $stable(sym_addr));

cp_sym_addr_stable_hit: cover property (
    @(posedge clk) disable iff (!rst_n) wr_en && (wr_addr == sym_addr));

// Data integrity at the symbolic address -- capture-at-trigger,
// single implication. Window fused at the write cycle (##0): the write
// consumes position 1, so 1[*2:17] puts the read at T1..T16, and the
// length-2 minimum excludes a same-cycle read -- matching ##[1:16].
property p_sym_data_integrity;
    logic [31:0] v_data;
    ((wr_en && (wr_addr == sym_addr), v_data = wr_data)
       ##0 ((rd_en && (rd_addr == sym_addr))[->1] intersect 1[*2:17]))
      |-> (rd_data == v_data);
endproperty
ap_sym_data_integrity: assert property (
    @(posedge clk) disable iff (!rst_n) p_sym_data_integrity)
  else $error("[FV] data corruption at symbolic addr %0h, t=%0t", sym_addr, $time);
```

The state-space contrast is the whole argument. A simulation scoreboard for this check would model per-address storage — 256 addresses times 32 bits of tracking state, all of it added to the proof's state space. The symbolic version carries exactly one captured word (the property-local variable v_data) and proves the same data-integrity claim for every address at once. The rule corpus's local-variable capture rules — capture referenced data at the triggering event, compare the captured copy later, never re-sample the original signal — are precisely the property-level half of this technique; the symbolic variable is the environment-level half.

Three construction notes on the example, included because a careful reviewer will ask. First, the property uses exactly one implication operator. The read event is detected in the antecedent — composed after the write with the goto operator, so the antecedent means “a write to the symbolic address, then the first subsequent read of that address within the window” — and the sole consequent carries only the data obligation. The tempting alternative, a nested implication in the consequent (… |-> ##[1:16] read_event |-> data_match), expresses the same intent but violates the single-implication rule; the goto composition is the rule-compliant equivalent. Second, the naive single-implication rewrite — moving the read into a plain conjunction in the consequent (|-> ##[1:16] (read_event && data_match)) — is semantically wrong: it would force a read to occur in the window, silently converting a data-integrity check into a liveness obligation. The goto-composed antecedent preserves the intended non-forcing behavior: if no read of the symbolic address ever occurs, the antecedent never completes and the property passes vacuously, exactly as a data-integrity check should. Third, the window operand and its bounds deserve care in two ways that are easy to get wrong: the intersect operand must be a sequence, so the window is spelled 1[*2:17] (a bare [2:17] is not legal SVA), and the offset must be applied exactly once — the window is fused at the write cycle with ##0, so the write consumes position 1 and 1[*2:17] places the read at T1..T16; combining a ##1 with the offset bounds double-counts the shift and silently moves the window to T2..T17.

5. Scoreboards in FV: A Different Construction, Not a Port

The term “scoreboard” means something structurally different in formal than in simulation, and the difference is exactly the symbolic-variable technique of Section 4. A simulation scoreboard tracks everything: per-entry storage for all outstanding transactions, compared as each one retires. Porting that construction into a formal environment multiplies the state space by the tracking store and is the wrong move essentially always.

The formal scoreboard instead tracks one symbolic transaction — and, through the symbolism, all of them at once. It is built from three small pieces:

- Symbolic selection — a rigid free variable (or free trigger) nondeterministically marks one packet at ingress. Because the variable is symbolic, the tool effectively examines every possible choice of tracked packet in a single proof.
- Ordering by counting, not storage — a small counter records how many transactions are ahead of the tracked one when it enters. At egress, the tracked packet must emerge after exactly that many prior exits. No queue model, no per-entry storage: one counter carries the entire ordering obligation.
- Integrity at egress — the payload captured at ingress (a property local variable or small auxiliary register) must match at egress, either exactly or under an agreed transformation — e.g., header fields may be rewritten while the payload must be untouched.

Total added state: one symbolic value, one counter, one captured word — regardless of how many transactions the design can hold in flight. Ordering and data integrity, the two properties a scoreboard exists to check, are both proven for every packet simultaneously. That is why the formal scoreboard converges where the ported simulation scoreboard does not: it is Section 4's abstraction wearing a scoreboard's job description.

Where genuinely multi-entry auxiliary tracking is unavoidable, it should be minimal, sized to the property being proven, and its own state treated as a candidate for the same abstraction techniques — which closes the loop back to Sections 2 and 3.

6. The Honest Boundary, and Future Work

To answer the question posed at the outset directly: the current rule corpus does not contain rules that decide when to cut a signal or which module to black-box. Those are proof-strategy decisions, made in the tool flow, informed by convergence behavior — and pretending a text corpus could automate them would overclaim. What the corpus does govern, without exception, is the residue every abstraction leaves behind: the assumptions. Every technique in this note terminated in the same place — an assume property with a documentation block, a risk level, a justification, and a paired reachability cover. That is not a coincidence; it is the design. Abstractions are applied at the tool and kept honest in the rules, and the chapters on assumptions, determinism, and vacuity are the machinery that does the keeping.

The natural future-work item is an explicit abstraction-patterns treatment in the rule corpus: named rules for the black-box INTENTION flip, cut-point range assumptions with mandatory companion-proof justification, and the symbolic-variable idiom as a first-class generation pattern with its stability assumption and reachability cover generated as a unit. The examples in this note are, in effect, the first draft of those rules.

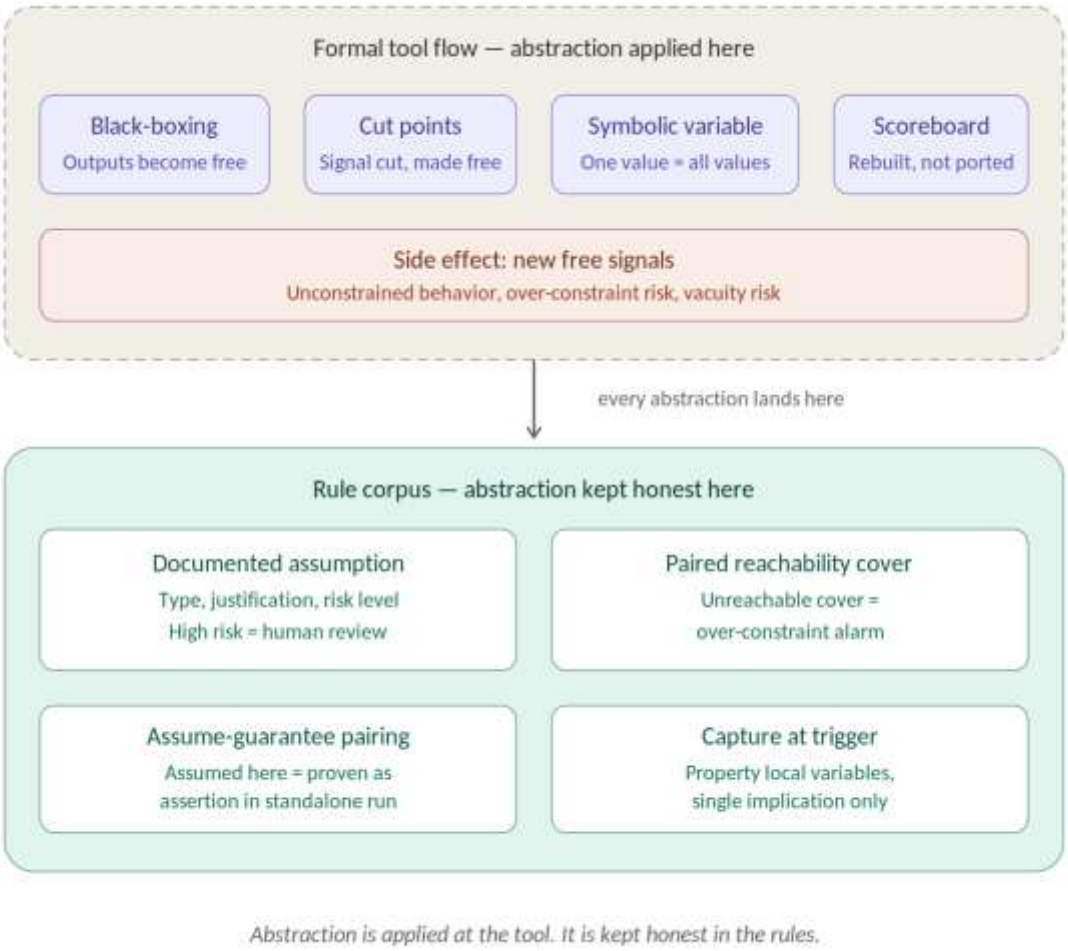

Figure 1: The division of labor — abstraction techniques are applied in the formal tool flow (top); their side effects, the assumptions, are governed by the rule corpus (bottom).

References

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

[2] E. Seligman, T. Schubert, M.V.A.K. Kumar. Formal Verification: An Essential Toolkit for Modern VLSI Design, 2nd Ed. Elsevier, 2023.

[3] B. Cohen, S. Venkataramanan, A. Kumari, L. Piper. SystemVerilog Assertions Handbook, Revised 4th Edition. VhdlCohen Publishing, 2023.

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