Testbench Styles Supported by RAG-SVA: Directed, Fast Random, and UVM

Ben Cohen, Kais Chibani
Cohen Publishing · systemverilog.us

1. Introduction

A SystemVerilog Assertion is only as trustworthy as the stimulus that has exercised it. A property that has never fired, never almost-fired (*e.g., becomes vacuous*), and never seen an input combination outside the one case its author had in mind offers very little real assurance, no matter how carefully the property itself was written. RAG-SVA treats testbench generation as part of the same workflow as assertion generation, not an afterthought, and it supports three distinct testbench styles for exercising a generated checker: a directed, deterministic test; a quick sanity check (with or without DUT) with constrained-random stimulus; and a full UVM-integrated environment.

These three styles are not competing alternatives to pick one of. They are a progression. Each stage reuses the exact same checker - the same assert property and assume property statements produced by the RAG-SVA classification step - and changes only how that checker's inputs are driven. Understanding why all three are supported, and what each one does and does not prove, is the subject of this paper.

The RAG-SVA classification step referenced throughout this paper - the INTENTION pass that decides `assert property` versus `assume property` before any testbench is written - is described in full in Cohen and Chibani, "*Knowledge-Driven AI for SystemVerilog Assertions*" (2026).

2. Why Three Styles, Not One

It would be simpler to standardize on a single testbench style project-wide. In practice, three pressures push RAG-SVA to support all three.

- Explainability comes first. Before trusting a property's behavior to random stimulus or to a UVM sequence library, an engineer needs to see it do the right thing on the one concrete scenario the requirement describes. A directed test gives a single, readable waveform that anyone reviewing the checker can walk through by hand.
- Breadth requires more than one engineer can write by hand. Even a careful reviewer will not think to write every dist-weighted combination of **req**, **rdy**, and **ack** that a constrained-random loop generates in a couple hundred cycles. Fast random stimulus finds interactions a human did not anticipate, cheaply.
- Project-scale reuse requires shared infrastructure. Once a checker leaves the sandbox and is deployed against a real DUT inside a larger SoC verification effort, its pass/fail results need to speak the same language - severity levels, log formatting, coverage hooks - as everything else in the environment. That is what the UVM-integrated style provides.

Because all three styles wrap the same underlying checker rather than re-deriving it, moving from one style to the next never risks re-introducing the INTENTION classification errors (*assume vs. obligation*) that the RAG-SVA framework front-loads and locks down before any testbench code is written.

3. Style 1 - Directed (Deterministic)

This is the mandatory first step and the one RAG-SVA insists on before any random stimulus is layered on top (Rule SIM_5). The testbench applies a small, fixed, hand-chosen sequence of values - **reset**, then **req** asserted for exactly one cycle, then **rdy** and **ack** in the timing relationship the requirement describes - and produces one clean, minimal waveform. Nothing is random; every value on every cycle is chosen by the author for a reason.

The purpose of this style is narrow and specific: prove that the checker fires (or does not fire) exactly when the requirement says it should, on the case that everyone agrees is the canonical example. It is a correctness check on the assertion itself, not a coverage exercise on the design. A directed test that passes tells you the property's logic matches its English description for that one scenario; it says nothing about any other input combination.

```
// tb_req_ack_directed.sv - SIM_5 directed sanity testbench for chkr
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// Checker under test (reused unmodified - do not edit):
//   req  : input      (INTENTION = assume, testbench-driven)
//   rdy  : output     (INTENTION = obligation, DUT-driven)
//   ack  : output     (INTENTION = obligation, DUT-driven)
//   resetn : active-low reset
//
// checker chkr (input bit clk, resetn, req, rdy, ack);
//   ap_req_ack: assert property (@(posedge clk) disable iff (!resetn)
//     req |-> (rdy, ##[1:5] ack))
//     else `uvm_error("SVA", $sformatf(
//       "Error in req->ack. req=%b rdy=%b ack=%b", req, rdy, ack));
// endchecker: chkr

module tb_req_ack_directed;
  bit clk, resetn, req, rdy, ack;

  // Clock generator
  initial forever #5 clk = !clk;

  // Checker instance - standalone, no DUT (SIM_5)
  chkr chkr1 (.clk(clk), .resetn(resetn), .req(req), .rdy(rdy), .ack(ack));

  initial begin
    // Initialize - reset asserted (active-low)
    resetn = 0; req = 0; rdy = 0; ack = 0;
    repeat (2) @(posedge clk);

    // ------------------------------------------------------------
    // Scenario 0 - PASS: property must be disabled during reset.
    // req is asserted with a wrong rdy and no ack at all; if the
    // disable iff (!resetn) guard is working, this must NOT fire.
    // ------------------------------------------------------------
    req = 1; rdy = 0;                // would violate the property if checked
    @(posedge clk);
    req = 0;
    repeat (2) @(posedge clk);       // ack never asserted - still no error expected

    resetn = 1;                      // release reset
    @(posedge clk);
```

```
    // ------------------------------------------------------------
    // Scenario 1 - PASS: ack at the earliest legal cycle (##1)
    // ------------------------------------------------------------
    req = 1; rdy = 1; @(posedge clk); // cycle t: req=1, rdy=1 sampled
    req = 0; rdy = 0;
    ack = 1; @(posedge clk);          // cycle t+1: ack=1 - PASS
    ack = 0;
    repeat (3) @(posedge clk);        // idle gap before next scenario

    // ------------------------------------------------------------
    // Scenario 2 - PASS: ack at the latest legal cycle (##5)
    // ------------------------------------------------------------
    req = 1; rdy = 1; @(posedge clk); // cycle t: req=1, rdy=1 sampled
    req = 0; rdy = 0;
    repeat (4) @(posedge clk);        // cycles t+1 .. t+4, ack still 0
    ack = 1; @(posedge clk);          // cycle t+5: ack=1 - PASS (window boundary)
    ack = 0;
    repeat (3) @(posedge clk);

    // ------------------------------------------------------------
    // Scenario 3 - FAIL (intentional): ack one cycle too late (##6)
    // ------------------------------------------------------------
    req = 1; rdy = 1; @(posedge clk); // cycle t: req=1, rdy=1 sampled
    req = 0; rdy = 0;
    repeat (5) @(posedge clk);        // cycles t+1 .. t+5 elapse, window closes
    ack = 1; @(posedge clk);          // cycle t+6: ack=1 - too late, FAIL expected
    ack = 0;
    repeat (3) @(posedge clk);

    // ------------------------------------------------------------
    // Scenario 4 - FAIL (intentional): rdy not asserted with req
    // ------------------------------------------------------------
    req = 1; rdy = 0; @(posedge clk); // cycle t: req=1 but rdy=0 - FAIL expected
    req = 0;
    repeat (3) @(posedge clk);

    $finish;
  end
endmodule: tb_req_ack_directed
```

Notice the checker module - **chkr** - is identical to the one used in every other style in this paper. The directed testbench differs only in how it drives **req**, **rdy**, and **ack**: by explicit assignment, on specific cycles, with no randomization at all.

4. Style 2 - Fast Random (Constrained-Random)

Once the directed test has established that the checker's logic is correct for the canonical case, RAG-SVA's Rules SIM_6 (with DUT) and SIM_7 (without DUT) add a short constrained-random phase on top of the same checker. A repeat loop - typically on the order of 200 cycles - calls **randomize()** with **dist** weights on each input signal every cycle, and any randomization failure or assertion failure is reported immediately.

The word "fast" matters here in two senses. First, the loop itself runs quickly - a couple hundred clock cycles is a light simulation load, not a long regression. Second, and more importantly, it is fast to write and fast to reason about: a few **dist** constraints (*which can easily be modified*) replace what would otherwise be dozens of hand-written directed cases, and it finds combinations - **req** and **ack** both toggling in adjacent cycles, back-to-back requests before an acknowledge completes - that a directed list is unlikely to include unless someone specifically thought to write them.

```
// tb_req_ack_fast_random.sv - SIM_7 fast-random sanity testbench for chkr
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// Style 2 - Fast Random (Constrained-Random), standalone checker, no DUT.
// Same chkr from the directed (SIM_5) testbench - unmodified.
//
// checker chkr (input bit clk, resetn, req, rdy, ack);
//   ap_req_ack: assert property (@(posedge clk) disable iff (!resetn)
//     req |-> (rdy, ##[1:5] ack))
//     else `uvm_error("SVA", $sformatf(
//       "Error in req->ack. req=%b rdy=%b ack=%b", req, rdy, ack));
// endchecker: chkr
//
// req  : input  (INTENTION = assume, testbench-driven -> randomized here)
// rdy  : output (INTENTION = obligation, DUT-driven -> randomized here only
//                because no DUT exists yet; once a DUT is present, rdy/ack
//                move to SIM_6 and are driven by the RTL, not randomize())
// ack  : output (INTENTION = obligation, DUT-driven -> see note above)

module quick_tb;
  bit clk, req, rdy, ack, resetn;
  string log_id = "SVA";

  // Clock generator
  initial forever #5 clk = !clk;

  // Checker instance - standalone, no DUT (SIM_7)
  chkr chkr1 (.clk(clk), .resetn(resetn), .req(req), .rdy(rdy), .ack(ack));

  initial begin
    // Reset sequence (active-low)
    resetn = 1; repeat (2) @(posedge clk);
    resetn = 0; @(posedge clk);
    resetn = 1;

    // Fast, constrained-random stimulus: ~200 cycles
    repeat (200) begin
      @(posedge clk); #1;
      if (!randomize(req, rdy, ack) with {
            req dist {1'b1 := 1, 1'b0 := 3};  // req high ~25%
            rdy dist {1'b1 := 1, 1'b0 := 3};  // rdy high ~25%
            ack dist {1'b1 := 1, 1'b0 := 5};  // ack high ~17%
          })
        `uvm_error(log_id, "This is a randomize error");
      `uvm_info(log_id, $sformatf("t=%0t req=%b rdy=%b ack=%b",
                $realtime, req, rdy, ack), UVM_MEDIUM);
    end

    $finish;
  end
endmodule: quick_tb
```

This style answers a different question than the directed test does. It does not re-prove the checker's logic - that was already established in Stage 1 - it asks whether the checker keeps behaving sensibly (and whether the design, once one exists, keeps meeting its obligations) across a wider, faster-to-generate sample of the input space. It is deliberately not exhaustive: **dist** weights bias toward "interesting" and away from illegal combinations, but a rare corner case can still slip through 200 random cycles by chance. Random breadth is a complement to directed precision, not a replacement for it, and neither one is a claim that "most" states have been covered in any provable sense - only formal verification, a separate RAG-SVA deployment target outside the scope of this paper, makes that claim exhaustively.

5. Style 3 - UVM-Integrated

The third style does not change the checker, or fundamentally the idea of driving it with stimulus; it changes the environment the checker lives in. Inside a UVM testbench, the same assert property statements report through `**uvm_error** and informational messages through `**uvm_info** rather than bare **$error** and **$display** calls, exactly as shown in the Stage 1 and Stage 2 examples above (both already use UVM macros for reporting, by RAG-SVA convention, so the checker code does not need to change at all when it is finally placed inside a class-based environment).

What changes at this stage is who and what drives the checker's inputs. Instead of a single initial block with fixed values or a single **randomize()** call, stimulus now comes from UVM sequences running on sequencers, monitored by UVM monitors, and cross-checked by scoreboards; coverage groups can sample the same signals the checker watches. The checker's pass/fail verdict is now one data point among many that the environment's reporting and regression infrastructure aggregates, categorizes by severity, and can gate a regression pass/fail decision on.

This style is the right one specifically when a check needs to live alongside other IP-level or SoC-level checks that already use UVM, when regression results need to roll up through a common report format, or when the stimulus itself needs to come from a reusable sequence library rather than a one-off testbench. It is not a replacement for Stages 1 and 2 - a checker is still sanity-checked with a directed test and, often, a fast-random test before it is trusted inside the larger, slower-to-iterate UVM environment.

A minimal example makes the structural change concrete: the RTL design now exists as a real module, the same **chkr** checker from Stages 1 and 2 is attached to it with a single bind statement rather than an explicit port connection, and the surrounding testbench drives req through a UVM sequence and sequencer instead of a fixed value or a stand-alone **randomize()** call.

```
// tb_req_ack_uvm.sv - Style 3 (UVM-Integrated) testbench for chkr
// LICENSE: Personal/academic use only. Commercial use requires a Commercial License.
//
// Same chkr from the directed (SIM_5) and fast-random (SIM_7) testbenches -
// unmodified. Only the environment around it changes: a real DUT now exists,
// chkr is attached via bind rather than instantiated directly, and req is
// driven by a UVM sequence/sequencer instead of a fixed value or a bare
// randomize() call. Reporting stays on `uvm_error/`uvm_info throughout
// (Item 19 = auto, UVM detected), exactly as chkr already does.
//
// checker chkr (input bit clk, resetn, req, rdy, ack);
//   ap_req_ack: assert property (@(posedge clk) disable iff (!resetn)
//     req |-> (rdy, ##[1:5] ack))
//     else `uvm_error("SVA", $sformatf(
//       "Error in req->ack. req=%b rdy=%b ack=%b", req, rdy, ack));
// endchecker: chkr
//
// req : INTENTION = assume (testbench-driven -> UVM sequence)
// rdy : INTENTION = obligation (DUT-driven -> real RTL, never randomized)
// ack : INTENTION = obligation (DUT-driven -> real RTL, never randomized)

`include "uvm_macros.svh"
import uvm_pkg::*;
```

```
// ------------------------------------------------------------------
// RTL DUT - a simple handshake implementation of req -> rdy, ##[1:5] ack
// ------------------------------------------------------------------
module dut (
  input  bit clk,
  input  bit resetn,
  input  bit req,
  output bit rdy,
  output bit ack
);
  // ...
endmodule: dut

// Same chkr - attached with a bind instead of an explicit instantiation,
// so the DUT source above never has to be touched.
bind dut chkr chkr_i (.clk(clk), .resetn(resetn), .req(req), .rdy(rdy), .ack(ack));

// ------------------------------------------------------------------
// Signal bundle the driver/monitor use to reach the DUT
// ------------------------------------------------------------------
interface dut_if (input bit clk);
  logic resetn;
  logic req;
  logic rdy;
  logic ack;
endinterface: dut_if

// ------------------------------------------------------------------
// UVM sequence item - the only thing randomized is req
// ------------------------------------------------------------------
class req_item extends uvm_sequence_item;
  rand bit req;
  constraint c_req_dist { req dist {1'b1 := 1, 1'b0 := 4}; }  // req high ~20%

  `uvm_object_utils(req_item)

  function new(string name = "req_item");
    super.new(name);
  endfunction
endclass: req_item

// ------------------------------------------------------------------
// UVM sequence - ~200 items, same order of magnitude as the SIM_7 loop
// ------------------------------------------------------------------
class req_seq extends uvm_sequence #(req_item);
  `uvm_object_utils(req_seq)

  function new(string name = "req_seq");
    super.new(name);
  endfunction

  task body();
    req_item item;
    repeat (200) begin
      item = req_item::type_id::create("item");
      start_item(item);
      if (!item.randomize())
        `uvm_error("SEQ", "req_item randomize failed")
      finish_item(item);
    end
  endtask
endclass: req_seq
```

```
// ------------------------------------------------------------------
// UVM driver - applies req to the DUT; rdy/ack are read back for logging
// only, never driven
// ------------------------------------------------------------------
class req_driver extends uvm_driver #(req_item);
  `uvm_component_utils(req_driver)
  virtual dut_if vif;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif))
      `uvm_fatal("DRV", "virtual interface not set")
  endfunction

  task run_phase(uvm_phase phase);
    req_item item;
    forever begin
      seq_item_port.get_next_item(item);
      @(posedge vif.clk);
      vif.req <= item.req;
      #1;
      `uvm_info("DRV", $sformatf("t=%0t req=%b rdy=%b ack=%b",
                $realtime, vif.req, vif.rdy, vif.ack), UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass: req_driver

// ------------------------------------------------------------------
// Env - sequencer + driver only; chkr is the checking element, so no
// separate scoreboard is needed for this sanity-level environment
// ------------------------------------------------------------------
class req_env extends uvm_env;
  `uvm_component_utils(req_env)
  uvm_sequencer #(req_item) sqr;
  req_driver                drv;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sqr = uvm_sequencer#(req_item)::type_id::create("sqr", this);
    drv = req_driver::type_id::create("drv", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass: req_env
```

```
// ------------------------------------------------------------------------
// Test - starts req_seq on the sequencer
// ------------------------------------------------------------------------
class req_test extends uvm_test;
  `uvm_component_utils(req_test)
  req_env env;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = req_env::type_id::create("env", this);
  endfunction

  task run_phase(uvm_phase phase);
    req_seq seq;
    phase.raise_objection(this);
    seq = req_seq::type_id::create("seq");
    seq.start(env.sqr);
    phase.drop_objection(this);
  endtask
endclass: req_test

// ------------------------------------------------------------------------
// Top level - clock, reset, DUT (+ chkr via bind), waveform, watchdog
// ------------------------------------------------------------------------
module uvm_tb;
  bit clk;
  initial forever #5 clk = !clk;

  dut_if vif (.clk(clk));

  dut dut1 (
    .clk    (clk),
    .resetn (vif.resetn),
    .req    (vif.req),
    .rdy    (vif.rdy),
    .ack    (vif.ack)
  );
  // chkr_i is live inside dut1 via the bind above - no separate
  // instantiation is needed here

  initial begin
    $dumpfile("uvm_tb.vcd");
    $dumpvars(0, uvm_tb);
  end

  // Reset sequence
  initial begin
    vif.resetn = 1;
    repeat (2) @(posedge clk);
    vif.resetn = 0;
    @(posedge clk);
    vif.resetn = 1;
  end

  // Timeout watchdog
  initial begin
    #15000;
    `uvm_fatal("TB", "TIMEOUT - simulation exceeded limit")
  end
```

```
  initial begin
    uvm_config_db#(virtual dut_if)::set(null, "*", "vif", vif);
    run_test("req_test");
  end
endmodule: uvm_tb
```

As in Stages 1 and 2, chkr itself is untouched; only the wiring around it changes. The bind statement is the common way to attach a checker once real RTL exists, because it keeps the assertion out of the DUT source entirely - the same checker module can be bound into a different DUT, or removed, without editing the design.

6. Comparing the Three Styles

Style	Stimulus Source	What It Answers	Limitation
Directed (SIM_5 deterministic sanity)	Hand-picked fixed values chosen to exercise one exact scenario	"Does the checker fire correctly on the case the requirement describes?"	Covers only the scenarios the engineer thought to write; does not sample the wider input space
Fast Random (SIM_6 / SIM_7 constrained-random)	randomize() with dist constraints, driven every cycle for ~200 cycles	"Does the checker still behave under combinations nobody wrote by hand?"	Samples the space rather than proving it; a rare combination can still be missed by chance
UVM-Integrated	UVM sequences/sequencers driving the same checker inputs, plus scoreboards and coverage groups	"Does this check plug into the project's shared reporting, regression, and coverage infrastructure?"	Heavier setup cost; still a simulation-based sample, not an exhaustive proof

7. When to Use Which Style

- Use directed alone when a checker is brand new and its logic has not yet been demonstrated to work on the textbook case - this is never skipped, per RAG-SVA's SIM_5 rule.
- Add fast random once the directed case passes and a wider, cheap sample of the input space is wanted before investing in a full UVM environment - useful for quick sanity on a standalone checker or a small DUT.
- Move to UVM-integrated once the checker needs to participate in shared regressions, coverage, and reporting alongside other IP or SoC-level checks, or once stimulus needs to come from a reusable sequence library rather than a one-off loop.
- Consider formal (outside the three styles in this paper) when the requirement is safety-critical or when directed and random sampling repeatedly fail to build confidence about corner cases - formal proves rather than samples.

8. Conclusion

RAG-SVA supports three testbench styles - directed, fast random, and UVM-integrated - because verification confidence is built in layers, not all at once. A directed test proves the checker's logic matches the requirement on the case everyone agrees on. A fast, constrained-random test then samples a much wider set of combinations cheaply, catching interactions no one thought to write by hand. A UVM-integrated environment finally gives the same, unmodified checker a home inside a project's shared stimulus, reporting, and coverage infrastructure. None of the three claims to be exhaustive on its own, and none needs to be rewritten to move to the next stage, because the assert/assume property logic - the actual verification intent - never changes between them.