2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 24-27, 2025

PSS Comes of Age: Runtime Behavioral Coverage, Methodology and More

Hillel Miller

accellera SYSTEMS INITIATIVE

Agenda

- What is PSS Runtime Behavioral Coverage?
- Runtime Behavioral Coverage flow
- Behavioral Coverage activity operators
- Covergroups in behavioral coverage
- Runtime Reactive Example
- Summary

PSS Runtime Behavioral Coverage – What is it?

- In PSS 2.0 we introduced Data Coverage:
  - Used to report data coverage
  - Like System Verilog cover-groups
  - Primary Application is at solve time
- In PSS 3.0 we introduced runtime behavioral coverage:
  - Used to report coverage of behaviors of actions
  - Primary application is for runtime coverage with behavior and data
  - Evaluated over a runtime trace of events
  - LRM formally specifies the expected coverage results given a coverage PSS model and a trace of events.
  - Usage of PSS Monitors to model activities required to observe to meet coverage goals
  - PSS Actions used to model activities for generating action traversal behaviors
  - PSS Monitors used to model activities for detecting action traversal behaviors
  - PSS Monitors have the look and feel of PSS Action . Making the adoption easier.

Goals of this presentation

- Assuming you have a good PSS knowledge
- Understand the flow and motivation for using behavioral runtime coverage
- Provide enough tools to make you LRM behavioral runtime coverage literate
- Challenge you on whether behavioral runtime coverage is something to be used on your next project

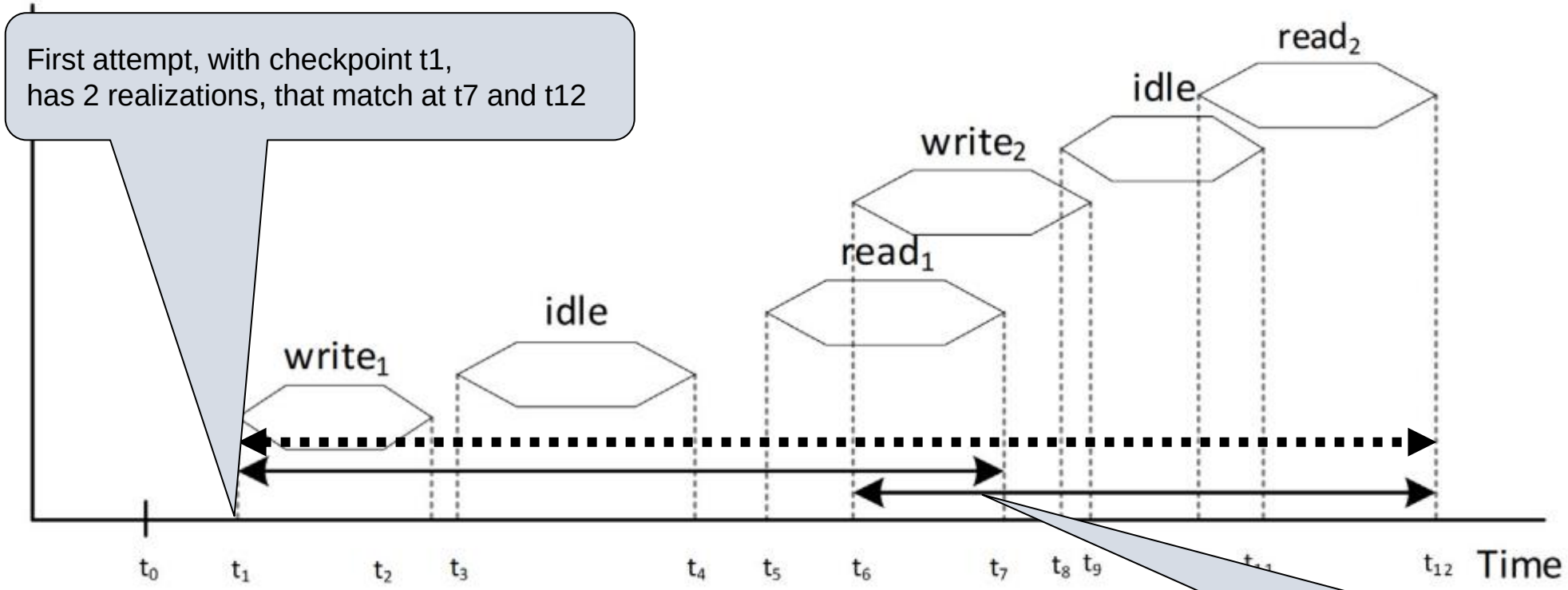
PSS Runtime Behavioral Coverage – Example

```
monitor m1 { write w; read r; activity { w; r; } }
c1: cover m1;
```

A *scenario checkpoint* is a time instant relative to which the scenario is observed.

An *attempt* is a scenario along with its checkpoint.

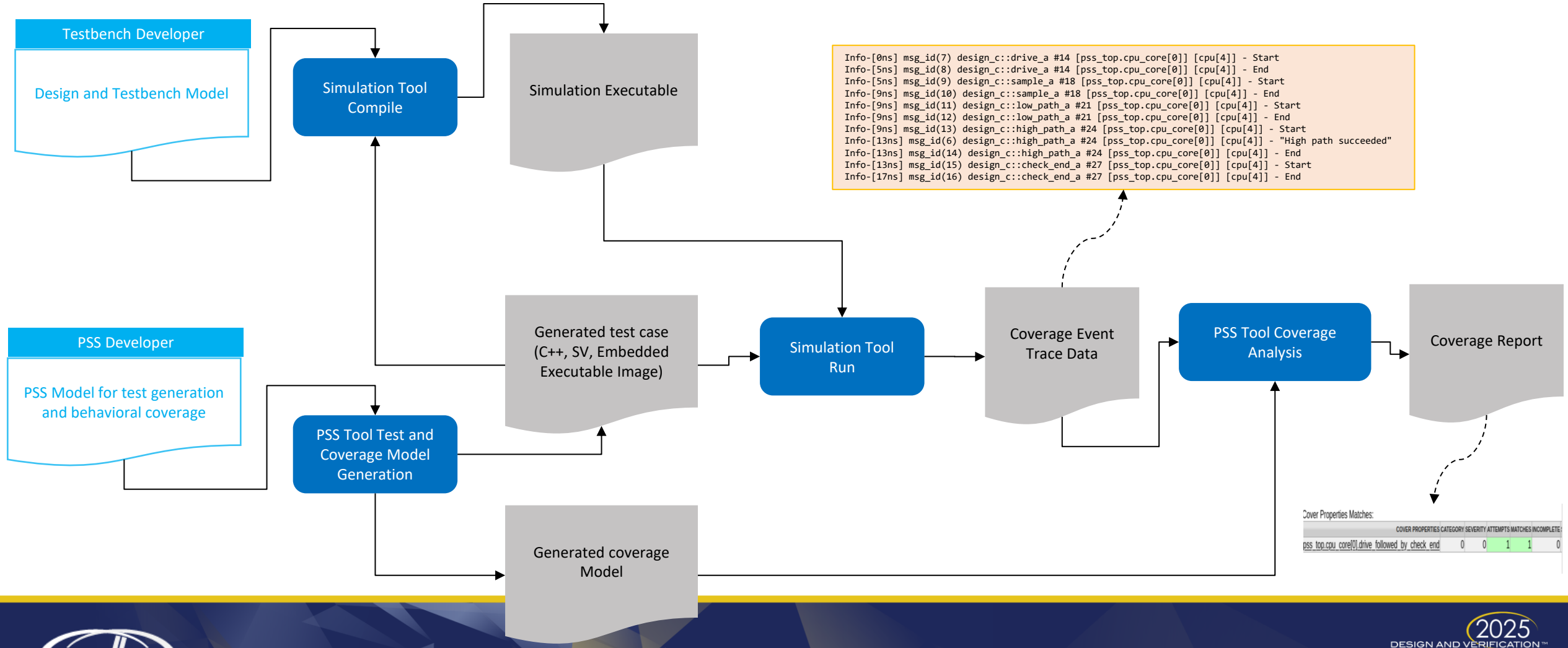
An *attempt scenario realization* is a set of action executions traversed by this attempt along with the mapping of action handles into specific action executions.



PSS Runtime Behavioral Coverage – Example

```
action read {}
action write {}
action idle {}
action send {}
action receive {}

monitor m1 { write w; read r; activity { w; r; } }
monitor m2 { activity { do write; select { do read; do send; }; do receive; } }
monitor m3 { activity { select { do write; do read; }; select { do send; do receive; }; } }

c1: cover m1;
c2: cover m3;
```

Runtime Behavioral Coverage Simulation Flow Example



PSS Runtime Behavioral Coverage – Motivation and Benefits

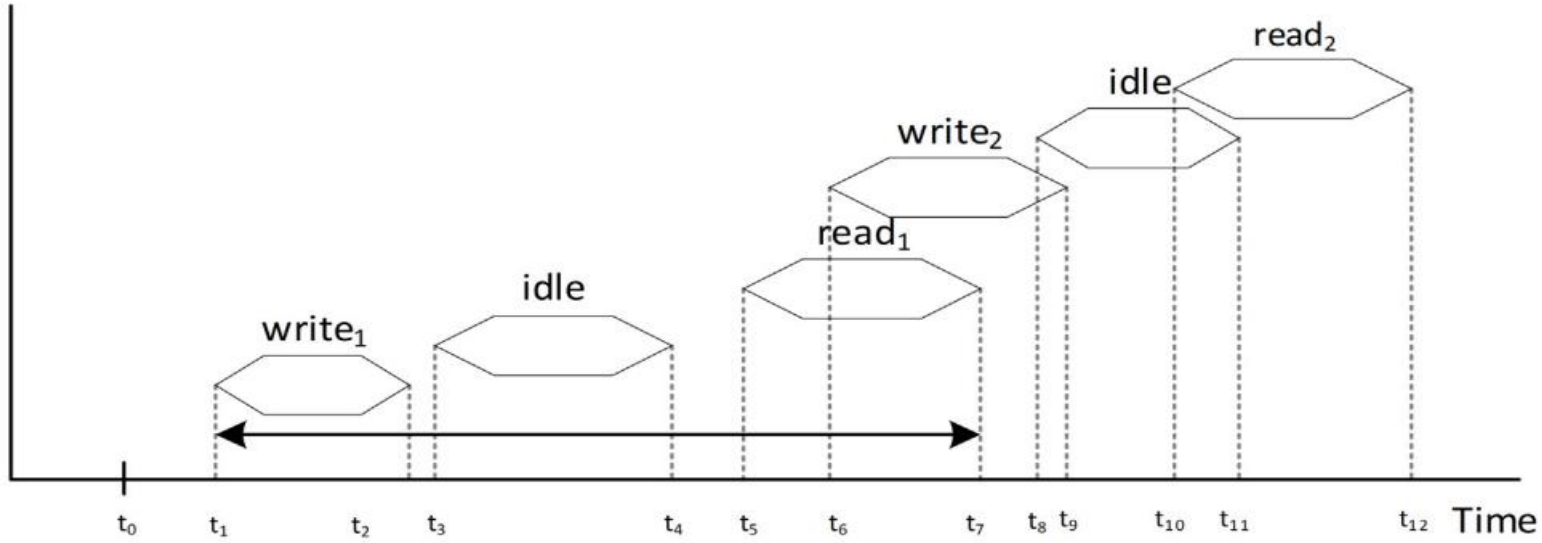
- Model monitors to detect behavioral patterns of action traversals
- Enables tools to report coverage of desired behavioral patterns
- Enables sampling behavioral patterns together with data combinations
- Desired overlapping behaviors can only be confirmed at runtime
- Portable across simulation, emulation, silicon, field … All that's needed is a trace for the required types
- Enables post processing flow, to sample coverage
- Enables sampling data updated at test runtime
- Modular modeling using monitors hierarchically, just like actions
- Usage of constraints to refine pattern detection based on data
- Usage of action like syntax to minimize learning new language (Actions are used for behavior generation. Monitors are used for behavior observation and detection.)
- Provides analysis information for incomplete runs

Monitor “sequence” activity operator

```
monitor m1 {
  activity {
     sequence {
          do write;
          do read;
     }
  }
}
```

Monitor `m1` defines a sequence of two action traversals: {`do write; do read;`}. Its subscenarios are `do write` and `do read`. First, the scenario `do write` is matched. Its realization is the $\texttt{write}_1$ action execution, and its end time is t_2. As a checkpoint of the second scenario `do read`, t_2 or a later time instant should be chosen. For the checkpoint t_2, the second scenario realization is $\{read_1\}$ action execution. For any checkpoint between t_2 and t_5 (the start point of $\{read_1\}$), the scenario realization will not change. For any checkpoint t, $t_5 < t \leq t_{10}$, the scenario realization is $\{read_2\}$ action execution. For any checkpoint $t > t_{10}$, there is no match. To summarize, scenario { `do write; do read;` } has two scenario realizations: $\{write_1, read_1\}$ and $\{write_1, read_2\}$, with the match points t_7 and t_{12}, correspondingly. Informally speaking, we choose the first execution of a `write` action at or after t_0, and at or after its endpoint (t_7), we choose any `read` action.

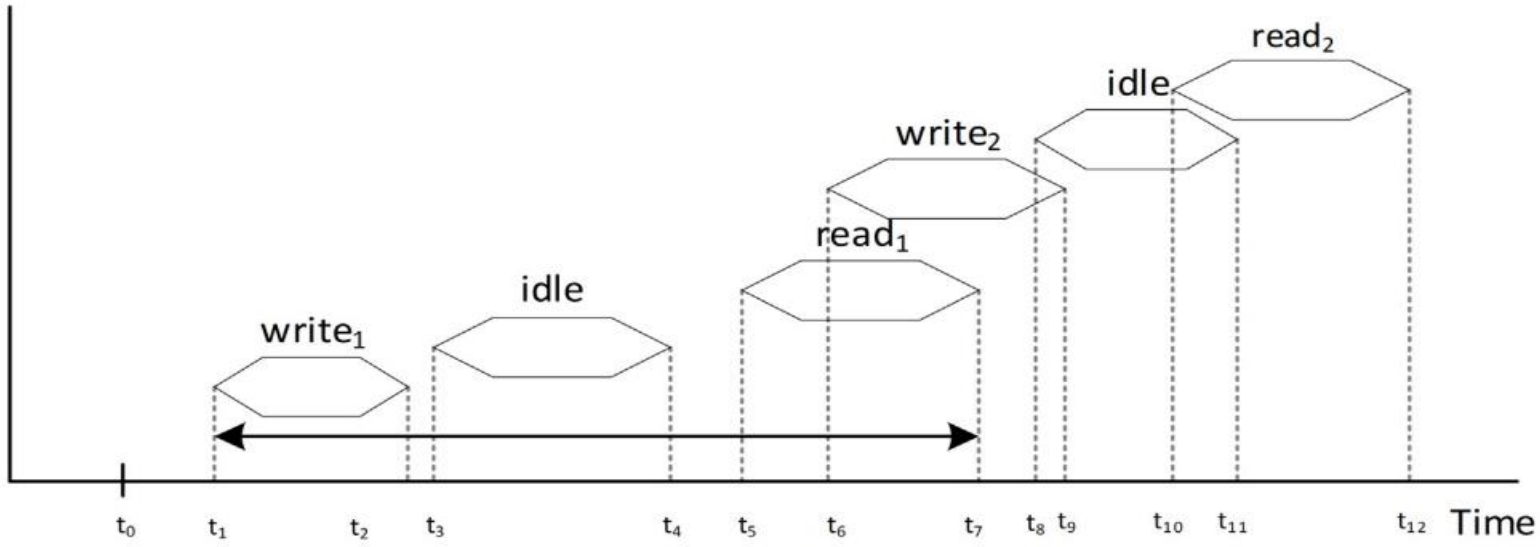
1



Monitor “concat” activity operator

The concatenation scenario defines an immediate consecutive matching of its sub-scenarios; the checkpoint of the next sub-scenario is the matching point of the previous one.

```
c1: cover { activity { concat { do write; do read; } } }
c2: cover { activity { sequence { do write; do read; } } }
```



Monitor inline constraints

```
c5: cover { activity { concat { do start; do write with core == 0; do read; } } }
c6: cover { activity { sequence { do start; do write with core == 0; do read; } } }
```

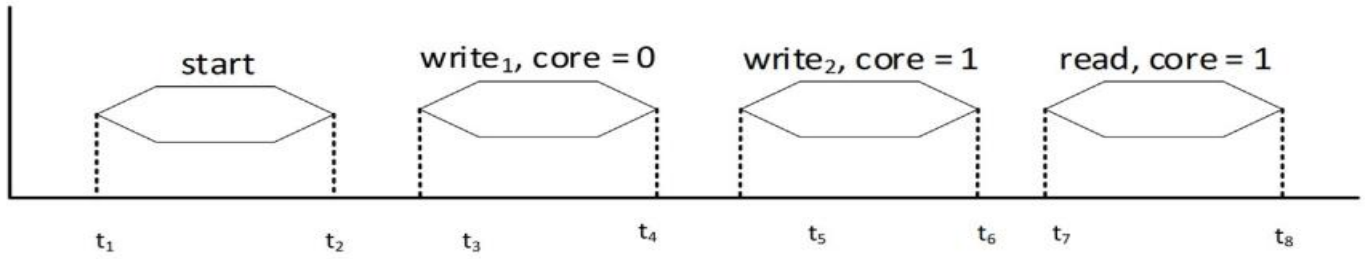


Figure 31—First alternative. Example 195

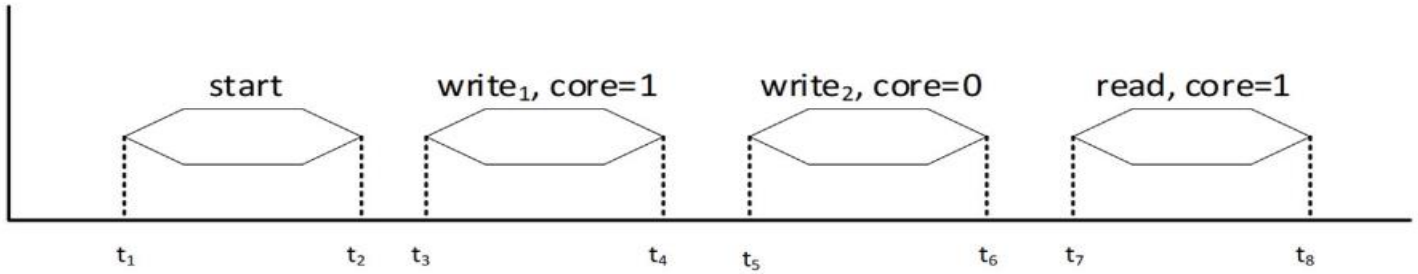


Figure 32—Second alternative. Example 195

- The top-level scenarios of c5 and c6 have the same realization for each trace: {start, write1, read} for the trace in Figure 31 and {start, write2, read} for the trace in Figure 32.
- The inline constraint in the concat scenario in cover statement c5 will match the first traversal of write that satisfies the constraint. In Figure 32, starting at t2, the match point of the start traversal, the subscenario “do write with core == 0” has the realization {write2}.

Monitor “schedule” operator

- The scheduling scenario defines execution of its sub-scenarios in any order, if scenario realizations of the member scenarios are not shared.
- There may be any overlaps or gaps between its member scenario spans.
- In the scheduling scenario, the sub-scenarios are matched from the checkpoint of the scheduling scenario or after it.
- The checkpoints of individual sub-scenarios are independent of each other.
- The realizations of scenario schedule { s1,…,sn } consist of member-wise realization unions of sub-scenarios s1,..., and sn, provided that these realizations of different scenarios are pairwise disjoint.
- For example, if set {a,b} is a realization of s1, set {c} is a realization of s2 and set {d,e,f} is a realization of s3, then set {a,b,c,d,e,f} is a realization of schedule { s1,s2,s3 }; here, a, b, c, d, e, and f are action executions

Monitor “schedule” operator

```
action read {}
action write {}
action send {}
action receive {}
monitor m1 { activity { schedule { sequence { do read; do write; }; sequence { do write; do send; } } }
monitor m2 { activity { schedule { sequence { do write; do send; }; sequence { do send; do receive; } } }
```

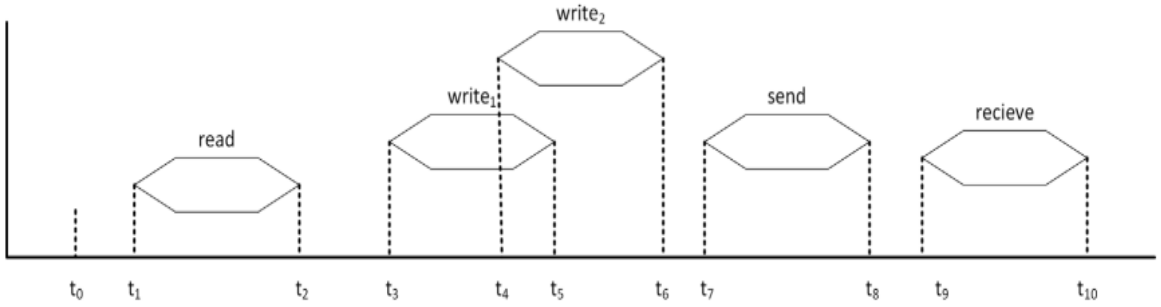


Figure 44—Scheduling scenario with common actions

- The first sub-scenario of the monitor m1 is sequence { do read; do write; }. It has two realizations: {read, write1} and {read, write2}.
- The second sub-scenario Is sequence { do write; do send; }, which also has two realizations: {write1, send} and {write2, send}.
- The realization of the top scenario is obtained either as a union of the first realization of the first sub-scenario and the second realization of the second sub-scenario or as a union of the second realization of the first sub-scenario and the first realization of the second sub-scenario.
- Both cases result in the same realization {read, write1, write2, send}.
- Other combinations of sub-scenario realizations cannot be united because they have a common element, either the first or the second write.
- The scenario of the monitor m2 has no match.
- Its first sub-scenario sequence { do write; do send; } has one realization: {write1, send}.
- Its second sub-scenario sequence { do send; do receive; } also has one realization: {send, receive}.
- Since both realizations intersect, the top-level scenario does not have any match.

Monitor overlap operator

```
action read {}
action write {}
monitor m { activity { overlap { do read; do write; } } }
```

- The overlapping scenario defines overlapping of its member sub-scenarios.
- The overlapping scenario meets the same conditions as the scheduling scenario and an additional condition.
- The additional condition is that there is a time instant where all its member scenarios are simultaneously active,

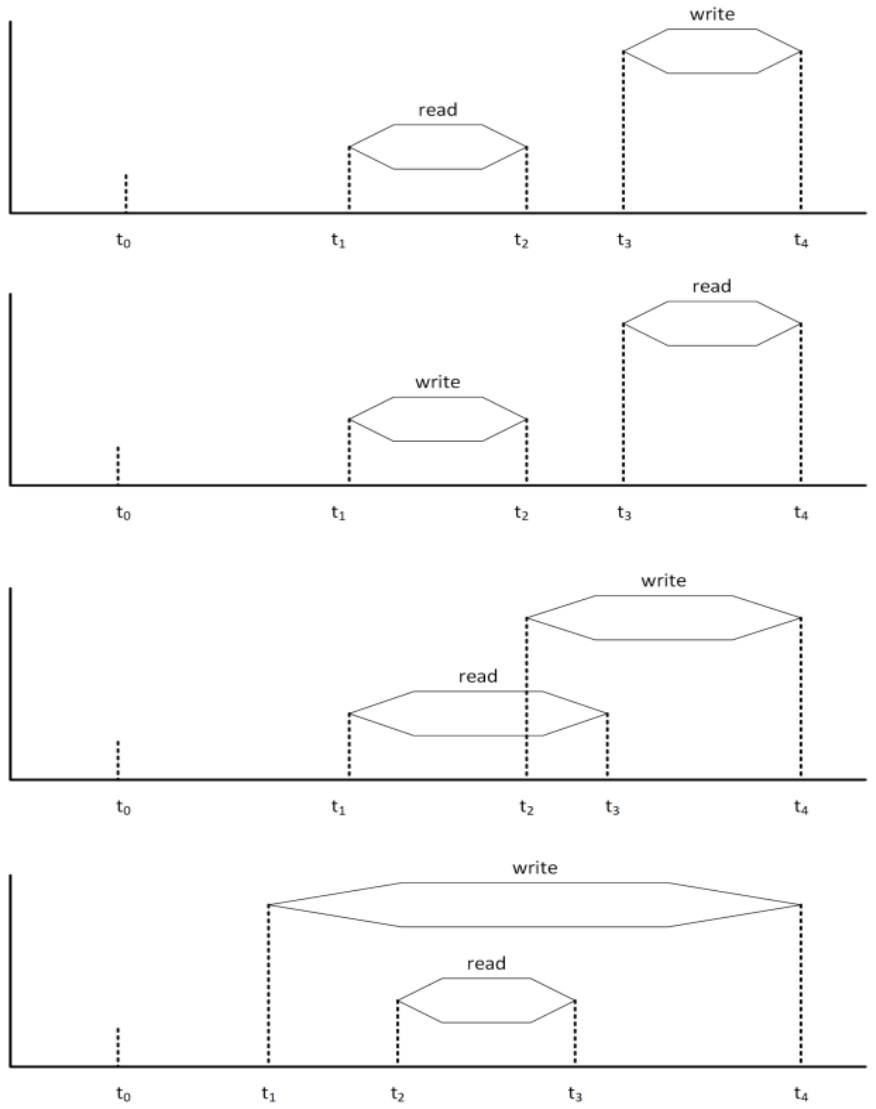
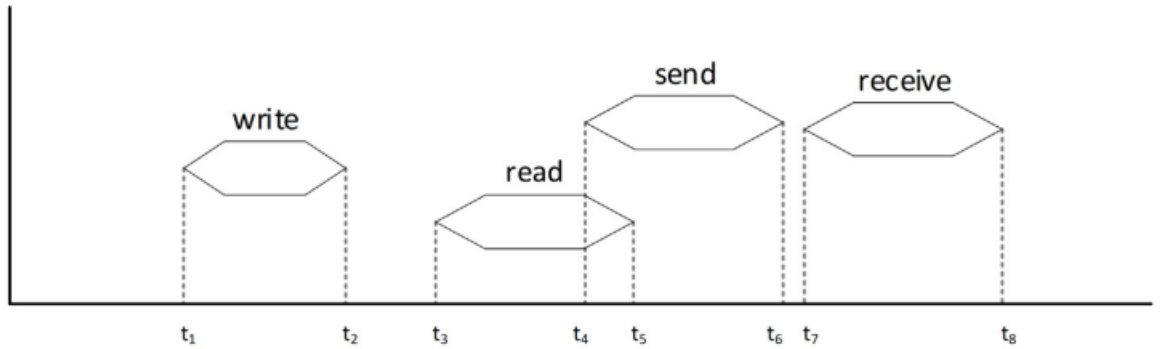


Figure 39—Overlapping scenario

Monitor select operator

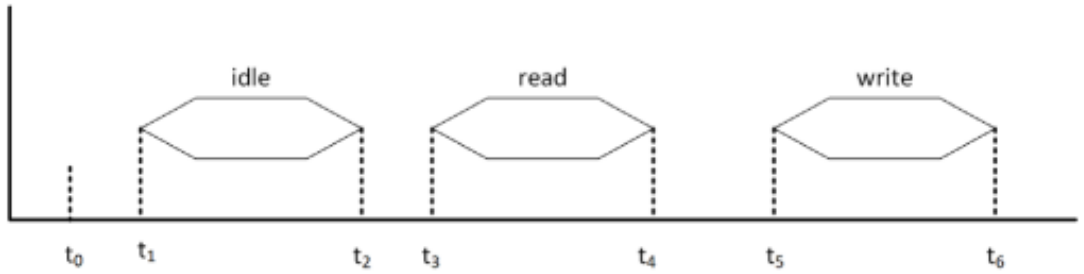
```
action read {}
action write {}
action idle {}
action send {}
action receive {}
c1: cover {
  activity {
    do write;
    select { do read; do send; };
    do receive;
  }
}
c2: cover {
  activity {
    select { do write; do read; };
    select { do send; do receive; };
  }
}
```



- Cover statement c1 has a successful attempt starting at t1 with two realizations: {write, read, receive} and {write, send, receive}.
- Cover statement c2 has two successful attempts, one starting at time t1 with two realizations {write, send} and {write, receive}, and the other has starting at time t3 with the realization {read, receive}.

Hierarchical Monitors for composing coverage Models

```
action idle {}
action read {}
action write {}
monitor rw { activity { do read; do write; } }
monitor irw { idle i; read r; write w; activity { i; r; w; } }
monitor irw1 { idle i; activity { i; do rw; } }
monitor irw2 { idle i; rw m; activity { i; m; } }
c1: cover irw;
c2: cover irw1
c3: cover irw2
```



- Cover statement c1, c2 and c3 are equivalent.
- Monitors irw, irw1, and irw2 are equivalent.
- Consider the monitor matching in the trace for the checkpoint t0.
- The monitor irw specifies a sequential scenario and its only realization is {idle, read, write}.
- Monitor irw1 defines a sequential scenario whose sub-scenarios are the traversal of action i with the realization {idle} and the traversal of an anonymous monitor of type rw.
- The monitor type rw, in its turn, defines a sequential scenario, and its checkpoint is t1 or later so that the realization of monitor irw1 is {idle, read, write}.
- The monitor irw2 differs from the monitor irw1 only in that it traverses the monitor of type rw using its handle m, and of course, has the same realization {idle, read, write}.

Cover Reports

Coverage Property	Attempts	Matches
Pss_top.cpu_core[0]. drive_followed_by_check_end	2	2

Covergroups in Monitors

```
enum locked_e { LOCKED, UNLOCKED };
enum write_mode_e { WRITE_BACK, WRITE_THRU };
action read { rand locked_e lock_mode; }
action write { rand write_mode_e write_mode; }
c: cover {
  write w;
  read r;
  activity { w; r; }
  covergroup {
    cpw: coverpoint w.write_mode;
    cpr: coverpoint r.lock_mode;
    wXr: cross cpw, cpr;
  } cg;
}
```

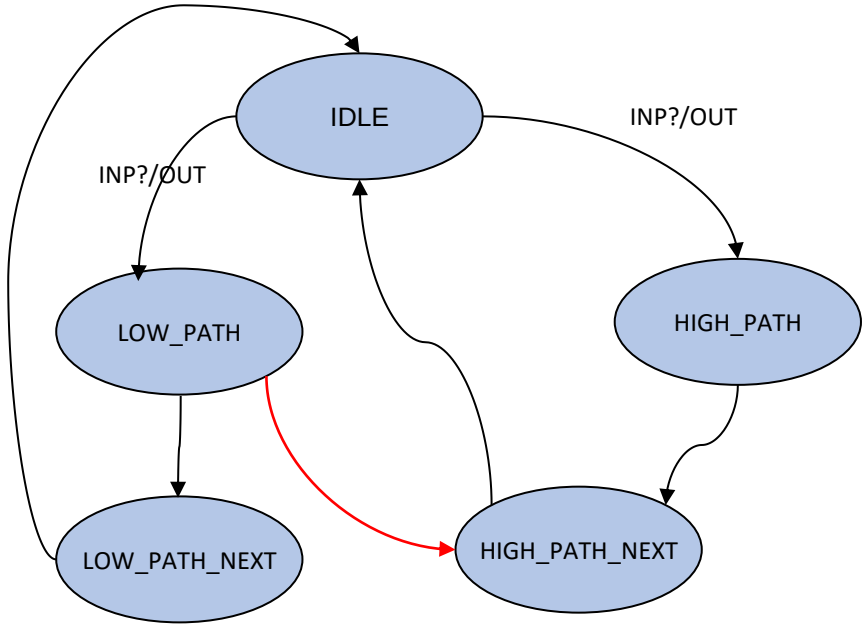
- Covergroups may be defined and instantiated in monitors and cover statements to collect data coverage along the scenario defined by the monitor.
- A monitor covergroup is sampled at the first match of the attempts of a cover statement where the monitor is traversed (directly or not).
- The sampling is done according to the action handle mapping associated with a first match scenario realization.
- If there are several first match scenario realizations, any realization may be selected for sampling by the implementation.

Verification flow example

- Example shows how the addition of Runtime Behavioral Coverage has bought PSS closer to satisfy all needs of a verification engineer to achieve his verification requirements.
- Example is chosen to be reactive to show how a PSS model can deal with behaviors occurring at Runtime
- The parts that are shown
  - Stimulus generation
  - Import functions to collect runtime behavior from Testbench interface
  - Checkers to check state machine behavior
  - Runtime Behavioral coverage to make sure specific paths were covered in state machine

Reactive Runtime Example Design

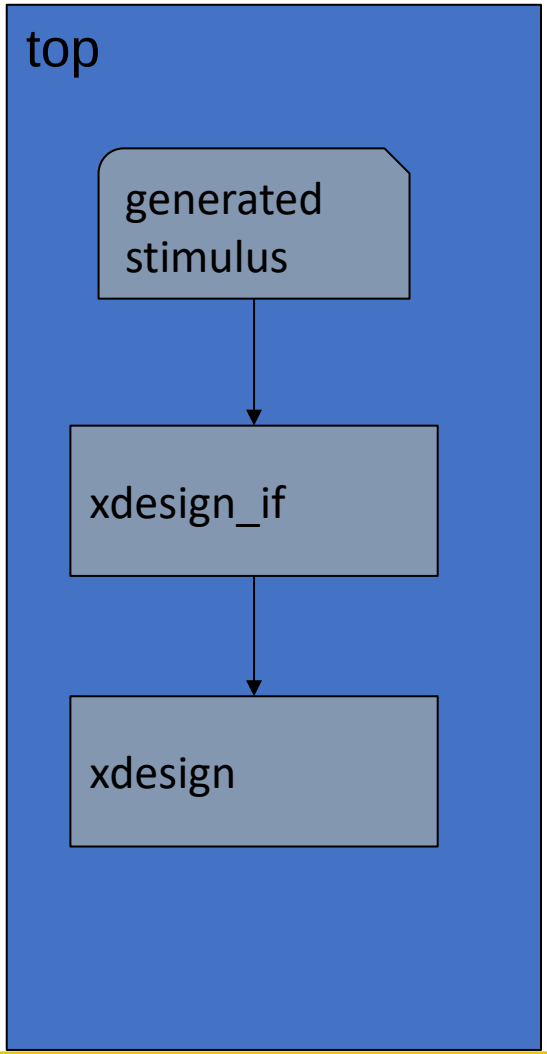
```
typedef enum bit[2:0] {IDLE=3'b000, LOW_PATH=3'b001, HIGH_PATH=3'b010, LOW_PATH_NEXT=3'b101,
HIGH_PATH_NEXT=3'b110} state_e;
module xdesign  (
   input resetn,
   input clk,
   input [7:0] inp,
   input goto_next_state,
   output reg [7:0] out,
   output state_e state
   );
   reg                                  prev_goto_next_state;
   wire               is_next_state_transition;
   assign is_next_state_transition = !prev_goto_next_state && goto_next_state;
   always @(posedge clk)
     if  (!resetn) begin
                  state <= IDLE;
                  out <= 0;
                  prev_goto_next_state <= 0;
     end
     else begin
                  prev_goto_next_state <= goto_next_state;
                  case (state)
                     IDLE : begin
                        if (is_next_state_transition) begin
                                    if (inp < 8'h80) begin
                                       state <= LOW_PATH;
                                       out <= inp + 1;
                                    end else
                                      begin
                                         state <= HIGH_PATH;
                                         out <= inp - 1;
                                      end
                        end
                      end // case: begin...
                    LOW_PATH :
                      if (is_next_state_transition)
                        state <= LOW_PATH_NEXT;
                    HIGH_PATH :
                      if (is_next_state_transition)
                        state <= HIGH_PATH_NEXT;
                    LOW_PATH_NEXT :
                      if (is_next_state_transition)
                        state <= IDLE;
                    HIGH_PATH_NEXT :
                      if (is_next_state_transition)
                        state <= IDLE;
                  endcase // case (state)
     end // else: !if(!resetn)
endmodule // design
```



Reactive Runtime Example Testbench

```
interface xdesign_if (
   input resetn,
   input clk,
   output reg [7:0] design_inp,
   output reg goto_next_state,
   input [7:0] design_out,
   input [2:0] design_state
);

   task automatic sample_out_imp(output [7:0] out);
      out = design_out;
   endtask // sample_out_imp

   task automatic sample_state_imp(output [2:0] state);
      state = design_state;
   endtask // sample_out_imp

   task transition_state_imp();
      goto_next_state <= 0;
      @(posedge clk);
      goto_next_state <= 1;
      @(posedge clk);
   endtask // transition_state_imp

   task drive_imp(input [7:0] inp);
       design_inp <= inp;
       @(posedge clk);
      transition_state_imp();

   endtask // drive_imp

endinterface // design_if
```

```
module top;
   reg clk;
   reg resetn;

   initial begin
      $fsdbDumpfile("top");
      $fsdbDumpvars(0, top);
      resetn <= 0;
      repeat(1) @(posedge clk);
      resetn <= 1;

   end
   initial clk = 0;
   always #1 clk = ~clk;

   wire [7:0] inp;
   wire       goto_next_state;
   wire [7:0] out;
   wire [2:0] state;

   xdesign xd(
                        .clk(clk),
                        .resetn(resetn),
                        .inp(inp),
                        .goto_next_state(goto_next_state),
                        .out(out),
                        .state(state)
                        );

   xdesign_if xdif(
                        .clk(clk),
                        .resetn(resetn),
                        .design_inp(inp),
                        .goto_next_state(goto_next_state),
                        .design_out(out),
                        .design_state(state)
                        );
   initial begin
      pss__shared__pkg::_if = top.xdif;
      pss__pkg::pss_run_solution();
      $finish;
   end
endmodule // top
```



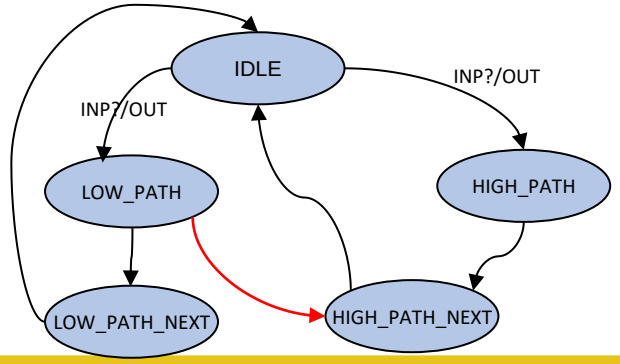
Reactive Runtime Example PSS Generation Model

```
action test_state_machine_a {
   activity {
     select {
       do drive_inp_a with {inp == 0x10;};
       do drive_inp_a with {inp == 0x90;};
     }
     do sample_path_taken_a;
     parallel {
       do check_low_path_a;
       do check_high_path_a;
     }
     do drive_final_a;
   }
 }
```

1

```
action drive_inp_a : base_a {
   rand bit[8] in [0x10, 0x90] inp;
   exec body {
      comp.drive(inp);
   }
 }
```

2




```
action sample_path_taken_a : base_a {
      output design_b design_out;
      exec body {
        state_e xstate;
        bit [8] out;
        xstate = comp.sample_state();
        out = comp.sample_out();
        if (xstate == LOW_PATH) {
          design_out.is_low_path = true;
          design_out.is_high_path = false;
        } else if (xstate == HIGH_PATH){
          design_out.is_low_path = false;
          design_out.is_high_path = true;
        }
        comp.transition_state();
      }
 }
```

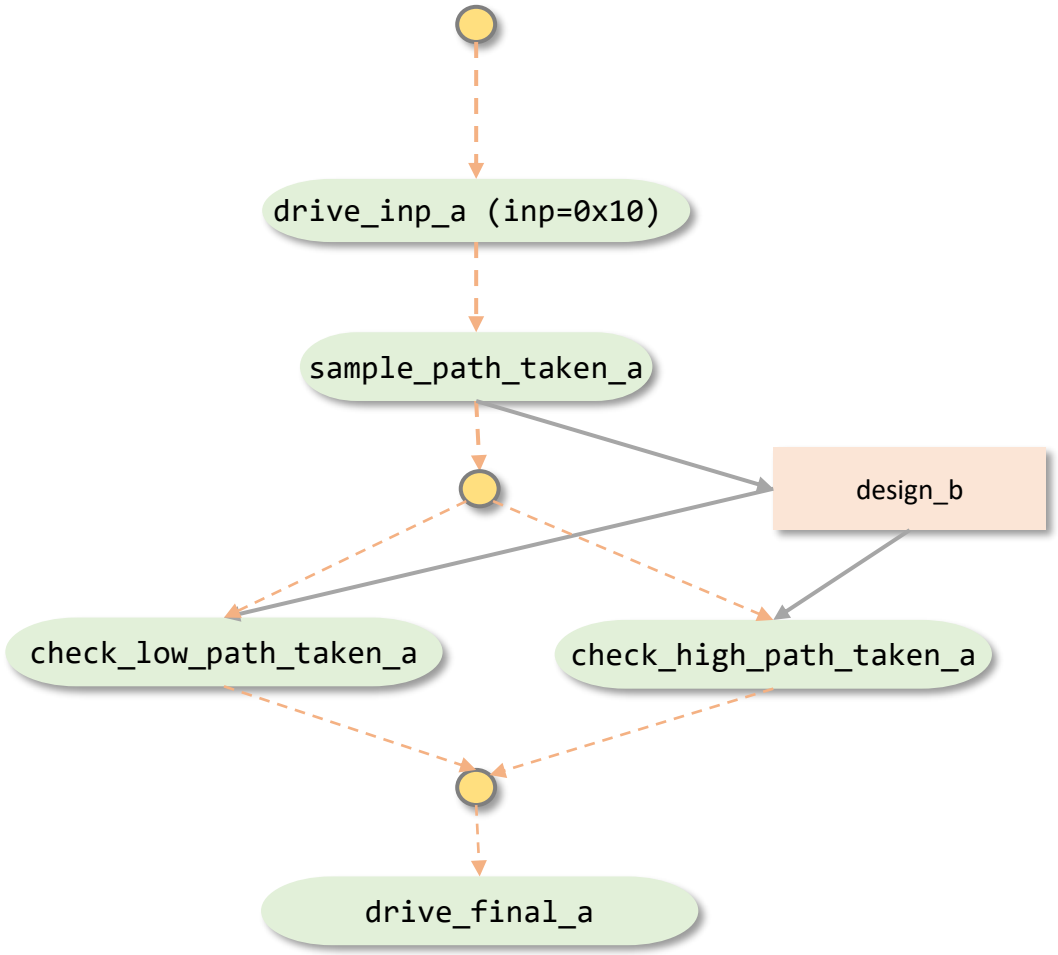
3

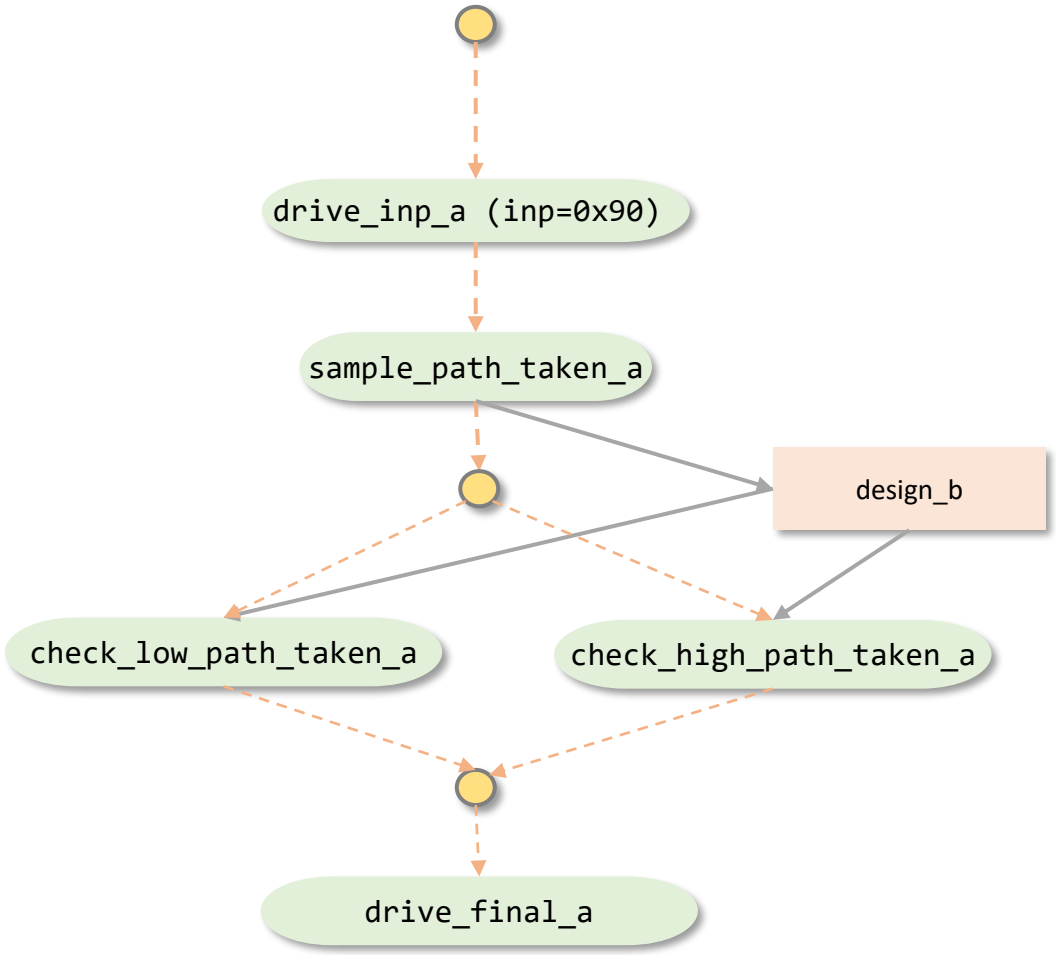
```
action drive_final_a : base_a {
      exec body {
        comp.transition_state();
      }
   }
```

4

```
action check_low_path_a : base_a {
  input design_b design_in;
  exec body {
    state_e xstate;
    if (design_in.is_low_path) {
      xstate = comp.sample_state();
      comp.transition_state();
      if (xstate != LOW_PATH_NEXT)
        message(LOW, "Low path failed");
      else
        message(LOW, "Low path succeeded");
    }
  }
}

action check_high_path_a : base_a {
  input design_b design_in;
  exec body {
    state_e xstate;
    if (design_in.is_high_path) {
      xstate = comp.sample_state();
      comp.transition_state();
      if (xstate != HIGH_PATH_NEXT)
        message(LOW, "High path failed");
      else
        message(LOW, "High path succeeded");
    }
  }
}
```

Reactive Runtime Example – Generated Test Cases

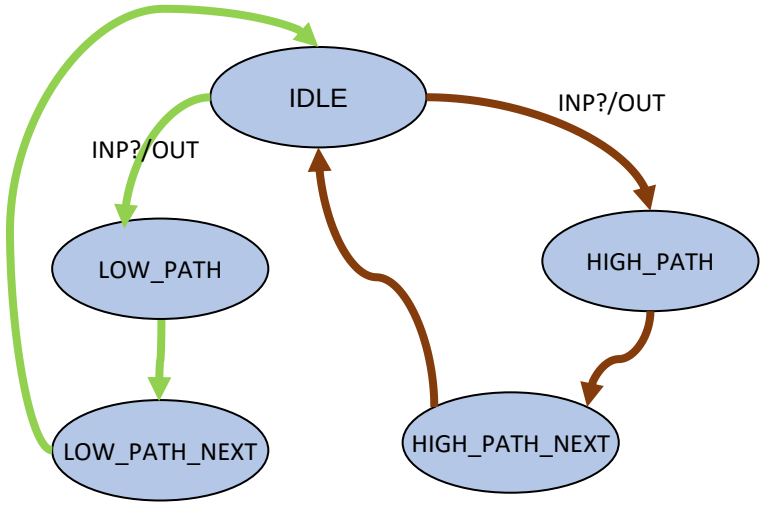




Reactive Runtime Example PSS Coverage Model

```
drive_followed_by_check_end: cover {
  drive_inp_a dr;
  activity {
    dr;
    do drive_final_a;
  }
  covergroup {
    drive_value: coverpoint dr.inp {
           bins a[] = [0x10, 0x90];
         };
  } cg;
 }
}
```

```
Info-[0ns] msg_id(7) design_c::drive_inp_a #14 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[5ns] msg_id(8) design_c::drive_inp_a #14 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[5ns] msg_id(9) design_c::monitor_path_taken_a #18 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[9ns] msg_id(10) design_c::monitor_path_taken_a #18 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[9ns] msg_id(11) design_c::drive_low_path_a #21 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[9ns] msg_id(13) design_c::drive_high_path_a #24 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[9ns] msg_id(14) design_c::drive_high_path_a #24 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[13ns] msg_id(4) design_c::drive_low_path_a #21 [pss_top.cpu_core[0]] [cpu[4]] - "Low path succeeded"
Info-[13ns] msg_id(12) design_c::drive_low_path_a #21 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[13ns] msg_id(15) design_c::drive_final_a #27 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[17ns] msg_id(16) design_c::drive_final_a #27 [pss_top.cpu_core[0]] [cpu[4]] - End
```




```
Info-[0ns] msg_id(7) design_c::drive_inp_a #16 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[5ns] msg_id(8) design_c::drive_inp_a #16 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[5ns] msg_id(9) design_c::monitor_path_taken_a #18 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[9ns] msg_id(10) design_c::monitor_path_taken_a #18 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[9ns] msg_id(11) design_c::drive_low_path_a #21 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[9ns] msg_id(12) design_c::drive_low_path_a #21 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[9ns] msg_id(13) design_c::drive_high_path_a #24 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[13ns] msg_id(6) design_c::drive_high_path_a #24 [pss_top.cpu_core[0]] [cpu[4]] - "High path succeeded"
Info-[13ns] msg_id(14) design_c::drive_high_path_a #24 [pss_top.cpu_core[0]] [cpu[4]] - End
Info-[13ns] msg_id(15) design_c::drive_final_a #27 [pss_top.cpu_core[0]] [cpu[4]] - Start
Info-[17ns] msg_id(16) design_c::drive_final_a #27 [pss_top.cpu_core[0]] [cpu[4]] - End
```

Reactive Runtime Example Coverage Report

Coverage Property	Attempts	Matches
Pss_top.cpu_core[0]. drive_followed_by_check_end	2	2

User Defined Bins for drive_value

Bins

NAME	COUNT	AT LEAST
a_10	1	1
a_90	1	1

Summary

- Provided understanding of the flow and motivation for using behavioral runtime coverage, motivations included:
- Provided enough tools to make you LRM behavioral runtime coverage literate
  - Went over text mentioned in LRM.
- Challenged you on whether behavioral runtime coverage is something to be used on your next project
  - Are you convinced Runtime Behavioral Coverage can help you? Why?