COMS30026 Design Verification

# High-level Verification
with specman and e

## Part 2: Advanced Features

Kerstin Eder

Trustworthy Systems Laboratory

https://www.bristol.ac.uk/engineering/research/trustworthy-systems-laboratory/

University of BRISTOL

Department of COMPUTER SCIENCE

# Randomized Test Generation needs…

**… repeatability:**

Same testbench version + same test
+ same random seed
= same stimulus data.

Is this all? The testbench evolves over time!

# Randomized Test Generation needs…

## Repeatability: ✓

Same testbench version + same test

+ same random seed

= same stimulus data.

## Random stability:

- Changes to the testbench should not affect **orthogonal** aspects!
  - Packet data structure:

```
struct packet {
   ...
   payload: list of byte;
   ...};
```

# Randomized Test Generation needs…

## Repeatability:

Same testbench version + same test
+ same random seed
= same stimulus data.

## Random stability:

- Changes to the testbench should not affect **orthogonal** aspects!
  - Packet data structure with new `interrupted` field:

```
struct packet {
  ...
  payload: list of byte;
  interrupted: bool;
  ...};
```

With same seed we should get the same `payload` data!

# Packing: Driving Stimulus into the DUV

`pack()` function:

- `pack()` is a Specman Elite system function.
  - `pack(`option: pack option`,` item: exp`, ...): list of bit`
  - Each item is a legal "e" expression that is a scalar or a compound data item, such as a struct, field, list, or variable.

# Packing: Driving Stimulus into the DUV

pack() function:

- pack() is a Specman Elite system function.
  - pack(option: pack option, item: exp, ...): list of bit
  - Each item is a legal “e” expression that is a scalar or a compound data item, such as a struct, field, list, or variable.
- Converts a higher-level data structure to the bit stream required by the DUV during simulation.

```
input_stream = pack(packing.high, opcode, op1, op2);
         cmd = pack(packing.high, opcode);
        data = pack(packing.high, op1); ...
```

- pack options are: packing.high, packing.low or NULL
  - packing.high: 1st item at MSB position in the bit stream
  - packing.low: 1st item at LSB position in the bit stream
  - NULL: Use global default - set initially to packing.low.

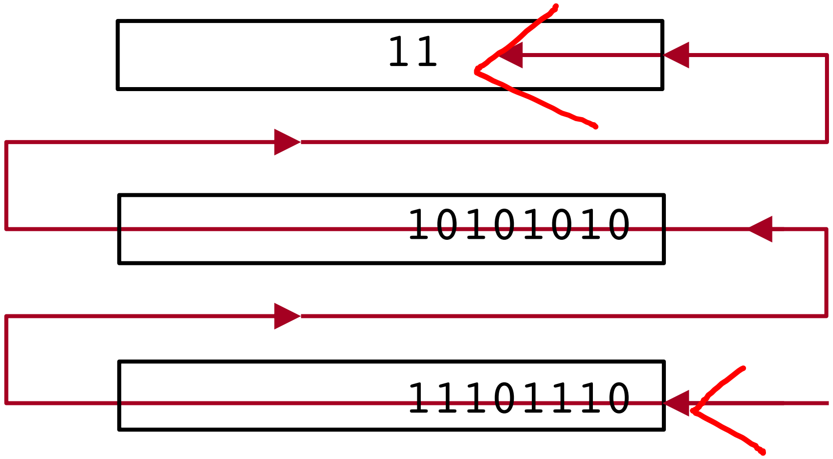
# Packing High

```
struct packet {
   addr: uint;
   data: list of uint;
};
```

packing.high: 1st item at MSB

```
input_stream = pack(packing.high,addr, data);
```

```
   packet.addr = 2'b11;

   packet.data[0] = 0xaa;

   packet.data[1] = 0xee;
```



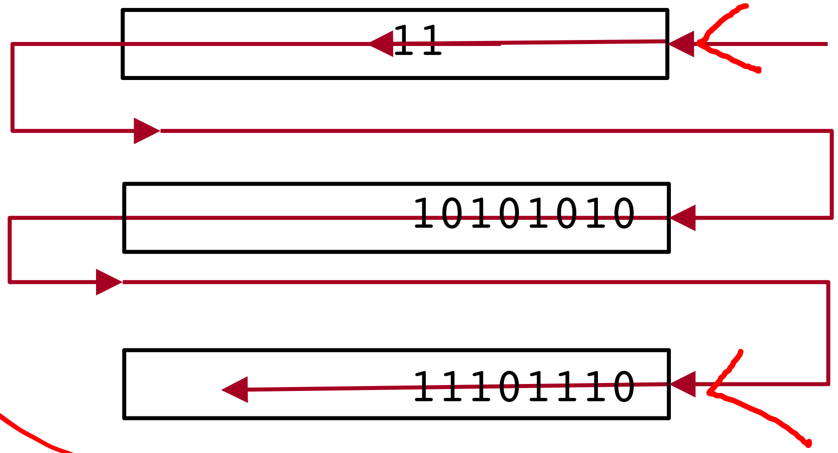
```
                 17.............................................0
   input_stream = 11 10101010 11101110
```

# Packing Low

```
struct packet {
  addr: uint;
  data: list of uint;
};
```

`packing.low`: 1st item at LSB

```
input_stream = pack(packing.low,addr, data);

  packet.addr = 2'b11;

  packet.data[0] = 0xaa;

  packet.data[1] = 0xee;
```



```
                 17..........................................0
  input_stream = 11101110 10101010 11
```

# Fields

`[!][%]` field-name`[:`type`]` `[[`min-val`..`max-val`]]` `[((`bits `|` bytes`):`num`)]`

**!** Denotes an ungenerated field.

% Denotes a physical field.

- The type for the field can be any scalar type, string, struct, or list.
- (bits | bytes: num) specifies the width of the field in bits or bytes.

```
type NetworkType: [IP=0x0800, ARP=0x8060] (bits: 16);
        struct header {
            %address: uint (bits: 48);
            %length: uint [0 .. 31];
        };
        struct packet {
            hdr_type: NetworkType;
            %hdr: header;
            is_legal: bool;
            !counter: uint;
        };
```

- The order of fields in the struct is important!
  - It is the packing order for the physical fields in the struct.

# Physical Fields

- Marked with %.
- Physical fields are packed when the struct is packed.
- Used for fields that represent data that will be sent to HDL design in the simulator.
- If no range is specified, width of field is determined by field's type.
- If the field's type does not have a known width, you must use (bits | bytes: num) syntax to define the width.
  - (Important for packing!)

**Non-physical fields** are called virtual fields.

- They are not packed automatically when the struct is packed.
  - (They can be packed individually if needed.)

# Ungenerated Fields

- Marked with **!**
  - Values for these fields are not auto generated.
  - **Useful for fields that:**
    - Are explicitly assigned values during verification.
    - Must contain values whose computation is too complicated to be expressed with constraints.

```
struct packet {
    addr: uint;
    payload: list of byte;
    !parity: bool;

    compute-even-parity(data: list of byte): bool is empty;

};
```

# Initialisation of Ungenerated Fields

Ungenerated fields are assigned a **default initial value:**

- 0 for scalars, NULL for structs and empty list for lists.
- Ungenerated fields whose value is from a range (e.g. [20..30]) **get initialized to the first value in the range.**
- If the field is a struct it won't be allocated and none of the fields in it will be generated.

# Limitations of e's AOP Implementation

- (Too) Many things can be extended!
  - So more discipline and foresight of the testbench structure are required.

# Limitations of e's AOP Implementation

- (Too) Many things can be extended!
  - So more discipline and foresight of the testbench structure are required.
- Fields in a struct can only be appended.
  - Fields are physically appended to existing fields in a struct.
  - Might create a problem when packing, wrt the packing order!
    - But items to pack can be listed individually to overcome this shortfall, i.e. the order of the fields in the item list when calling `pack` does not need to match the order in which the fields have been listed/declared in the struct.

# Limitations of e's AOP Implementation

- (Too) Many things can be extended!
  - So discipline and foresight of the testbench structure are required.
- Fields in a struct can only be appended:
  - Fields are physically appended to existing fields in a struct.
  - Might create a problem when packing (wrt packing order)!
    - But items to pack can be listed individually to overcome this shortfall, i.e. the order of the fields in the item list when calling `pack` does not need to match the order in which the fields have been listed/declared in the struct.
- Variance control fields: Extensions can only be specified for a single value of a control field.
  - But we can use the following trick!

# Extensions via *variance control fields* can only be specified for a single value of the control field!

– Example: Extension to an instruction struct (for calc_1 design):

```
type opcode_t : [  NOP, ADD, SUB, INV, INV1, SHL, SHR ] (bits : 4);
      struct instruction_s {
         %cmd_in : opcode_t;
         %din1   : uint (bits:32);
         %din2   : uint (bits:32);
         !resp   : uint (bits:2);
         !dout   : uint (bits:32);
         check_response(ins : instruction_s) is empty;

      }; // struct instruction_s

      extend instruction_s {
         is_a_shift : bool;
         keep is_a_shift == cmd_in in [SHL, SHR];

         when is_a_shift instruction_s {
          // Common extension to SHL and SHR goes here.
          ...
         }
      }
```

Extensions via *variance control fields* can only be specified for a single value of the control field!

- To get around this, introduce an additional *virtual* field.
- This field controls the common extensions.
- Example: Extension to an instruction struct (for calc_1 design):

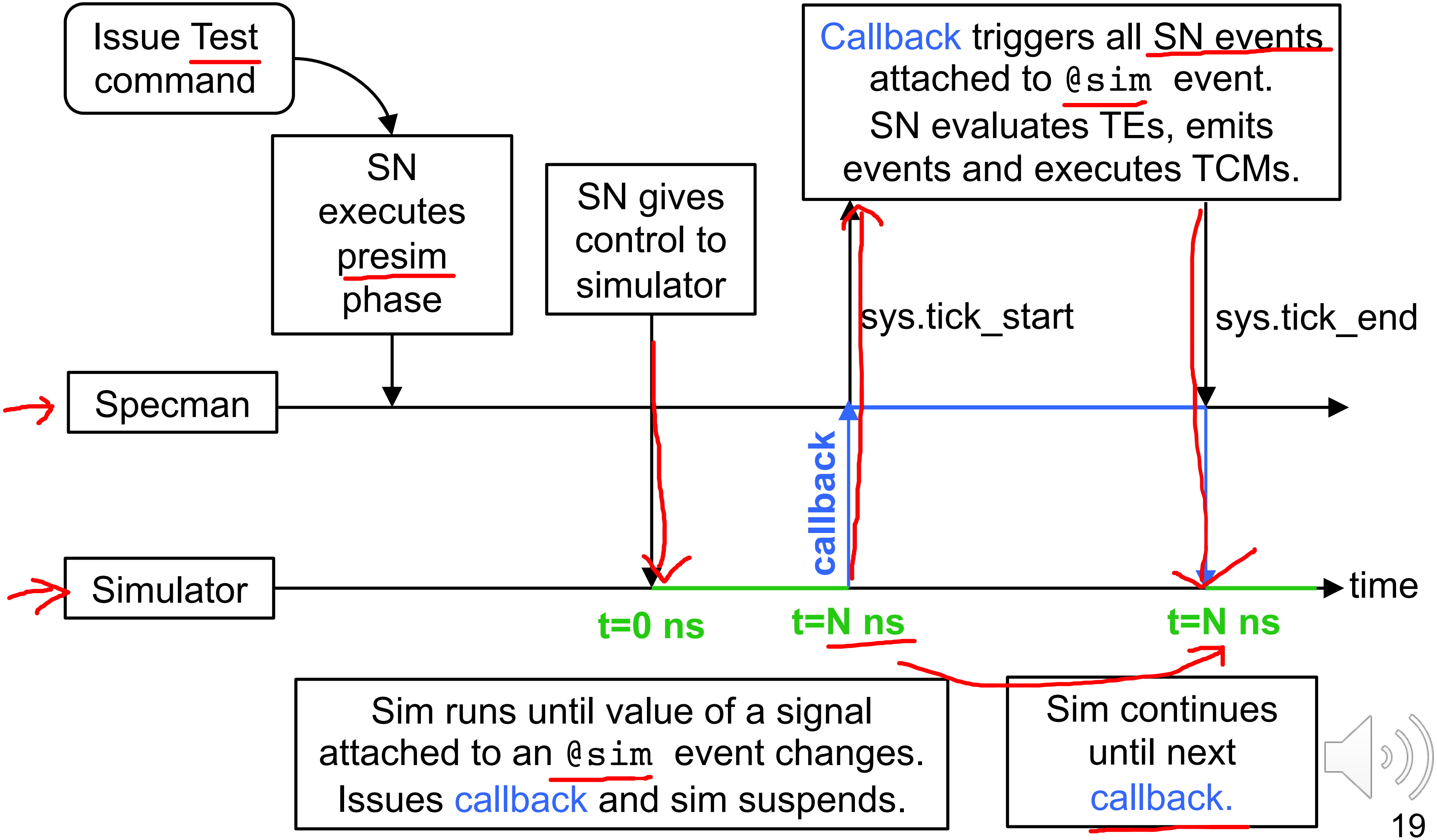
```
type opcode_t : [  NOP, ADD, SUB, INV, INV1, SHL, SHR ] (bits : 4);
      struct instruction_s {
         %cmd_in : opcode_t;
         %din1   : uint (bits:32);
         %din2   : uint (bits:32);
         !resp   : uint (bits:2);
         !dout   : uint (bits:32);
         check_response(ins : instruction_s) is empty;

      }; // struct instruction_s

      extend instruction_s {
         is_a_shift : bool;
         keep is_a_shift == cmd_in in [SHL, SHR];

         when is_a_shift instruction_s {
          // Common extension to SHL and SHR goes here.
          ...
         }
      }
```

# Limitations of e's AOP Implementation

- (Too) Many things can be extended!
  - So discipline and foresight of the testbench structure are required.
- Fields in a struct can only be appended:
  - Fields are physically appended to existing fields in a struct.
  - Might create a problem when packing (wrt packing order)!
- Variance control fields: Extensions can only be specified for a single value of a control field.

  **Example:**
  - Instructions **`SHL`** and **`SHR`** have a common feature.
  - We'd need to specify / code this for each (attracts higher maintenance). But we can use the trick from previous slide!
- Methods can only be appended, prepended or replaced.
- Aspects are order-dependent (on loading).

# Synch between SN and Simulator



Issue Test command

SN executes presim phase

SN gives control to simulator

Callback triggers all SN events attached to `@sim` event. SN evaluates TEs, emits events and executes TCMs.

sys.tick_start

sys.tick_end

Specman

Simulator

callback

time

t=0 ns

t=N ns

t=N ns

Sim runs until value of a signal attached to an `@sim` event changes. Issues callback and sim suspends.

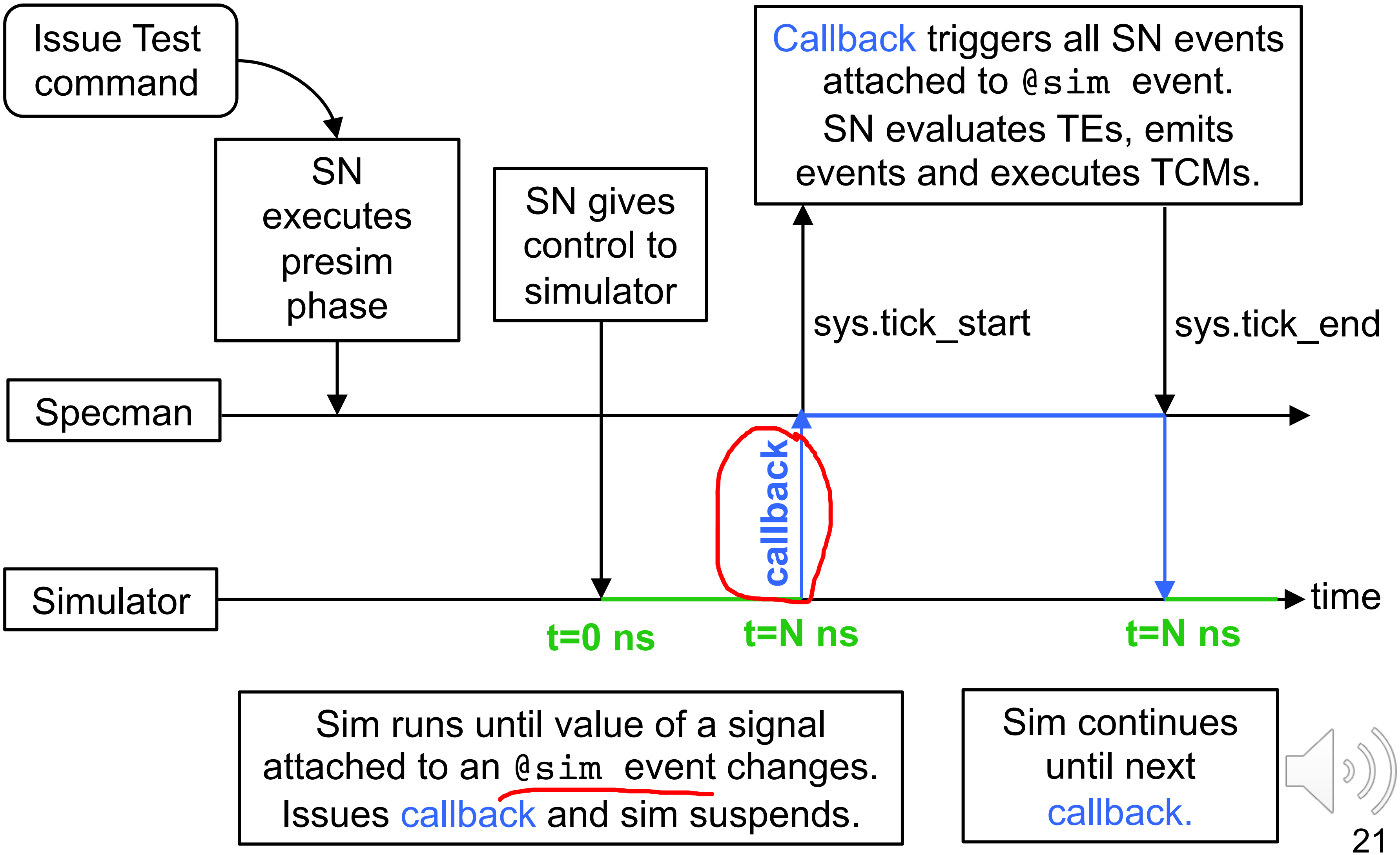
Sim continues until next callback.

# SN Predefined Event: **@sim**

```
event clk is rise (clk_p$) @sim;
```

- `@sim` is a special sampling event.
- `@sim` occurs at any ***simulator callback.***

# Synch between SN and Simulator



Issue Test command

SN executes presim phase

SN gives control to simulator

Callback triggers all SN events attached to @sim event.
SN evaluates TEs, emits events and executes TCMs.

Sim runs until value of a signal attached to an @sim event changes.
Issues callback and sim suspends.

Sim continues until next callback.

# SN Predefined Event: `@sim`

```
event clk is rise (clk_p$) @sim;
```

- `@sim` is a special sampling event.
- `@sim` occurs at any ***simulator callback.***
  - Expression must be an HDL signal path in the simulated model.
- Signal does not have to be a clock.
  - No restriction for signal to be periodic or synchronous.
- Heavy use of `@sim` events might slow down simulation!
  - Clock signal can also be emitted from “e” code and driven into DUV. (But usually more efficient to generate clock in HDL.)
- When not running with a simulator attached to SN, use `@sys.any.`

# Events in SN

- Events are struct members.
- Events are used to synchronize with the DUV or to debug a test.

## Automatic emission of events:

```
<'
    extend driver_s {
        event clk is fall(clk_p$) @sim;
        event resp is change(out_resp1_p$)@clk;
    };
'>
```

# Events in SN

- Events are struct members.
- Events are used to synchronize with the DUV or to debug a test.

## Explicit emission of event:

```
<'
  extend driver_s {
     collect_response(cmd : command_s) @clk is also {
        emit cmd.cmd_complete;
     };
  };
’>
```

# Advanced Techniques: SN temporal checking

## SN Temporal Language

- Capture behaviour over time for synchronization with DUV, functional coverage and protocol checking.
- Language consists of:
  - temporal expressions (TEs)
  - temporal operators
- Use **`event`** struct members to define occurrences of events during a sim run
- Use **`expect`** struct members for checking temporal behaviour
- PSL/Sugar and SVA compatible expressions.

# Temporal Expressions in “e”

- Each TE is associated with a sampling event.
- Sampling event indicates when the TE should be evaluated by SN.

- Syntax examples:

```
true(boolean-exp)@sample-event
rise/fall/change(expression)@sample-event
```

# Temporal Checking Methodology

1. Capture important DUV temporal behaviour with events and TEs.
2. Use **expect** struct members to declare temporal checks.

```
expect TE else dut_error(string);
```

**Example temporal checks:**

```
expect @req => {[..4];@ack} @clk
else dut_error("Acknowledge did not follow
               request within 5 clock cycles.");

expect @buffer_full => eventually @int @clk
else dut_error("Buffer full, but interrupt did not
               occur.");
```

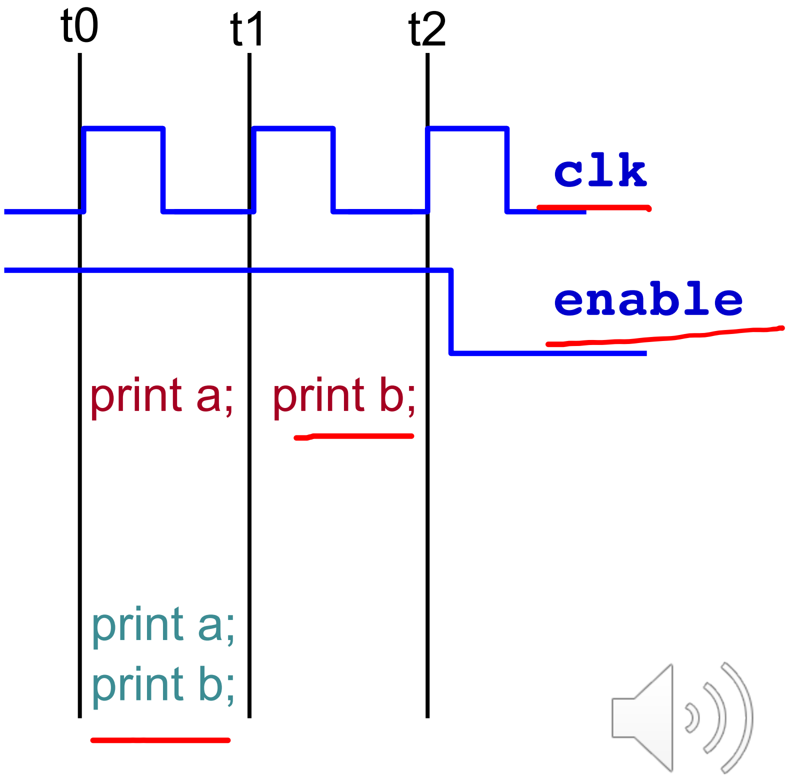
**Remember,** **eventually** means sometime before the end of simulation!

# Conforming to Stimulus Protocol

- Need be able to react to state of DUV during simulation!
  - clock, signal changes, sequences of events
- “e” language provides **wait** (till next cycle) and **sync** actions which allow to pause procedural code until the key event occurs.

```
print a;
wait true(enable_p$==1)@clk;
print b;
```

```
print a;
sync true(enable_p$==1)@clk;
print b;
```



# Methods with a Notion of Time

**TCMs - Time Consuming Methods**

- Depend on sampling event.
- Can be executed over several simulation cycles.

```
collect_response(cmd : command_s) @clk is {
   wait @resp; -- wait for the response
   cmd.resp = out_resp1_p$;
   cmd.dout = out_data1_p$;
}; // collect_response
```

- Implicit *synchronization action* at beginning of TCM.

# Methods with a Notion of Time

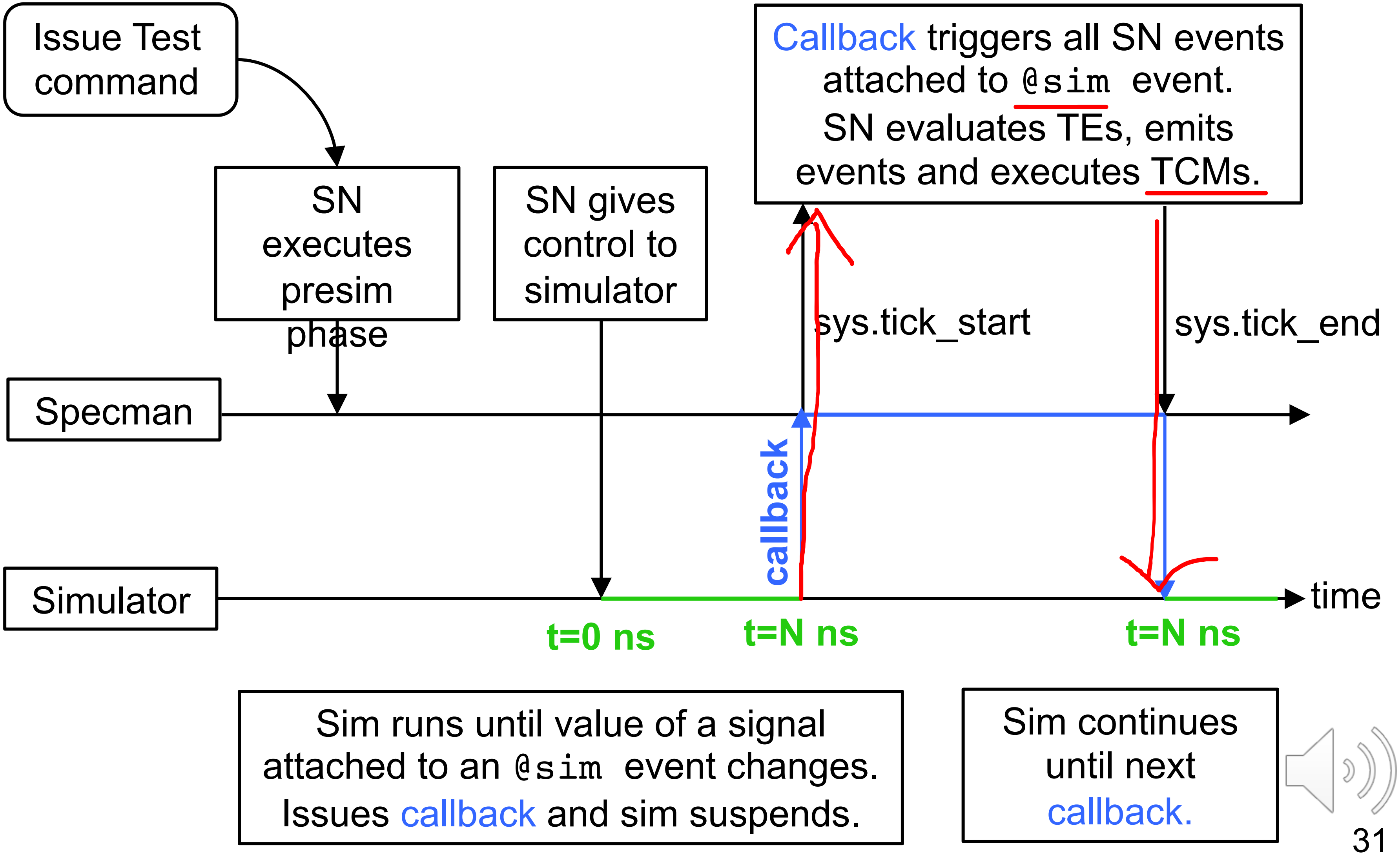
## TCMs - Time Consuming Methods

- TCM must be called or started to execute.

```
run() is also {
   start drive();        // spawn
}; // run
```

- Non-TCMs can't **call** TCMs because they have no notion of time.
- Instead, TCMs can (only) be **started** (using **`start`**) from a non-TCM!

# Synch between SN and Simulator



Issue Test command

SN executes presim phase

SN gives control to simulator

Callback triggers all SN events attached to `@sim` event.
SN evaluates TEs, emits events and executes TCMs.

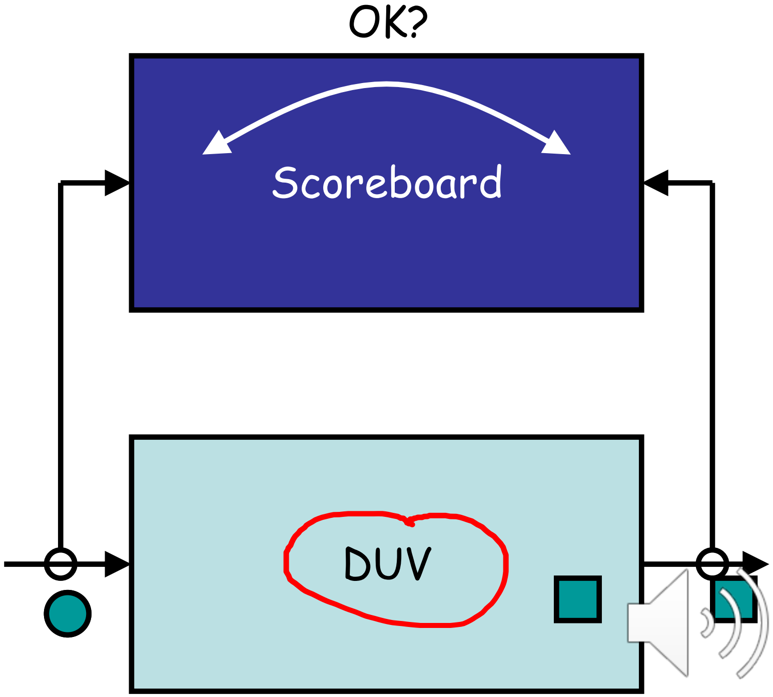
Sim runs until value of a signal attached to an `@sim` event changes. Issues callback and sim suspends.

Sim continues until next callback.

# Advanced Checking:

## Scoreboarding in e

(this refers back to the lecture on checking)



# Scoreboarding in e - 1

Assume: The DUV does not change the order of packets.

- Hence, the first packet on the scoreboard has to match the received packet.

# Scoreboarding in e - 1

Assume: The DUV does not change the order of packets.

- Hence, the first packet on the scoreboard has to match the received packet.

```
import packet_s;
unit scoreboard {
  !expected_packets : list of packet_s;
  add_packet(p_in : packet_s) is {
    expected_packets.add(p_in);
  };
};
```

# Scoreboarding in e - 1

Assume: The DUV does not change the order of packets.

- Hence, the first packet on the scoreboard has to match the received packet.

```
import packet_s;
unit scoreboard {
  !expected_packets : list of packet_s;
  add_packet(p_in : packet_s) is {
    expected_packets.add(p_in);
  };
  check_packet(p_out : packet_s) is {
    var diff : list of string;
     -- Compare physical fields of first packet on scb with p_out.
     -- Report up to 10 differences.
    diff = deep_compare_physical(expected_packets[0], p_out, 10);
    check that (diff.is_empty())
           else dut_error(‘‘Packet not found on scoreboard.’’,diff);
    -- If match was successful, continue.
    out(‘‘Found received packet on scoreboard.’’);
    expected_packets.delete(0);
  };
};
```

# Scoreboarding in e - 2

## Recording a packet on the scoreboard:

Extend driver such that

- When packet is driven into DUV call **`add_packet`** method of scoreboard.
  - Current packet is copied to scoreboard.
- It is useful to define an event that indicates when packet is being driven.

# Scoreboarding in e - 2

## Recording a packet on the scoreboard:

Extend driver such that

- When packet is driven into DUV call **`add_packet`** method of scoreboard.
  - Current packet is copied to scoreboard.
- It is useful to define an event that indicates when packet is being driven.

## Checking for a packet on the scoreboard:

Extend receiver such that

- When a packet was received from DUV call **`check_packet`**.
  - Try to find the matching packet on scoreboard.
- It is useful to define an event that indicates when a packet is being received.

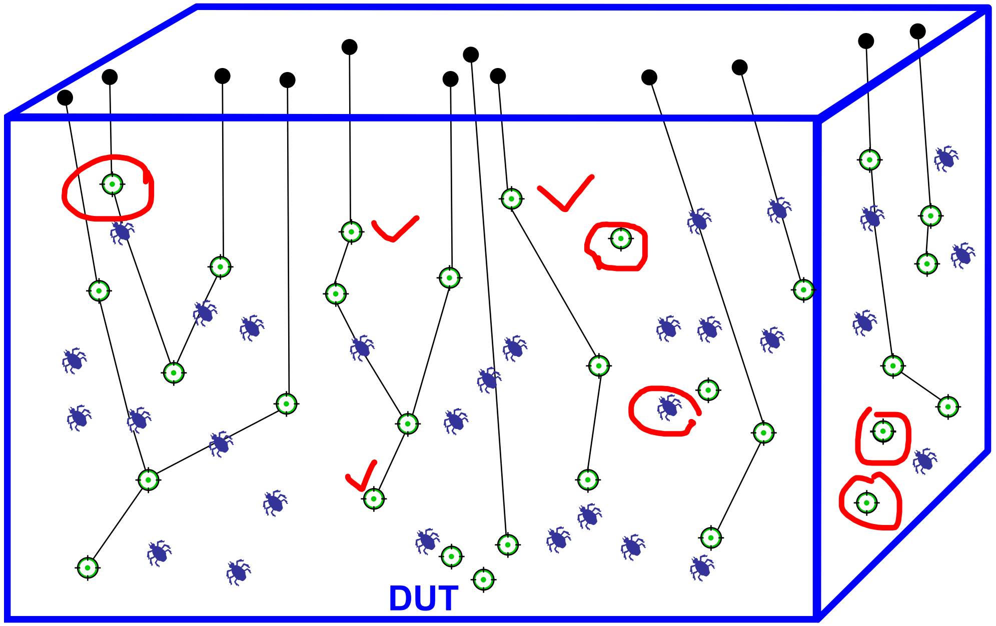
# High-level Verification

- Aim:
  - Raise level of abstraction, enable automation and, thus, enhance productivity.
- Strategy
  - Putting Coverage, Stimuli Generation and Checking together:

**The Coverage-Driven Verification Environment**

# Traditional Approach: Directed Testing

Verification engineers set goals (shown in **green** below) and write directed test for each item in the Verification Plan; these directed tests are then executed:



Redo if design changes

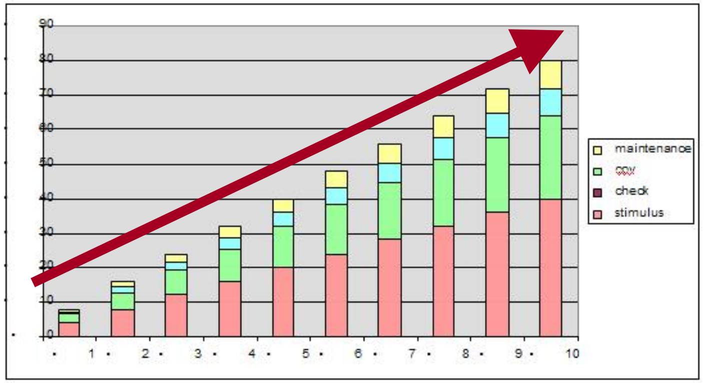
~~Automation~~ Significant manual effort to write all the tests

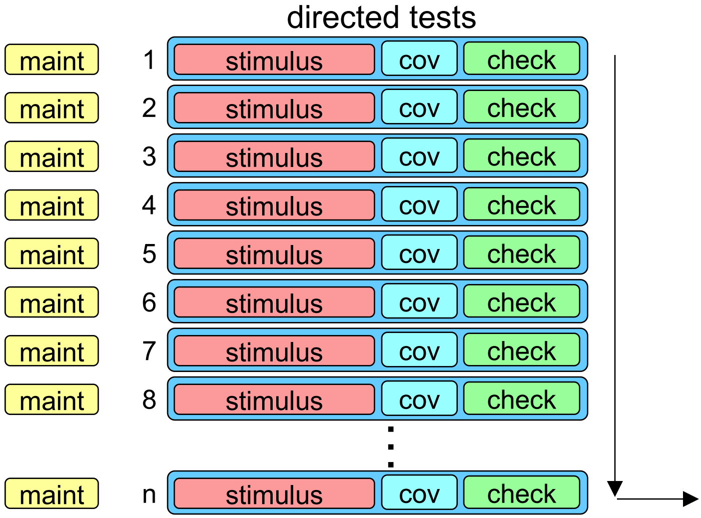
~~Automation~~ Work required to verify each goal was reached

~~Completeness~~ Poor coverage of non-goal scenarios
… especially the cases that you didn't "think of"

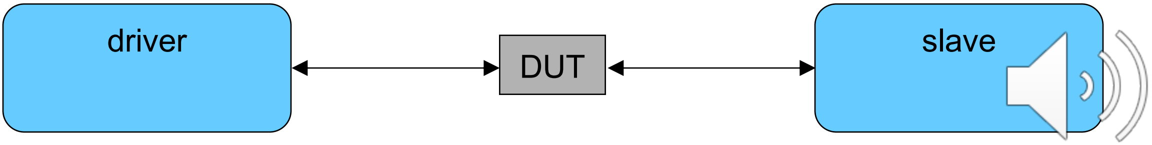
# Directed Test Environment

- Composition of directed tests
  - Directed tests contain more than just stimulus.
  - Checks are embedded into the tests to verify correct behavior.
  - The passing of each test is the indicator that a functionality has been exercised.
- Reusability and maintenance
  - Tests can become quite complex, making it difficult to understand the intent of what functionality is being verified.
  - Since the checking is distributed throughout the test suite, it is a lot of maintenance to keep checks updated.
  - It is usually difficult or impossible to reuse the tests across projects or from module to system level.
- **The more tests you have the more effort is required to develop and maintain them.**



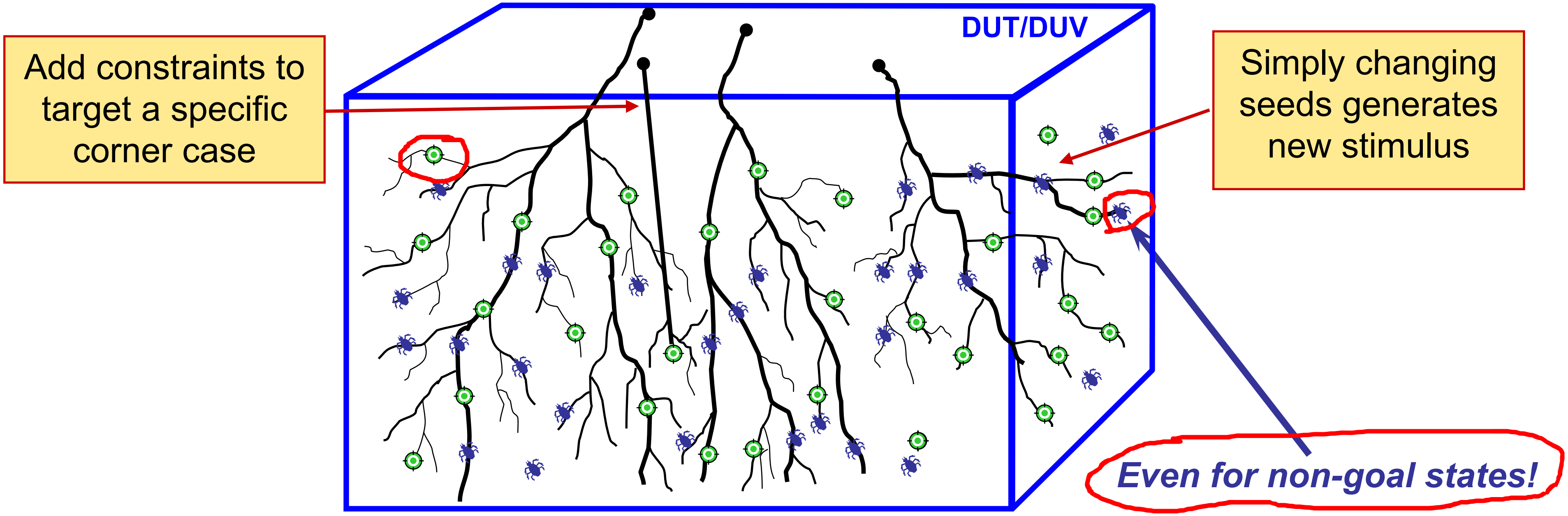


Directed test approach



# Coverage-Driven Verification Methodology

Focuses on reaching **goal areas** (*versus execution of test lists*):



Add constraints to target a specific corner case

Simply changing seeds generates new stimulus

*Even for non-goal states!*

**Constrained-random stimulus generation** explores goal areas (& beyond).

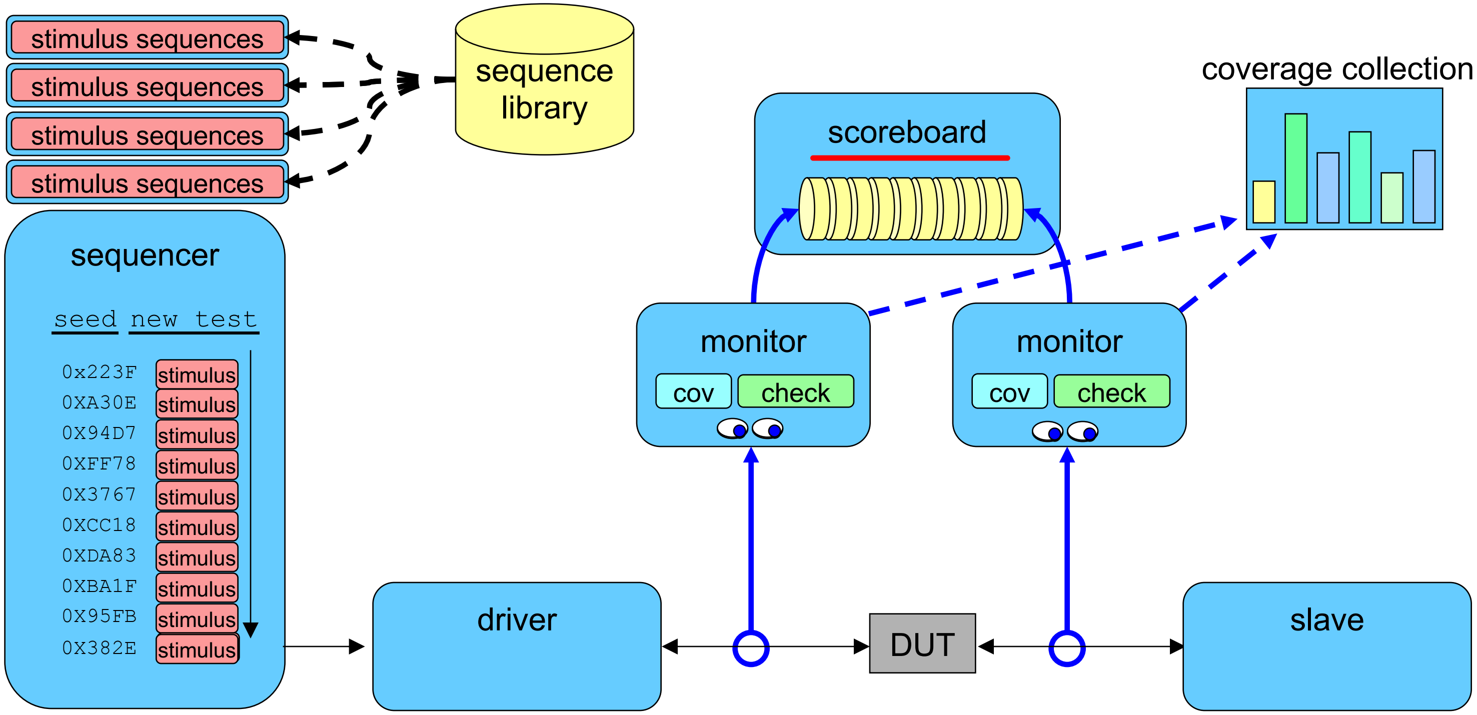
**Coverage** shows which ***goals*** have been exercised and which need attention.

(Checking needs to be in place to assess DUT/DUV response.)

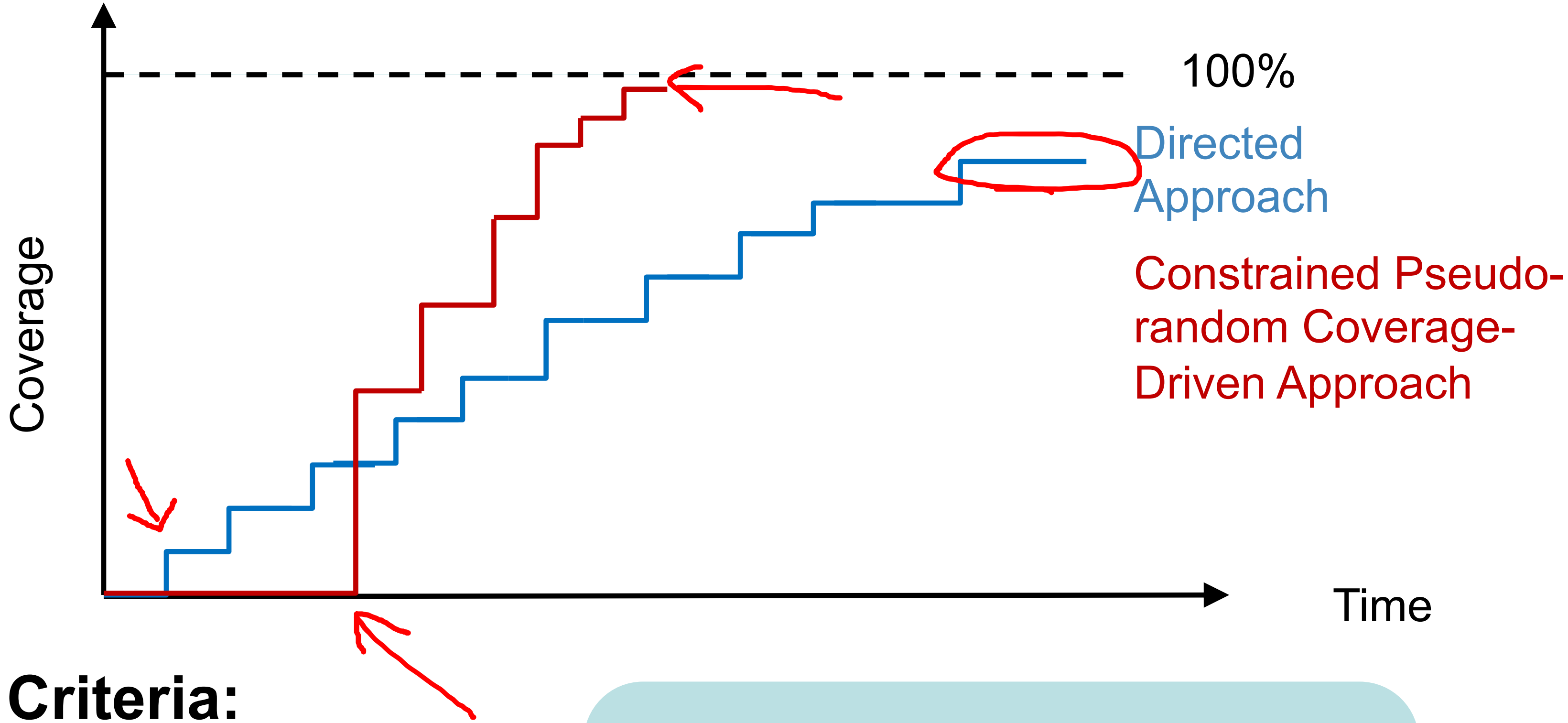
**Defining Coverage "Goals" Enables Automation:** Constrained-random stimulus generation accelerates hitting coverage goals and exposing bugs. Coverage and the results of checking indicate effectiveness of each simulation. This also enables many parallel runs.

# Coverage-Driven Environment

- **Composition of a coverage driven environment**
  - Reusable stimulus sequences developed with “constrained random” generation.
  - Running unique seeds allows the environment to exercise different functionality.
  - Monitors independently watch the environment.
  - Independent checks observe correct behavior and flag incorrect behavior.
  - Independent coverage points indicate which functionality has been exercised.



# Directed Testing vs CDV



**Criteria:**

- Effectiveness
- Efficiency
- Maintainability
- Re-usability

Seeing that directed testing has many shortfalls wrt these criteria.

*Why would one use Directed Testing?*

# Specman Elite Tutorial

- **DUV:** simple CPU (ALU, 4 regs, PC, PC_Stack, fetch/exec FSM)
  - Interface: clock, reset, instruction [8 bit]
- Learn how to:
  - Design the verification environment
  - Define DUV interfaces
  - Generate a simple test
  - Drive and check the DUV
  - Generate constraint-driven tests
  - Define and analyse test coverage
  - Create corner case tests
  - Create temporal and data checks
  - Analyse and bypass bugs
- About 100 pages. A really easy “learn by doing” lab. Takes about 2h. ☺

# On-line Help

- All Specman and “e” language help is on-line:
  - e language reference
  - Command reference for Specman Elite
  - User guide etc.

Make sure you follow the EDA setup:

```
> module use /eda/cadence/modules
> module load course/COMS30026
```

Then:

- For sn and “e” help or other help with Cadence tools use
  - `sn_help.sh` from command line or
  - `cdnshelp` from command line.

# We have now covered

- Basics of the “e” verification language and the important features of SN.
  - If you take this unit with coursework, you should be registered for the ***Specman Fundamentals for Block-Level Environment Developers*** *online training course, which introduces you to SN and e in more detail and provides you with exercises.*
- DEMO session of SN and e code for calc1 DUV
- Next:
  - Practical 2 (available on BB to be done by the END of week 11):
    - On BB template .e code and guidance on verification strategy