Functional Hardware Verification

Introduction to Constrained Random Coverage Driven Verification

This material is adopted from Internet.

Agenda

- Introduction
- Coverage
- Planning
- Testbench structure
  - UVM
- Specific aspects of verification:
  GLS, Formal, Mixed signal, Low power

Directed vs. Random

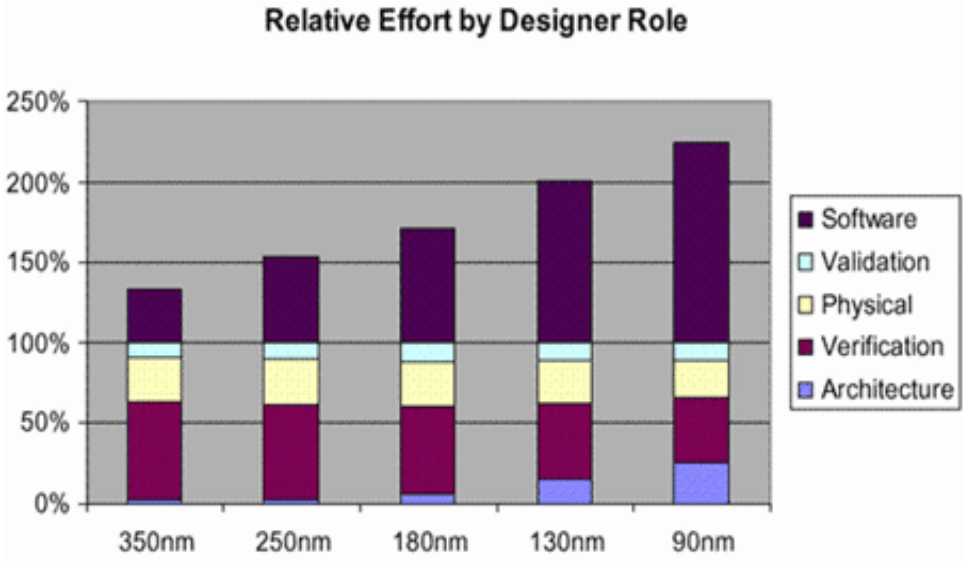
INTRODUCTION

What is Functional Verification

- Functional verification is the task of verifying that the logic design conforms to specification.
- Functional verification attempts to answer the question: "Does this proposed design do what is intended?"

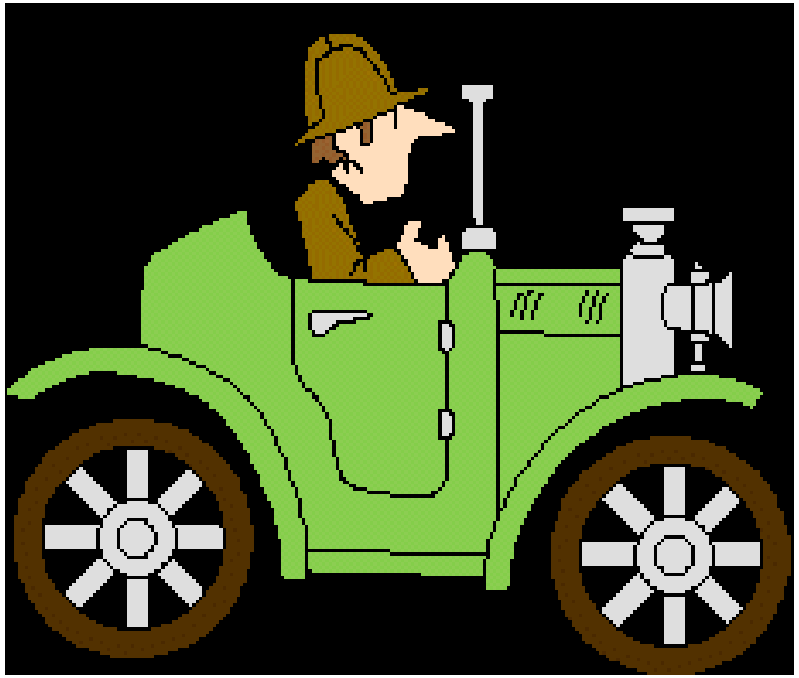
Why is Verification Important

- The estimated mean time required for IP verification is 70% of the overall IP development time.
  Design process time is only 30%



Directed Approach

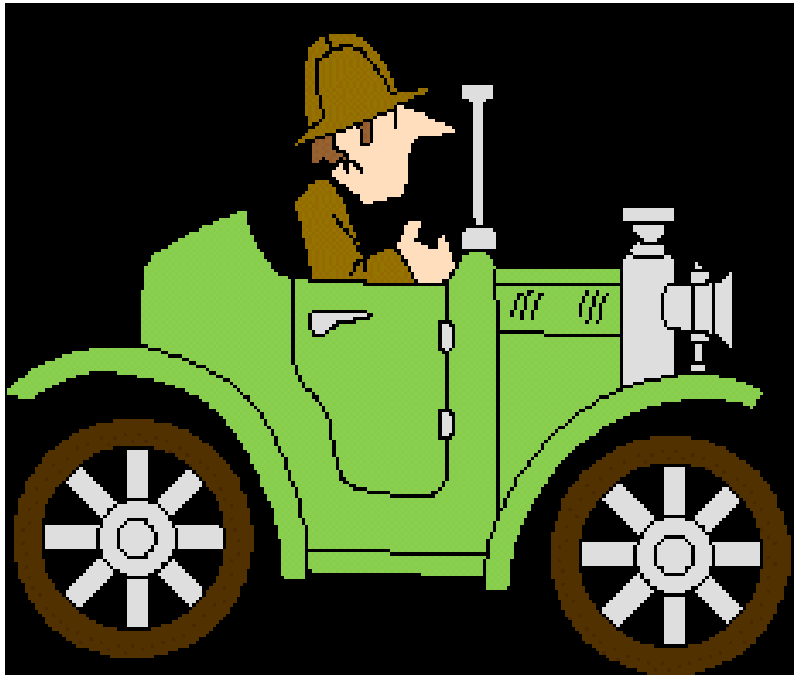
- Imagine verifying a car using directed method
  - Requirement: Fuse will not blow under any normal operation
  - Scenario 1: Accelerate to 50 km/h, put in the Barry White CD, and turn on the windshield wipers



A few weeks later …

Directed Approach

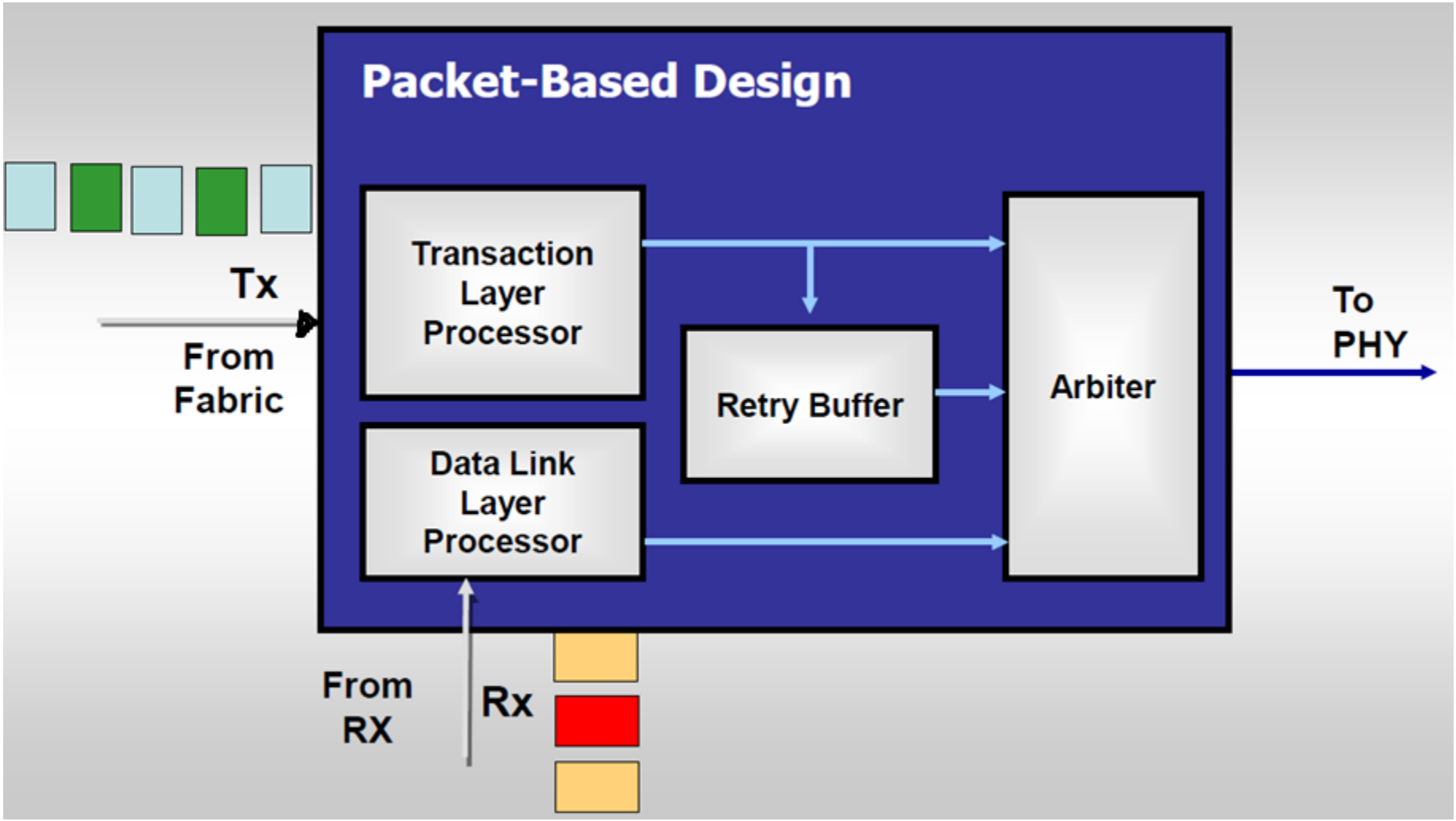
- Imagine verifying a car using directed method
  - Requirement: Fuse will not blow under any normal operation
  - Scenario 714: Accelerate to 60 km/h, roll down the window, and turn on the left turn signal



Concurrency Challenge

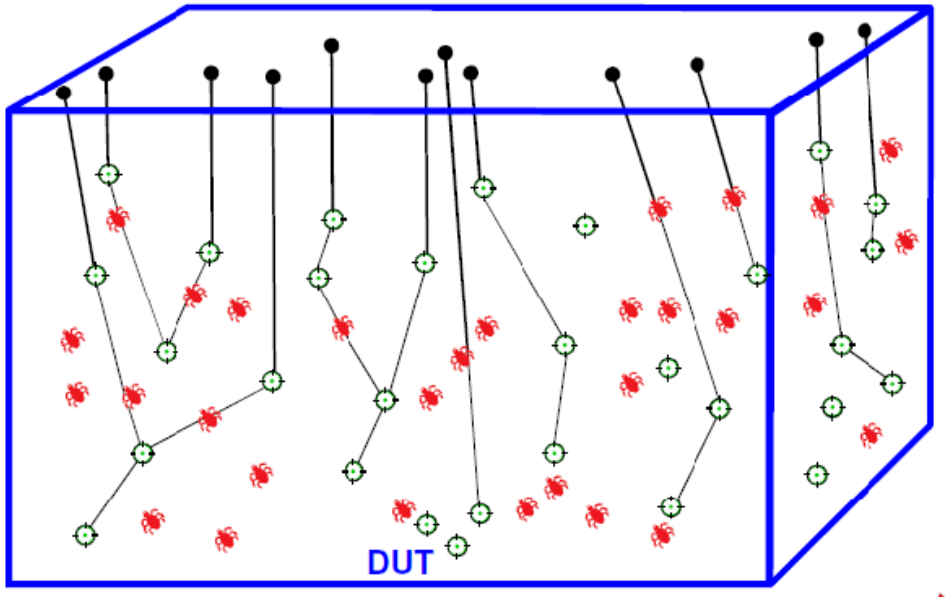
- A purely directed test methodology does not scale.
  - Imagine writing a directed test for this scenario!

Concurrency Challenge



Directed Testing

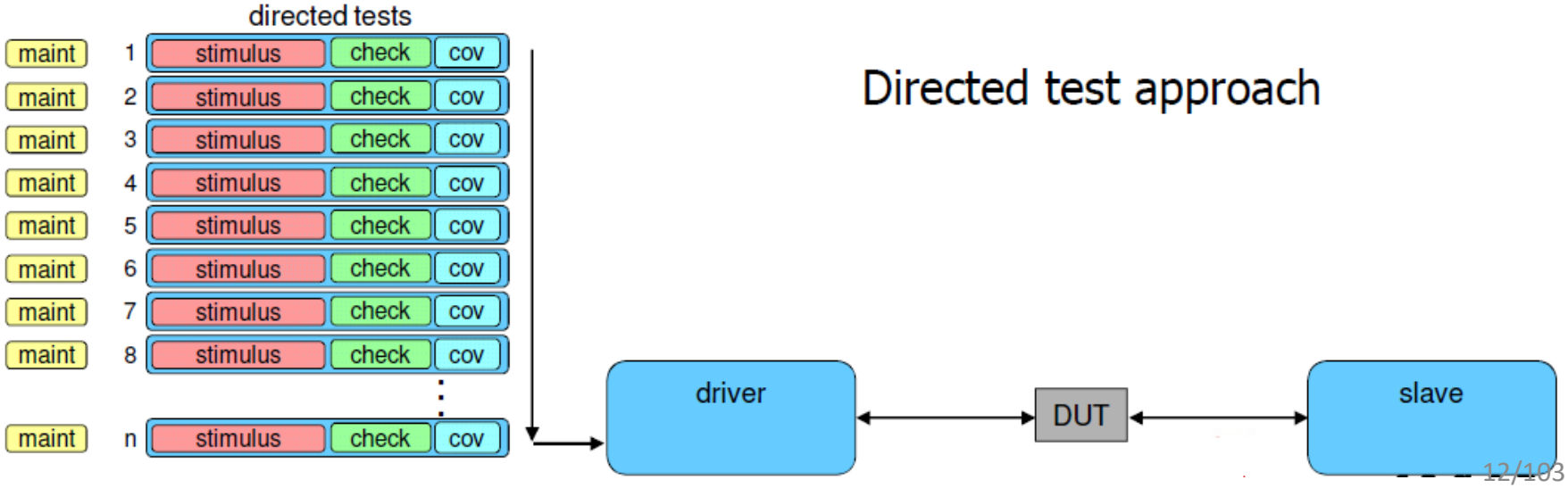
- Verification engineer:
  - Defines DUT states to be tested based on spec behavior and corner cases
  - Writes directed test to verify each item in Test Plan



Significant manual effort to write all the tests!
Lots of work required to verify each goal.
Poor coverage of non-goal scenarios …
Especially the cases you didn't "think of"

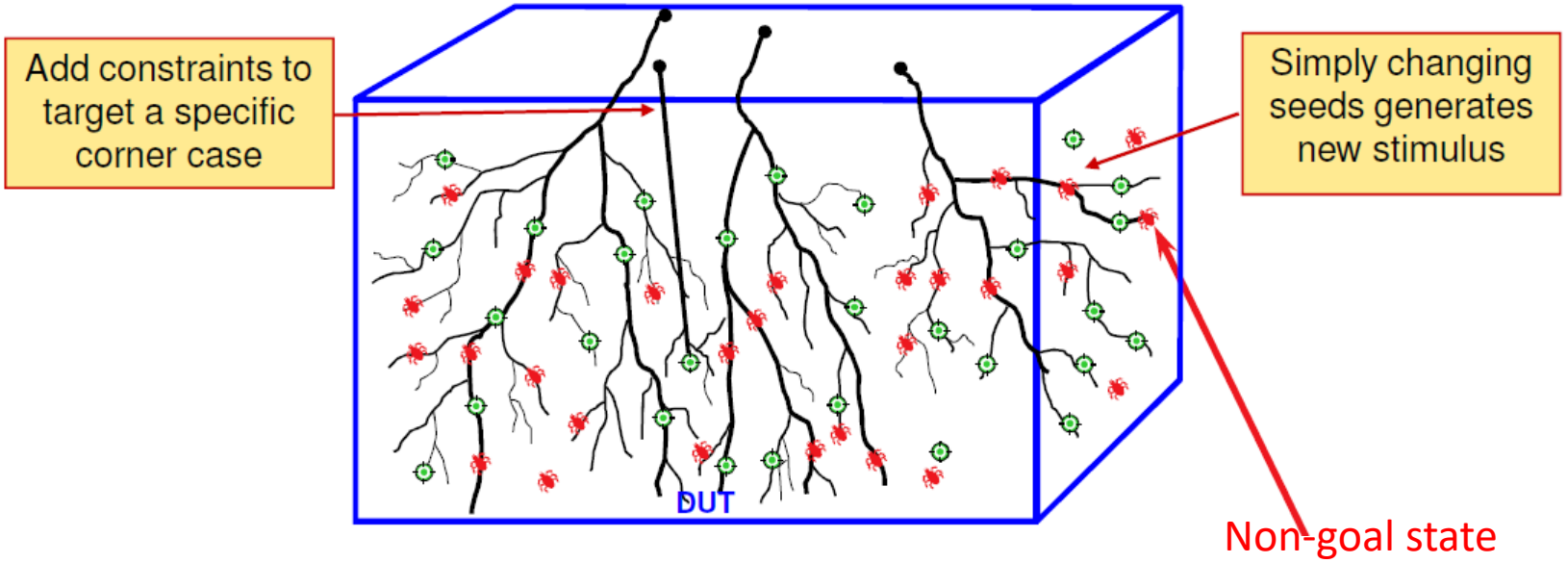
Directed Test Environment

- Composition of a directed test
  - Directed tests each contain stimulus, but each requires more than that…
  - Checks are typically embedded into the tests to verify correct behavior (*usually **specific** to that test scenario)*
- *Reusability and maintenance*
  - Tests can become quite complex and difficult to understand
  - 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



Constrained Random Coverage Driven Verification

- Verification engineer
  - Defines DUT states to be tested based on spec behavior (the function to be implemented) and corner cases (edge values and function discontinuities)
  - Focus moves to reaching goal areas (creative), versus execution of test lists (brute force)



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

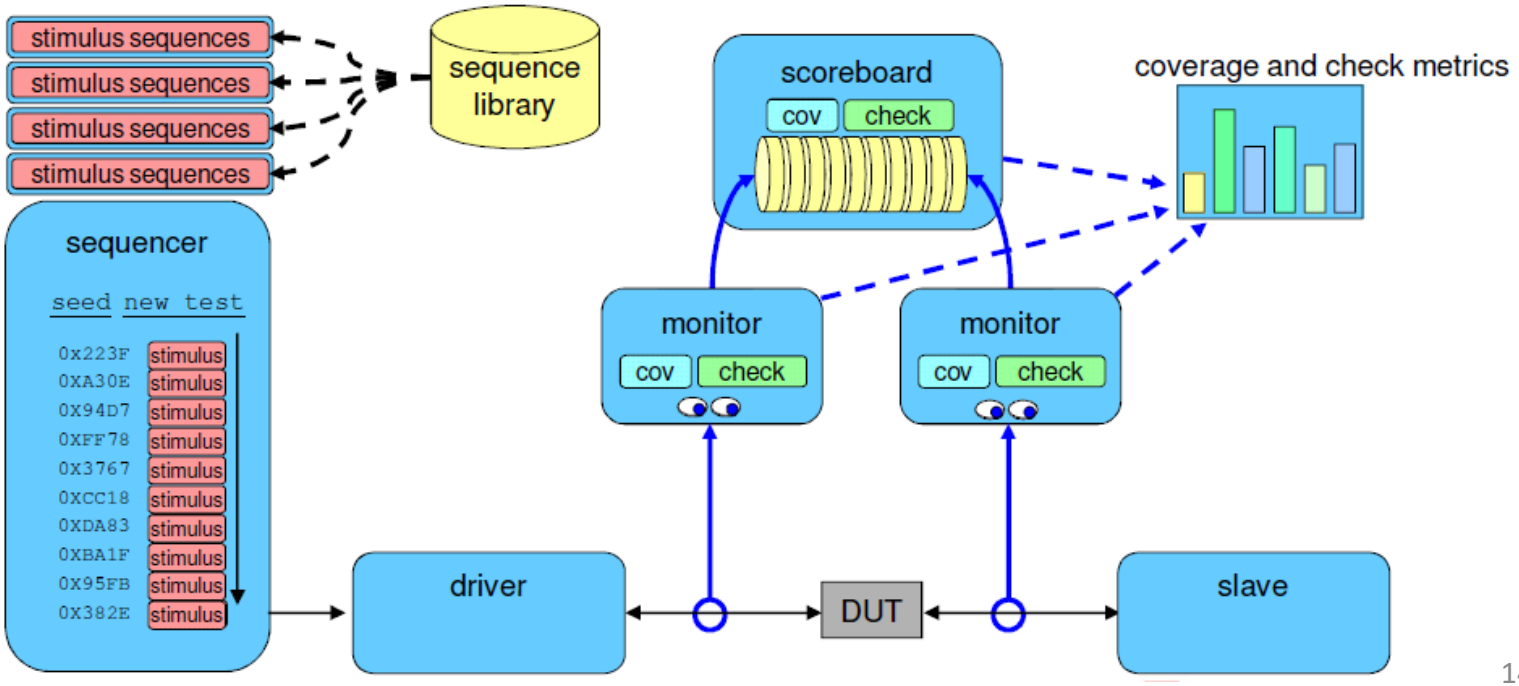
Stimuli are automatically generated, based on the constraints specified by VE

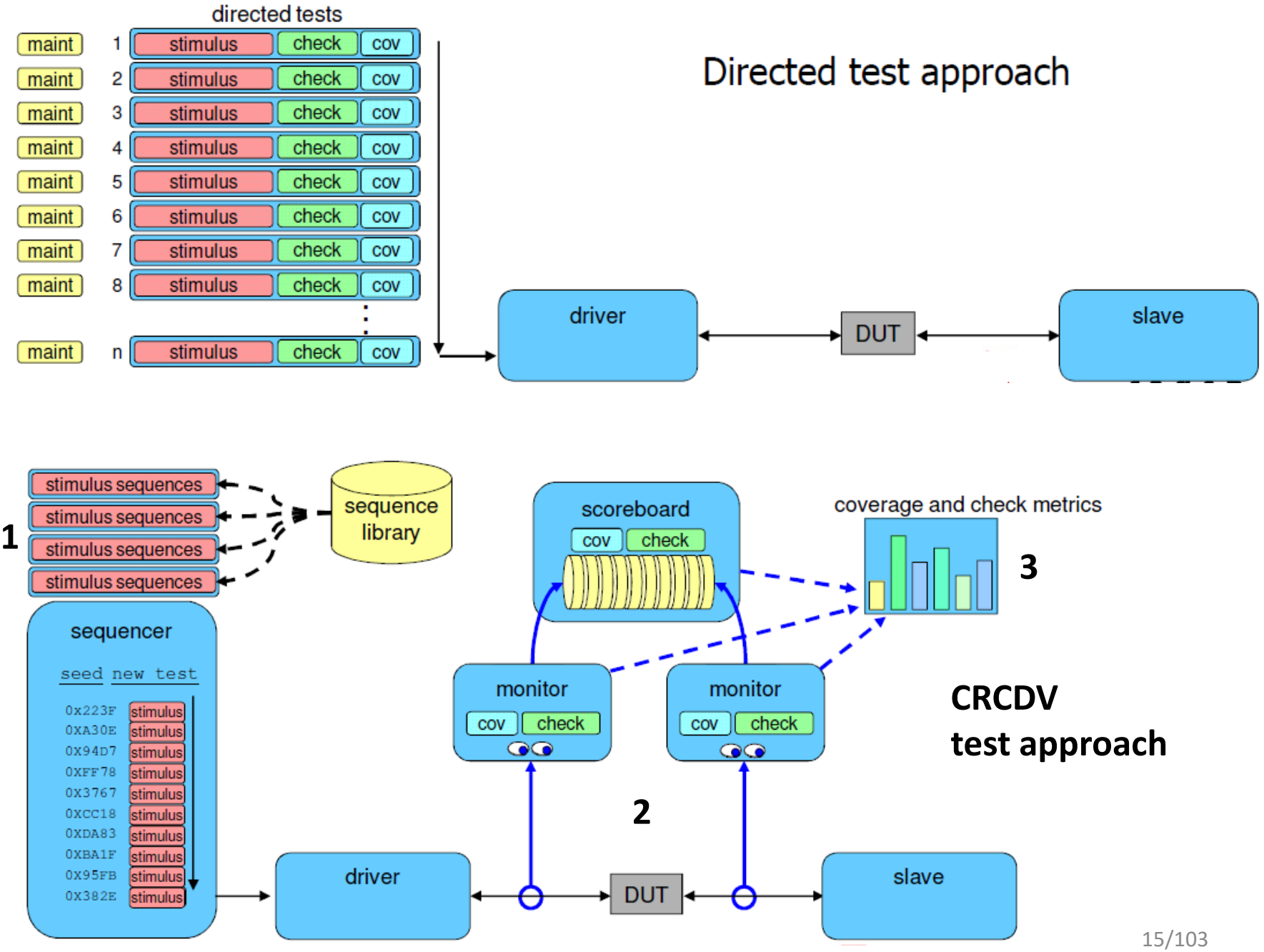
Coverage reports which ***goals*** have been exercised and which need attention

Self-Checking ensures proper DUT response

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 ensure correct behavior
  - Independent coverage points indicate which functionality has been exercised




directed tests
maint 1 stimulus check cov
maint 2 stimulus check cov
maint 3 stimulus check cov
maint 4 stimulus check cov
maint 5 stimulus check cov
maint 6 stimulus check cov
maint 7 stimulus check cov
maint 8 stimulus check cov
maint n stimulus check cov
Directed test approach
driver
DUT
slave
stimulus sequences
stimulus sequences
stimulus sequences
stimulus sequences
1
sequence library
scoreboard
cov check
coverage and check metrics
3
sequencer
seed new test
0x223F stimulus
0XA30E stimulus
0X94D7 stimulus
0XFF78 stimulus
0X3767 stimulus
0XCC18 stimulus
0XDA83 stimulus
0XBA1F stimulus
0X95FB stimulus
0X382E stimulus
monitor
cov check
monitor
cov check
2
driver
DUT
slave
CRCDV
test approach

Where Are the Major Differences?

1. Stim… gen…….
2. Mon……. and che…..
3. Cov….. Collection

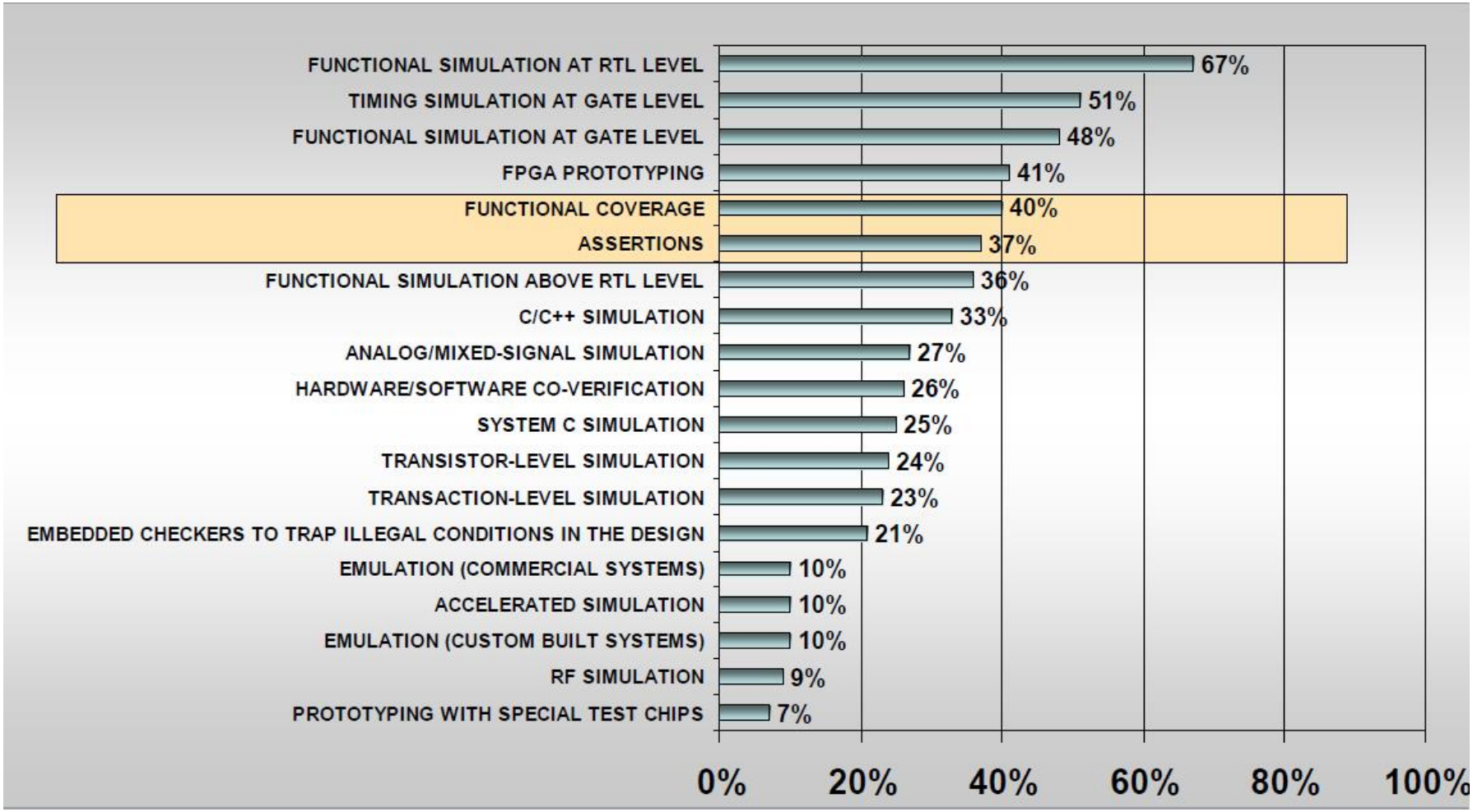
What is the Major Benefit?

Coverage Collection vs. Test Vector Generation

Where Are the Major Differences?

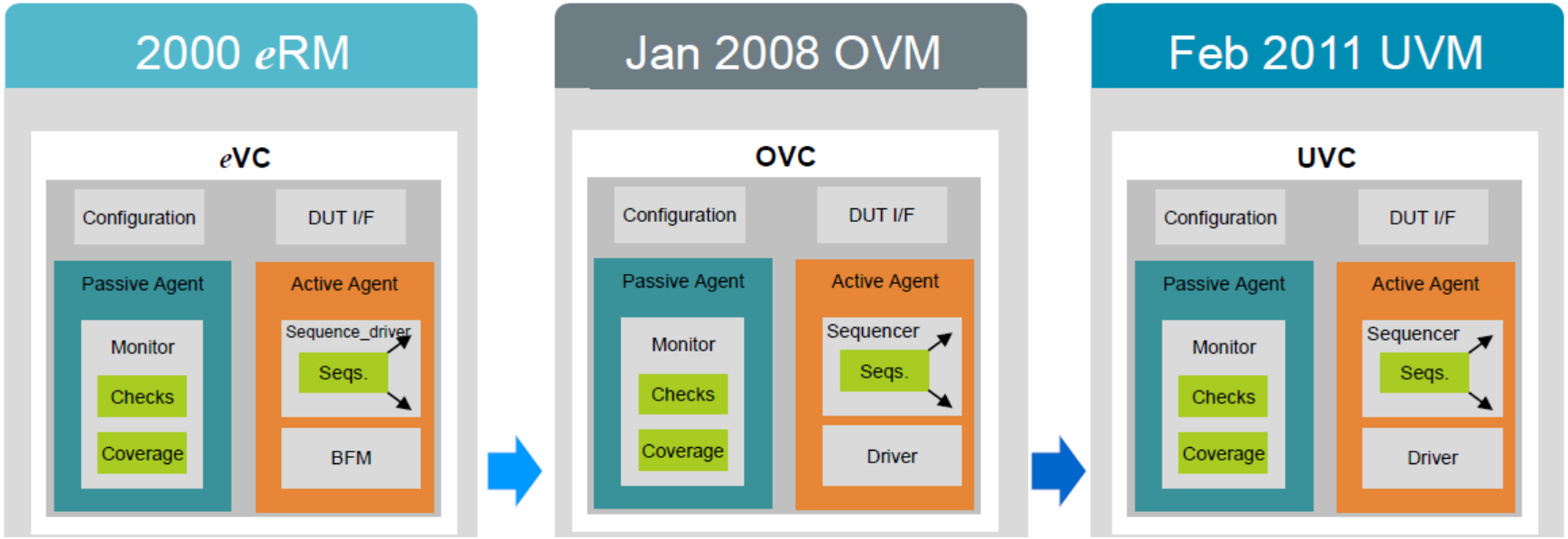
1. Stimuli generation
2. Monitoring and checking
3. Coverage collection

Dynamic Techniques Adopted in Industry



Existing Technologies

- Languages: e + System Verilog, C/C++, SystemC
- Methodologies: eRM + UVM
- Tools: Cadence Incisive + Mentor Questa, Synopsys VCS…



When is verification finished?

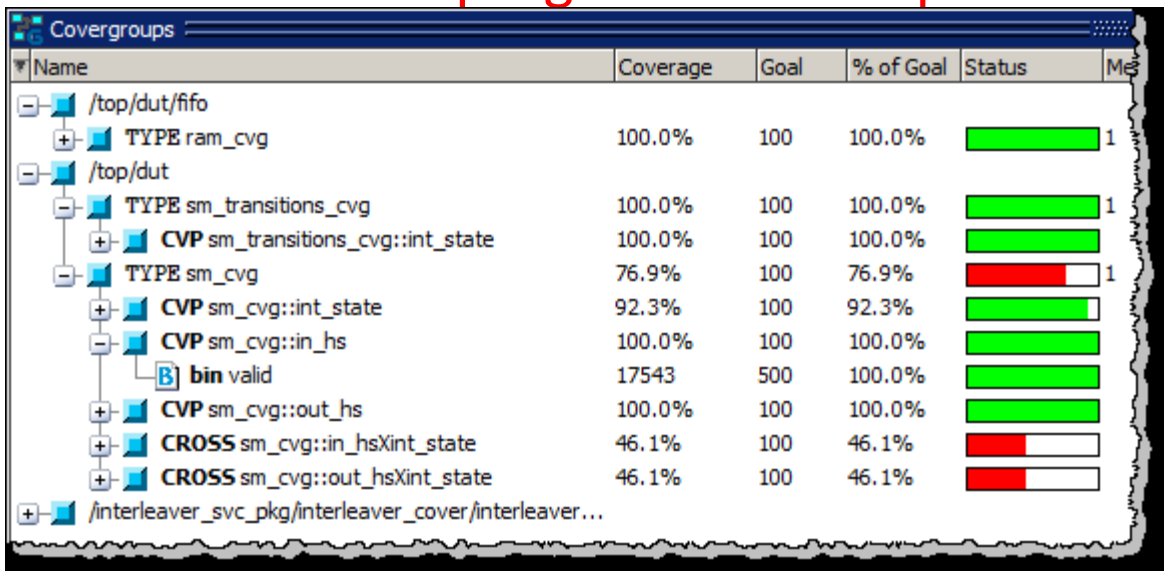
COVERAGE

What is Coverage?

- "What doesn't get measured might not get done."
- To answer the question "Are we done?" we need to answer:
  - Were all the design features and requirements (functions) identified in the testplan verified?
  - Were there lines of code or structures (code) in the design model that were never exercised?
- Coverage is a simulation metric we use to measure verification completeness.

What is Coverage? (cont'd)

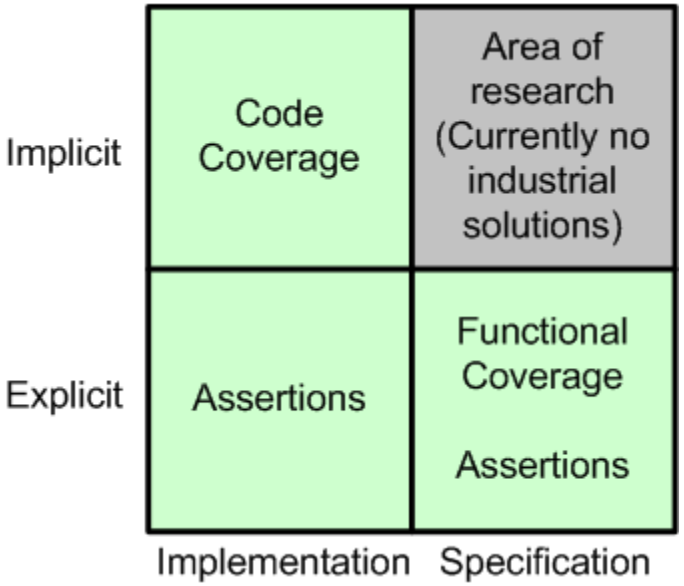
- Once we have a measure of coverage
  we might ask some more questions:
  - When we tested the feature *X*, did we ever test the feature *Y*
    at the exact same time?
  - Has our verification progress stalled for some unexpected reason?
  - Are there tests that we could eliminate
    to speed up our regression suite
    and still achieve our coverage goals?
- Coverage is a simulation metric we use
  to measure verification progress and completeness.



Name	Coverage	Goal	% of Goal	Status	Me
/top/dut/fifo					
TYPE ram_cvg	100.0%	100	100.0%		1
/top/dut					
TYPE sm_transitions_cvg	100.0%	100	100.0%		1
CVP sm_transitions_cvg::int_state	100.0%	100	100.0%		
TYPE sm_cvg	76.9%	100	76.9%		1
CVP sm_cvg::int_state	92.3%	100	92.3%		
CVP sm_cvg::in_hs	100.0%	100	100.0%		
bin valid	17543	500	100.0%		
CVP sm_cvg::out_hs	100.0%	100	100.0%		
CROSS sm_cvg::in_hsXint_state	46.1%	100	46.1%		
CROSS sm_cvg::out_hsXint_state	46.1%	100	46.1%		
/interleaver_svc_pkg/interleaver_cover/interleaver...					

Kinds of Coverage

- Classification by:
  - the method (explicit vs. implicit)
  - the source (specification vs. implementation)



- Two forms of coverage are used in industry today:
  - Code Coverage (Implicit coverage): Inherited from the design
  - Functional Coverage/Assertion Coverage (Explicit coverage): Made by VE

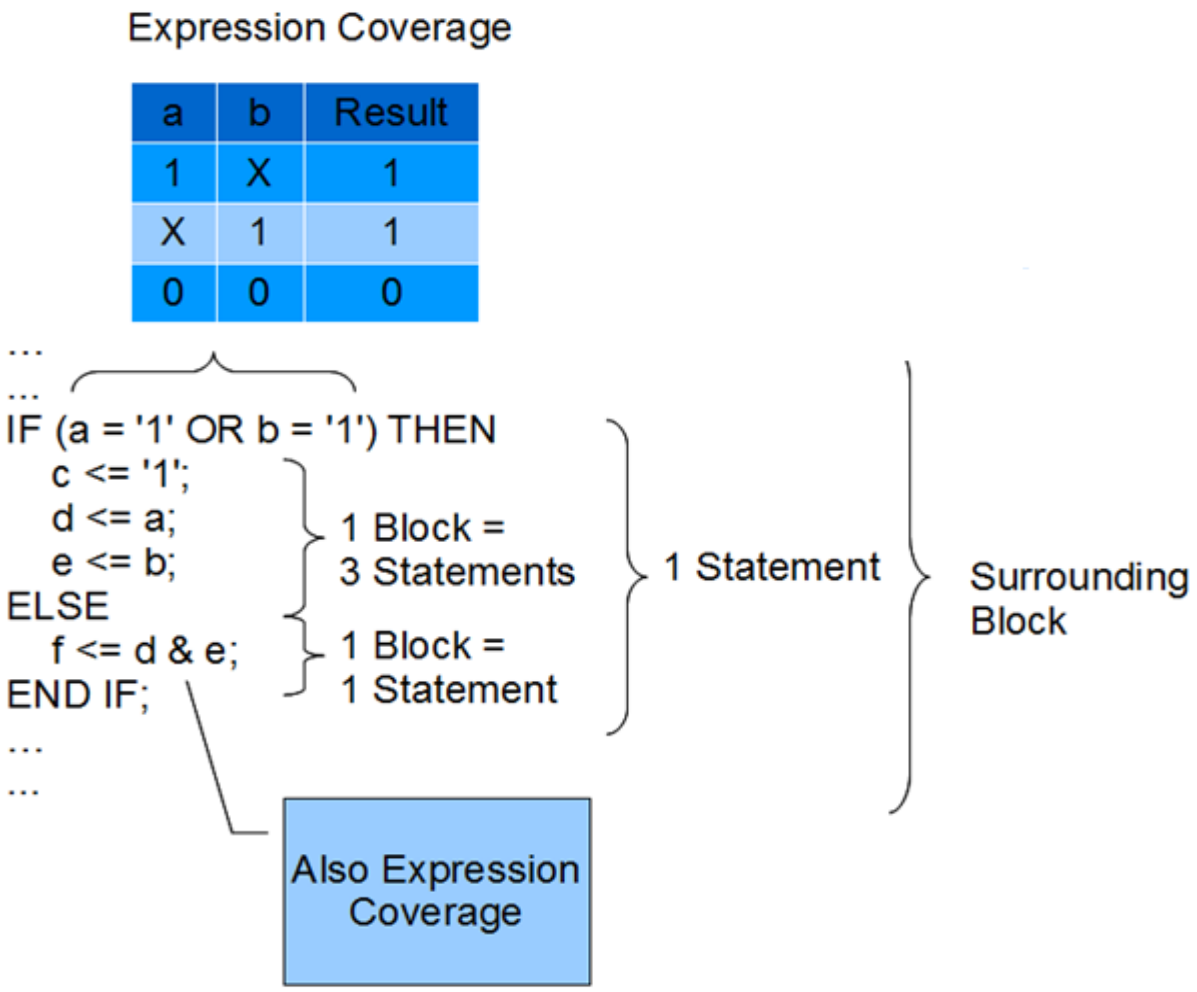
Code Coverage (remember: implicit)

- Code coverage is a measurement of structures within the source code that have been activated during simulation.
- Benefits:
  - It is implicit, so it is automatically generated with little effort
- Limitations:
  - The testbench must generate proper input stimulus to activate a design error
  - The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port
  - The testbench must contain a monitor that can detect the design error
  - Even with 100% of code coverage, there could be functionality defined in the specification that was never tested

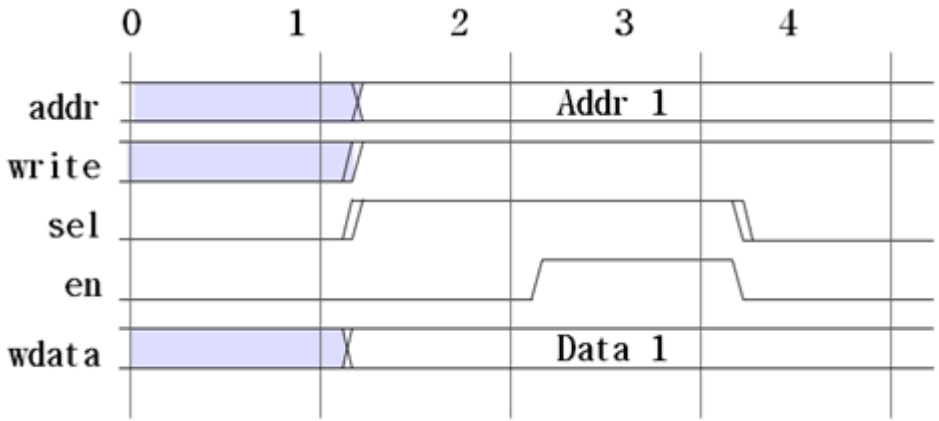
Types of Code Coverage Supported by Most Tools, in the Raising Level of Sophistication

- **Toggle coverage** - measure the number of times each bit of a register or wire has toggled. Good for connectivity checking.
- **Block coverage** is a variant on the statement coverage which identifies whether a block of code has been executed or not. Good for module verification.
- **Line coverage** - which lines of source code have been executed.
- **Statement coverage** - which statements within our source code have been executed.
- **Expression coverage** (or condition coverage) – - has each condition evaluated both to true and false. Good for exhaustive verification.
- **Branch coverage** – have Boolean expressions in control structures (*if*, *case*, *while*, *repeat*, *forever*, *for* and *loop* statements) evaluated to both true and false.

Types of Code Coverage – Statement, Block, Expression



Types of Code Coverage - Toggle Coverage



	0 -> 1	1 -> 0
sel	Y	Y
en	Y	Y
write	Y	N
Addr[0]	Y	N
Addr[1]	N	N
…	…	…

Code Coverage Flow

- It is best to wait until the RTL implementation is close to complete, before seriously starting to gather and analyze code coverage results.
- Once the results are available, coverage holes are analyzed.
- Coverage holes can be due to one of the following reasons:
  - Missing input stimulus required to activate the uncovered code
  - A bug in the design (or testbench) that is preventing the input stimulus from activating the uncovered code
  - Unused code for certain IP configurations or expected unreachable code during normal operating conditions

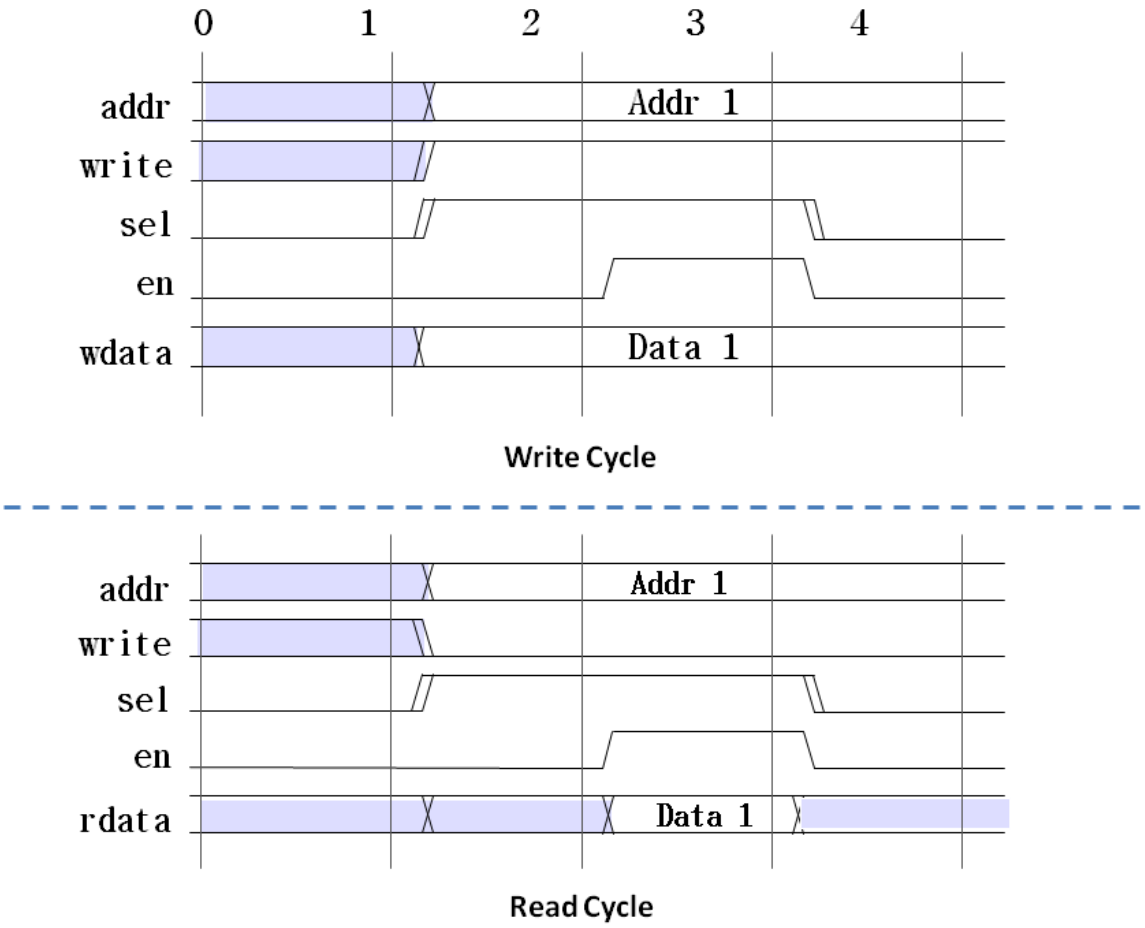
Functional Coverage (remember: explicit)

- Functional coverage helps us answer the question:
  Have all specified functional requirements been implemented, and then exercised during simulation?
- Benefits are defined in coverage definition:
  - It helps us measure verification progress and completeness
- Limitation is that it is explicit, which means it is required to:
  - Identify the functionality or design intent that you want to measure
  - Implementing the machinery to measure the functionality or design intent

Types of Functional Coverage

- **Cover Groups**
  consists of state values observed on buses,
  grouping of interface control signals, as well as register.
  The values that are being measured
  occur at a single explicitly or implicitly sampled point in time.
- **Cover Properties/Assertions**
  measure temporal relationships between sequences of events
- Definitions!
  - Assertion: Statement that proves a fact
  - Property: Coverage point of view property

Cover Groups vs. Cover Properties/Assertions



Example of a cover group: Have we exercised both, Read and Write commands?
Example of a cover property: Have we exercised the command
in which the EN signal is delayed, after the SEL signal, for 3 clock cycles?

Functional Coverage Flow

- The typical flow is:
  - Create a functional coverage model
  - Run simulation to capture and record coverage metrics
  - Report and analyze the coverage results
- Coverage holes have to be minimized;
  can be due to one of the following reasons:
  - Missing input stimulus (required to activate the uncovered code), because the VE missed to specify properly:
    (a) the related functionality,
    (b) the related edge condition;
    typical error: over constraining!
  - A bug in the design (or testbench) that is preventing the input stimulus from activating the uncovered code, e.g., existence of unreachable points
  - Unused functionality for certain IP configurations or expected unreachable functionality during normal operating conditions, e.g., existence of unreachable code

First thing first

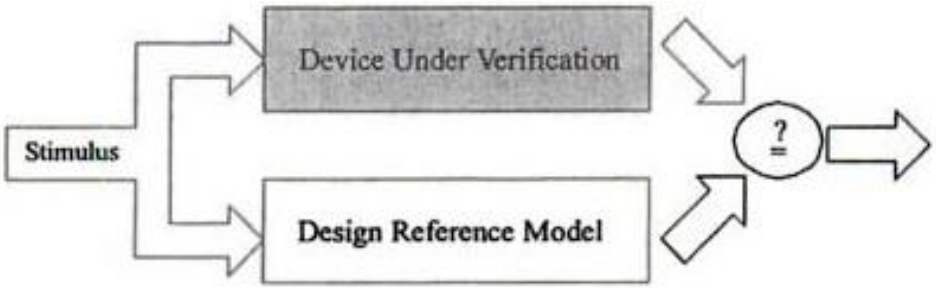
PLANNING

Planning

- What shall we cover:
  - Black box vs. white box verification
  - Cycle accuracy problem
  - Why is the plan important
  - What it needs to include
  - How to create it
  - What is executable plan

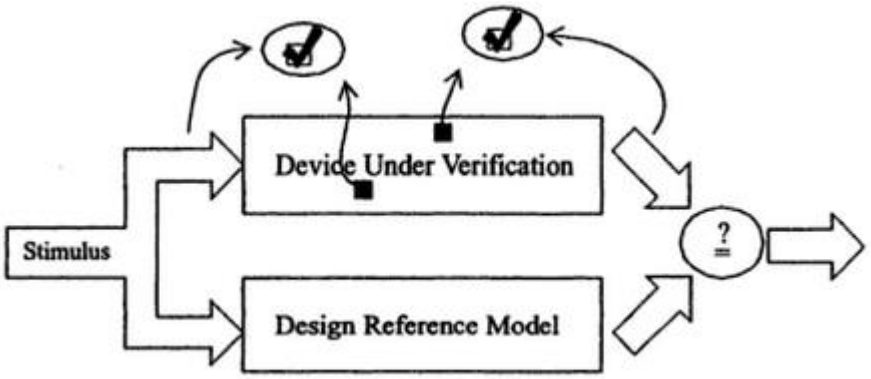
Black Box Verification

- In this approach, the DUT is treated as a black box, i.e., only through its boundary.
- Benefits:
  - DUT is verified realistically - as it will be used in the actual system
  - Testbench can be reused at system level
- Limitations:
  - Reference model needs to be developed
  - Some scenarios can be time consuming or impossible to verify



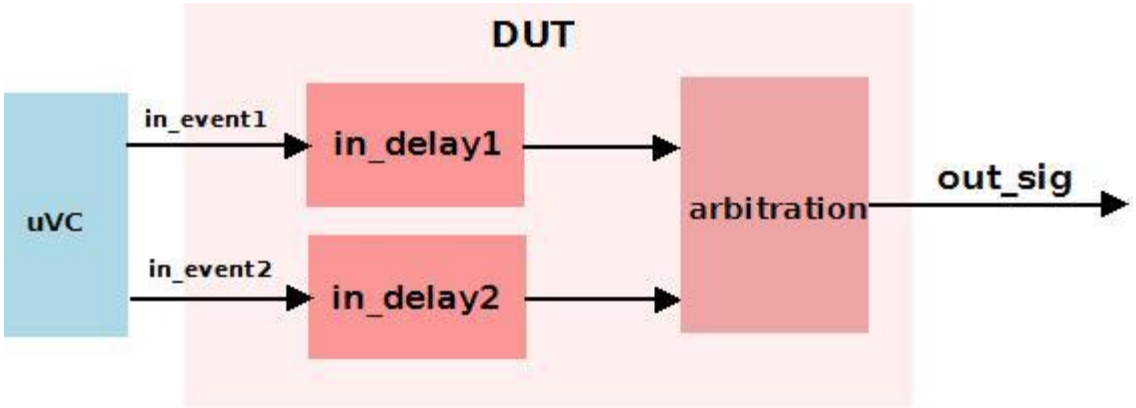
White Box Verification

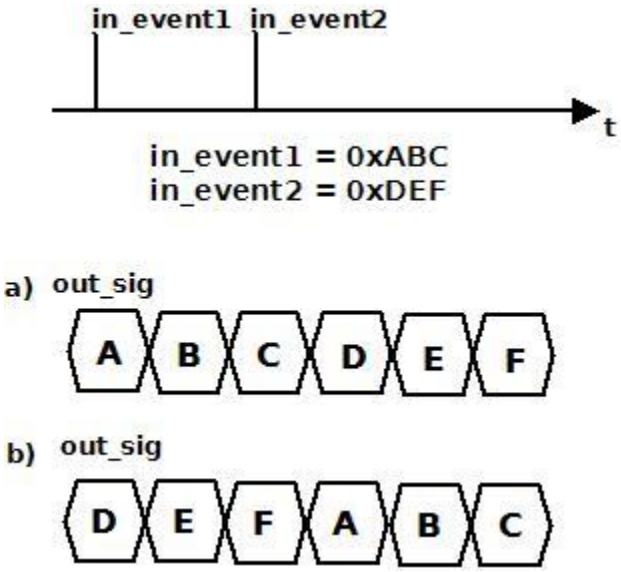
- In this approach, the DUT is treated as a white box, i.e., internal signals of DUT are used for additional checkers, coverage, or in a reference model to predict the output.
- Benefits:
  - Easy implementation
- Limitations:
  - Not reusable
  - Depends on design, so the VE may repeat the same error



Cycle Accuracy Problem

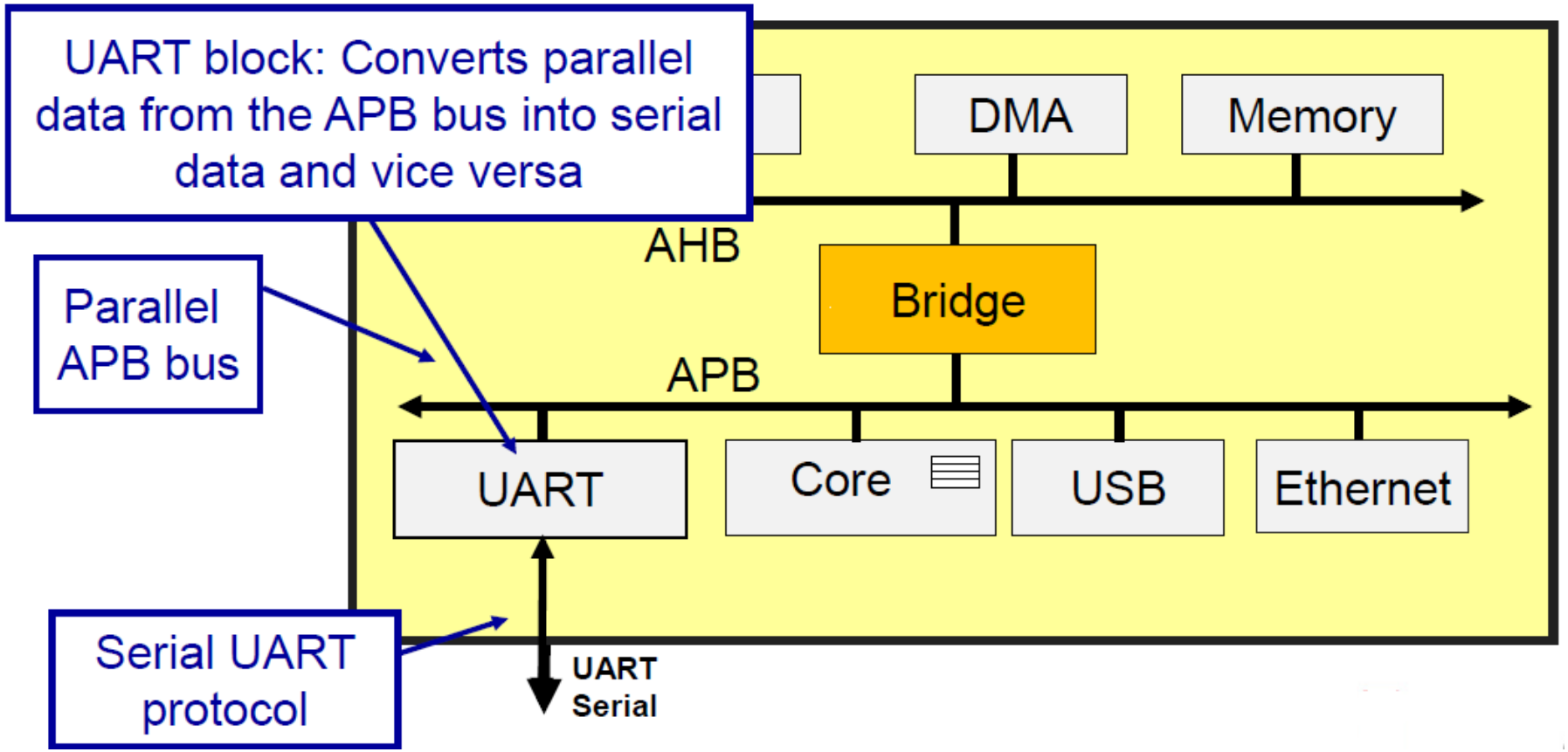
- How do we predict the order of packets?
- This is the Cycle Accuracy problem.





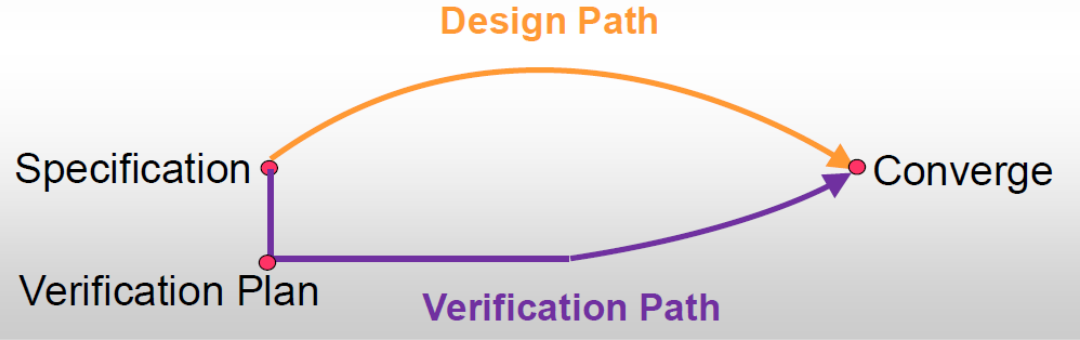
- The verification can not predict accurately which one of the above will happen after the HW is implemented
- The solution:
  - Make your testbench robust enough to be insensitive to cycle accuracy.
  - Use the white box approach

System on Chip



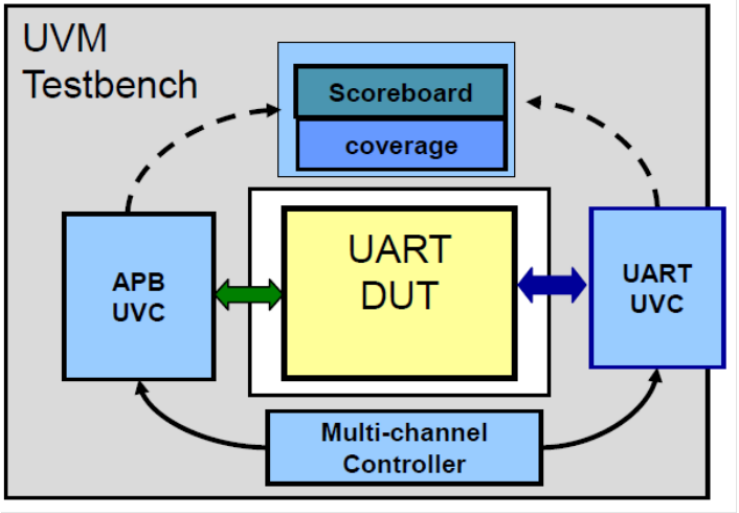
Why is Planning Important

- We already said that the verification accounts for approx. 70% of development time.
- Good planning is the best chance we have to shorten that time.



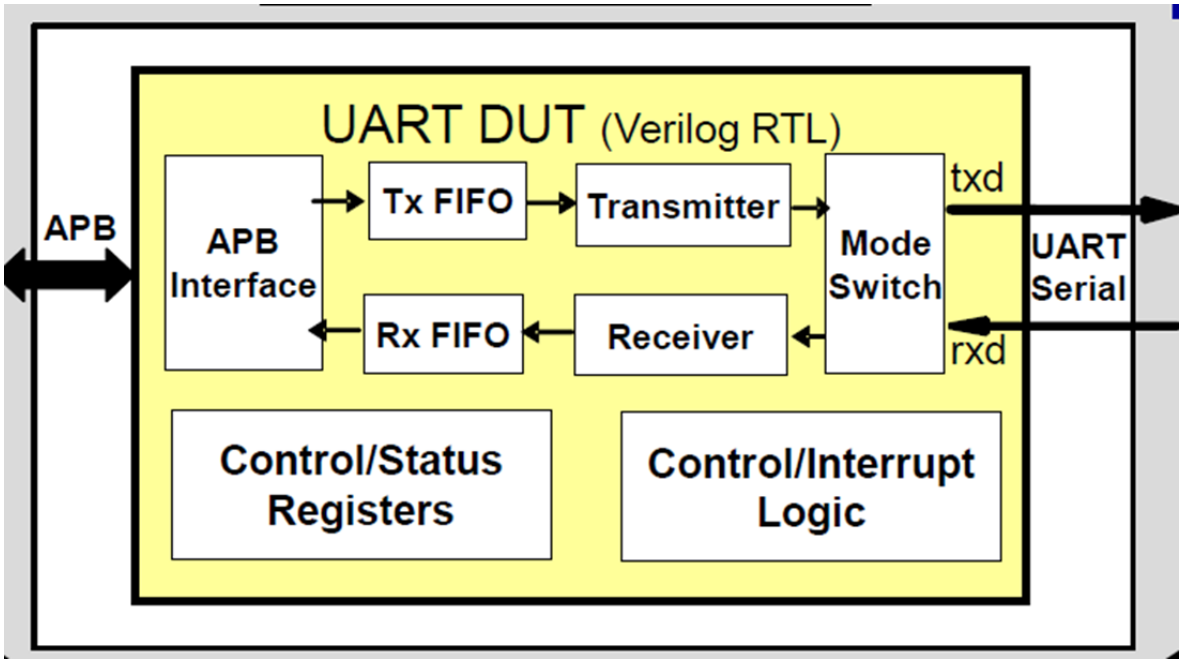
What is Contained in the Verification Plan

- Architecture of the testbench
- Strategy definition:
  - Language (SV, e,…)
  - Which components will be developed and which will be reused
  - Completeness definition (functional and code coverage targets)
- List of tests
- Functional coverage definition
- List of Checkers
- List of features (requirements)



Spec to Plan

- Excerpt from the UART functional spec:
  "The core implements the APB SoC bus interface for communication with the system. The core implements UART serial transmitter and receiver. The core requires one interrupt. TX FIFO of size 16 with programmable threshold is implemented as data buffer on TX path. RX FIFO of size 16 with programmable threshold is implemented on RX path. The block diagram of the core is given below. "



Spec to Plan (cont'd)

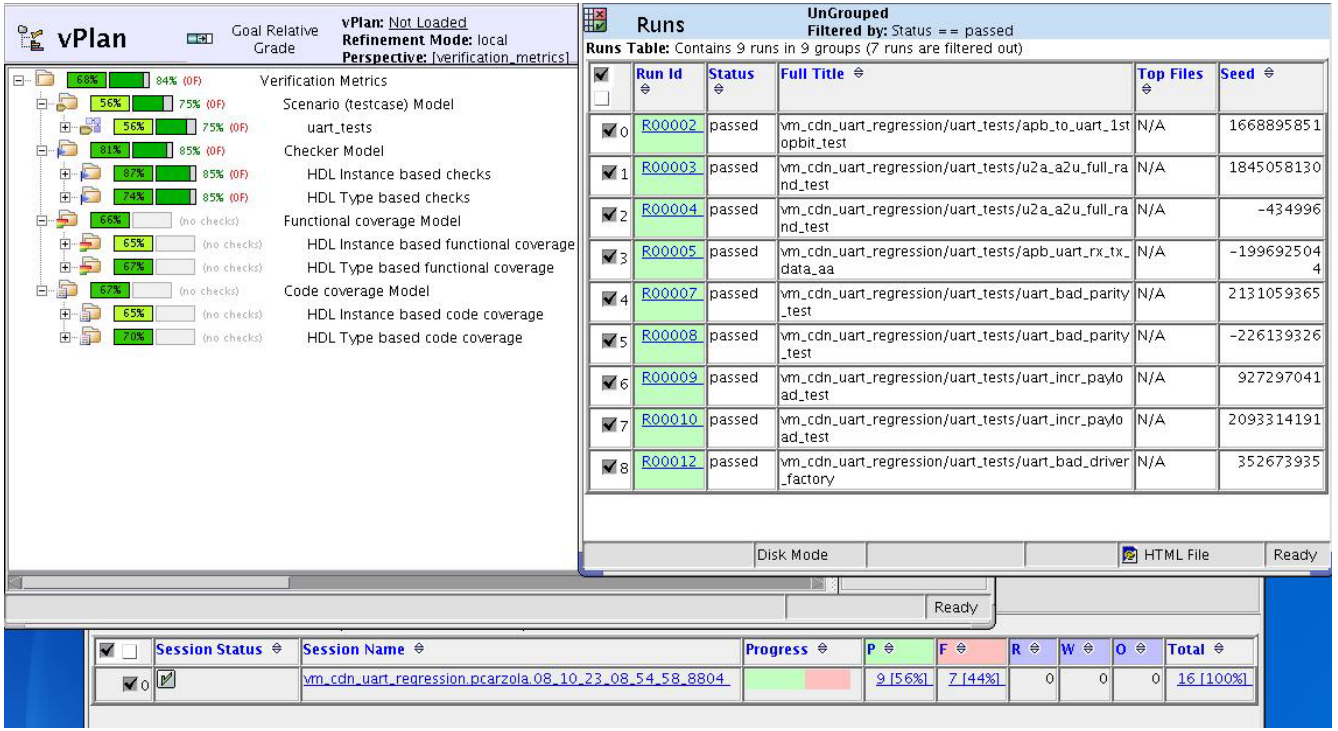
- Excerpt from the verification plan
  (obtained after the functional spec is mapped into the verification plan):

Tag	Description	Coverage	Directed	Priority
1	UART IP shall be compatible with APB protocol	YES		1
2	UART IP shall be compatible with UART protocol	YES		1
3	TX FIFO			
3.1	Maximum supported size is 16 words	YES	YES	1
3.2	Thresholds supported (2, 4, 8, 14)	YES		1
3.3	Overflow	YES	YES	2
3.3.1	Overflow happens exactly when 16 + 1 word is written in TX_FIFO		YES	2
3.3.2	Overflow recovery procedure is supported	YES	YES	2
4	RX FIFO			

Note: Directed method is still useful for specific features

Executable Plan

- Executable verification plan is the logical next step
- It directly links requirements from the plan
  with the actual metrics from the testbench:
  code coverage, functional coverage, properties/assertions, tests
- Verification progress is then tracked automatically.



Verification Environment Structure

UVM

SystemVerilog

- SystemVerilog is a combined Hardware Description Language and Hardware Verification Language
  based on extensions to Verilog
- Key verification extensions:
  - New data types (strings, dynamic arrays, associative arrays, queues)
  - Classes
  - Constrained random generation
  - Assertions
  - Coverage

What is UVM?

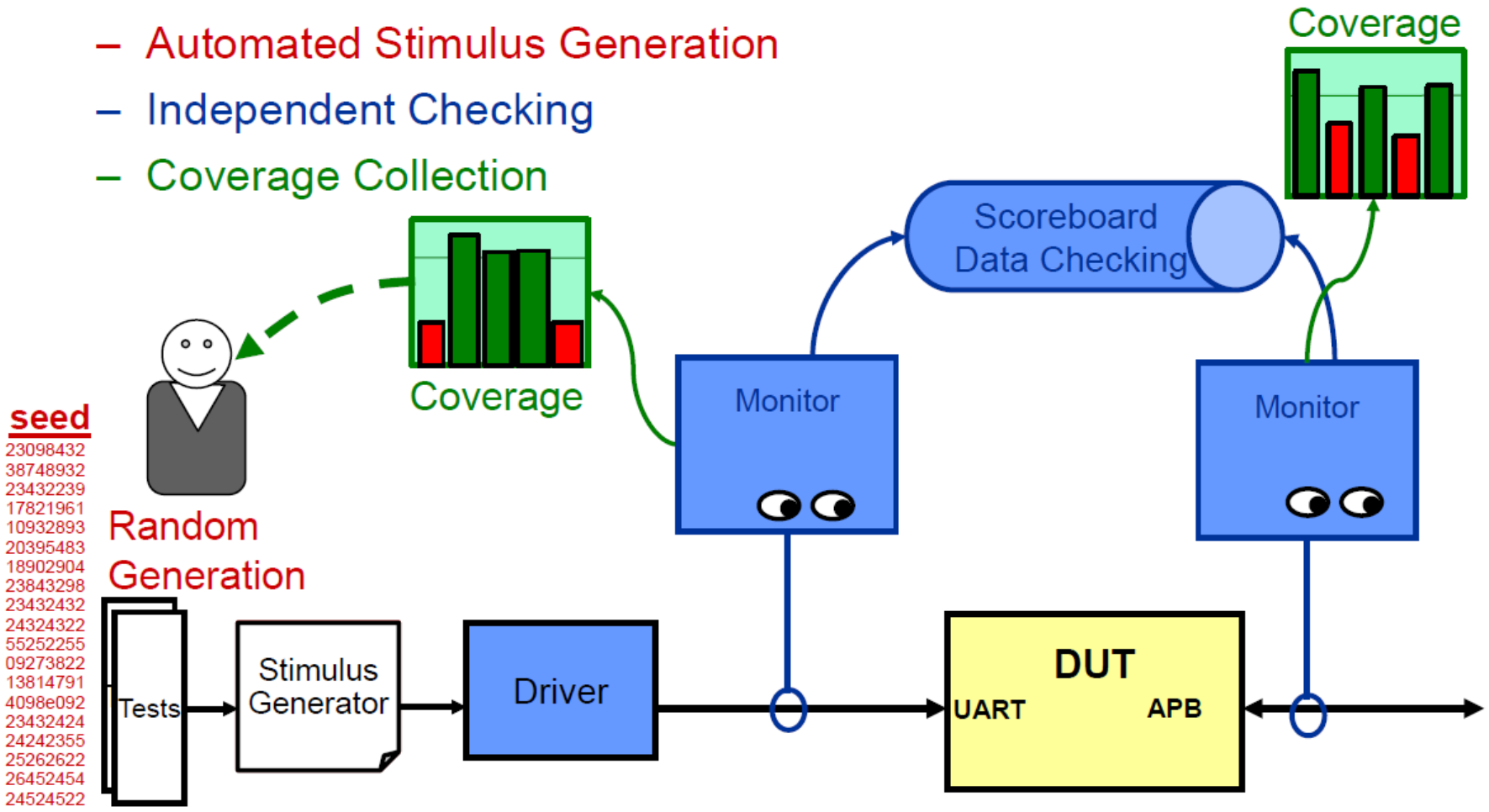
- UVM is the Universal Verification Methodology
  - A methodology and a library that codifies the best practices for efficient and exhaustive verification
- A complete, proven solution
  - Proven solution, with a success record and large community of users with methodology knowledge and commitment
  - Well-thought-out solution for a wide variety of verification challenges
- Open
  - Standard supported by all major EDA providers
- Enables reuse of verification environments
  - Verification IP can dramatically speed-up delivery and improve quality

UVM and Coverage Driven Verification

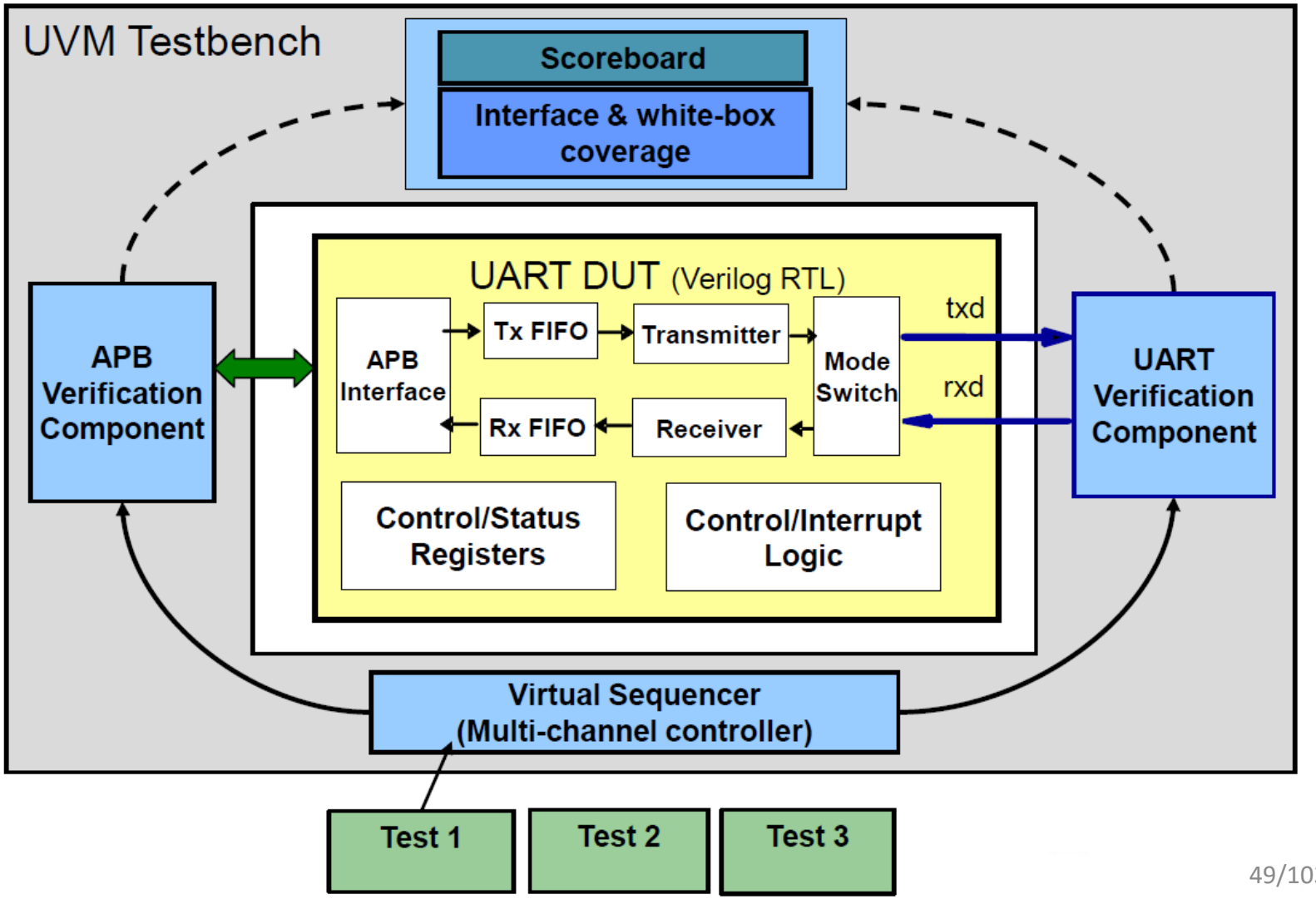
- Coverage-driven verification(CDV) combines
  the following aspects
  to significantly reduce the time spent verifying a design
  - Automatic stimulus generation
  - Self-checking testbenches
  - Coverage metrics
- From the UVM point of view: Why CDV?
  - Eliminate the effort and time to write hundreds of tests
  - Ensure thorough verification using upfront goal settings
- UVM provides the framework to achieve CDV

Coverage Driven Verification

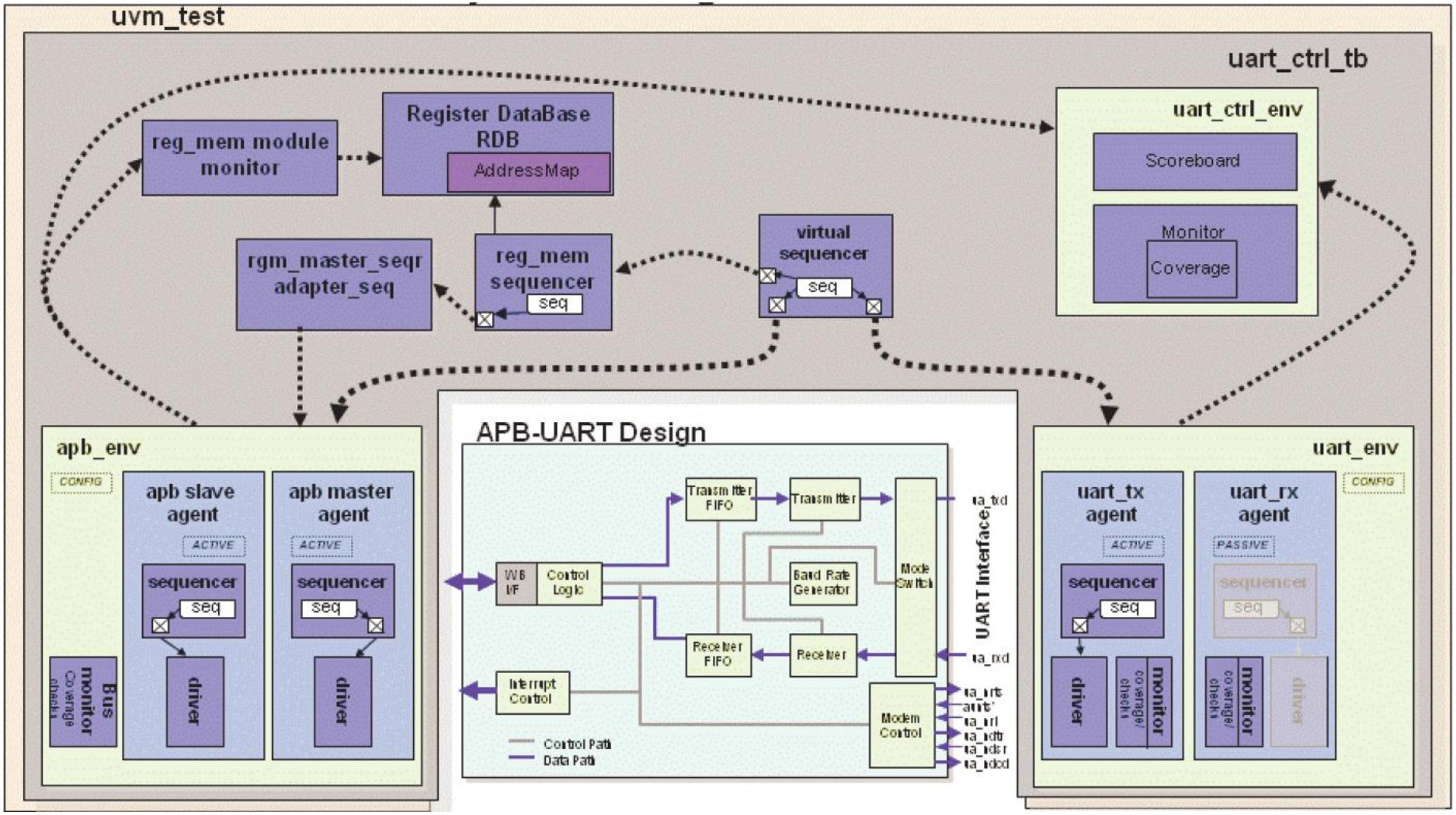
- Automated Stimulus Generation
- Independent Checking
- Coverage Collection



DUT and Verification Environment



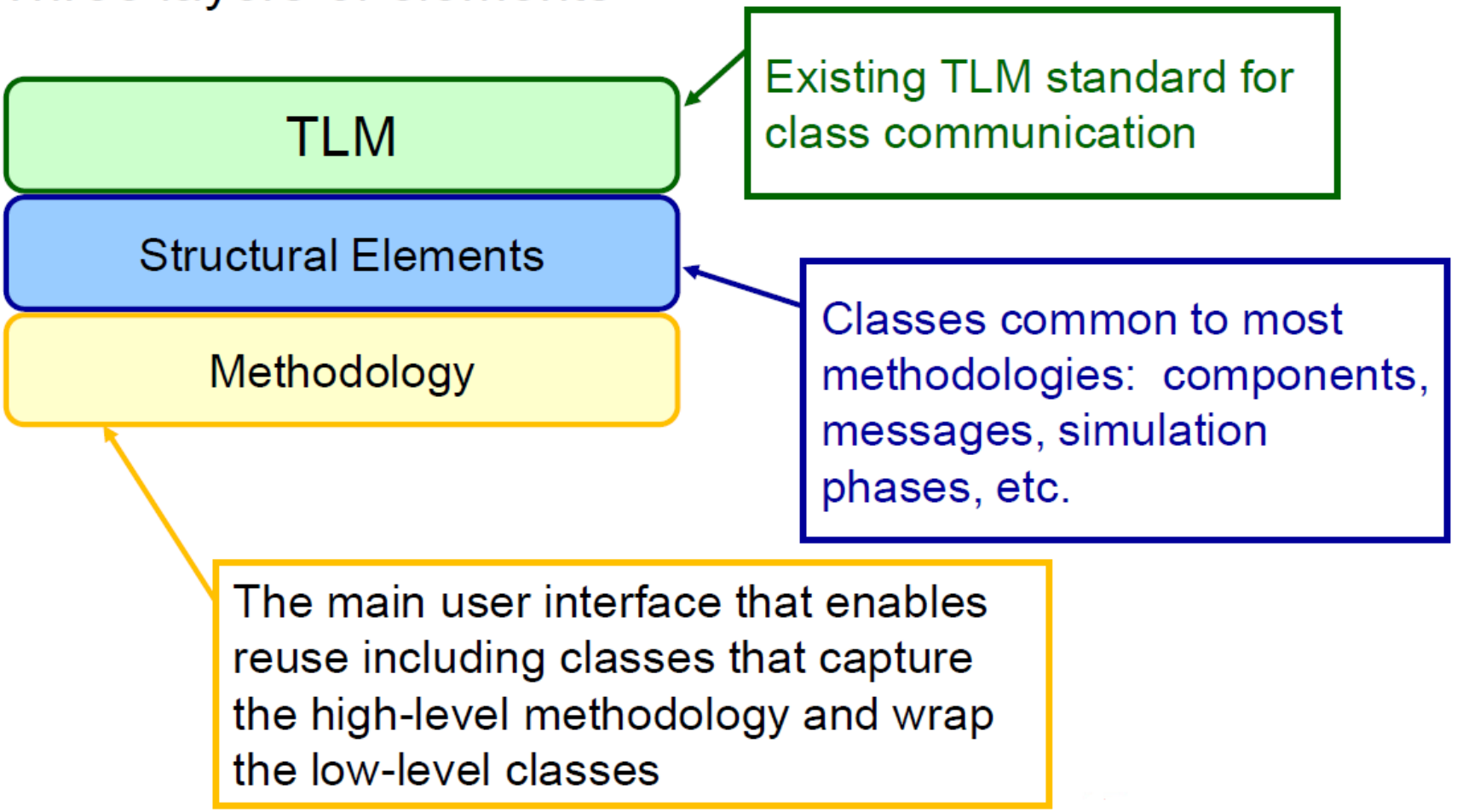
Detailed Verification Environment



Each purple block corresponds to one System Verilog class.

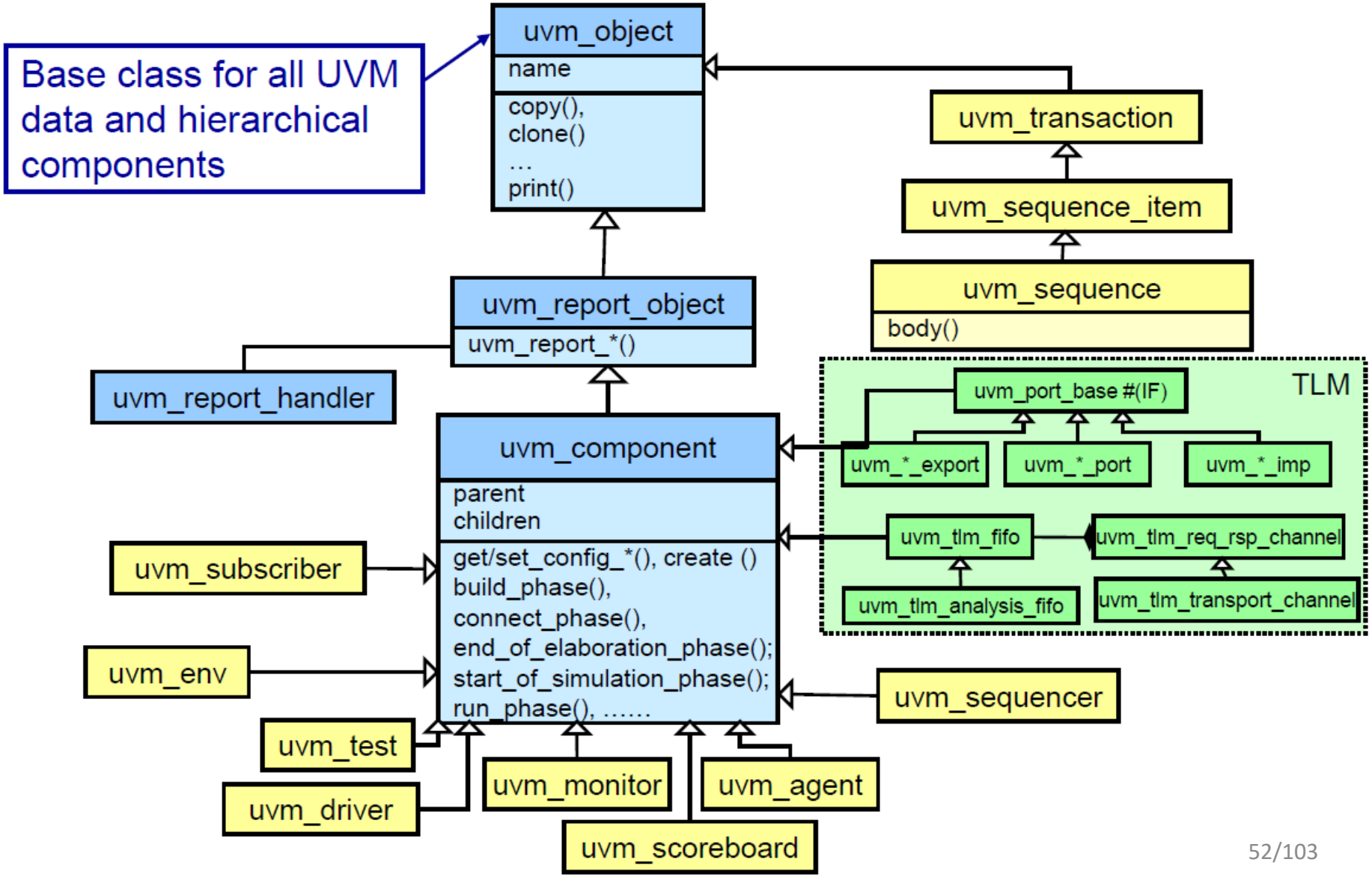
UVM Library

- Three layers of elements



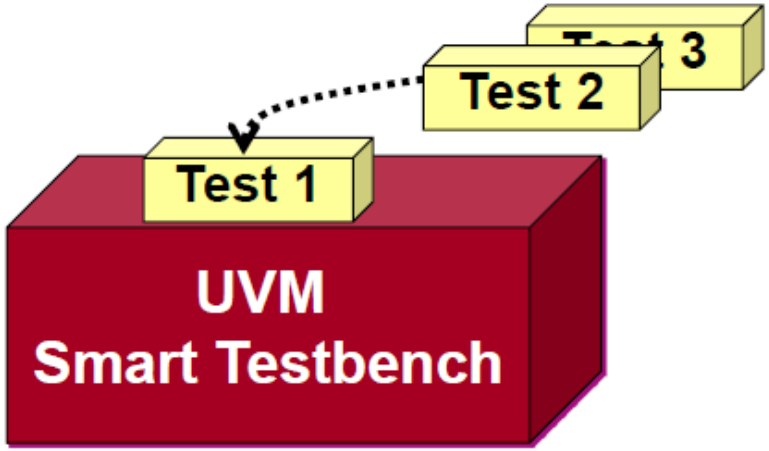
TLM = Transaction Level Model

UVM Class Hierarchy



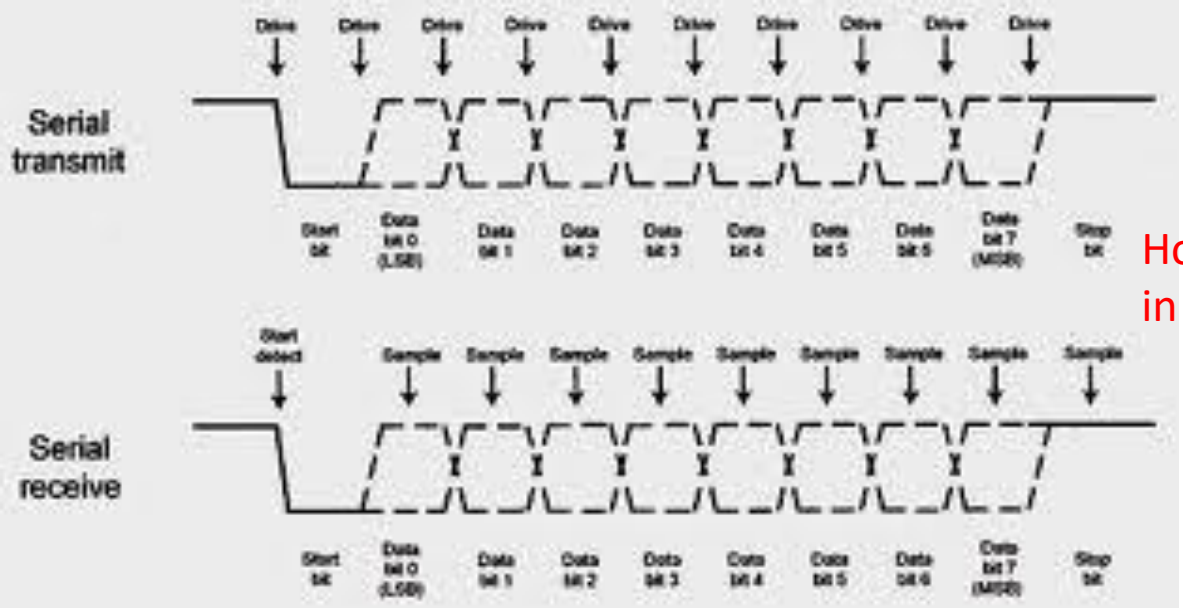
Modeling Data Items

- Data Items
  - Represent the main transaction input to the DUT
  - Adhering to a protocol, consistent values are generated and sent
  - Examples include packets, transactions, instructions, and so on
- Test environment randomizes data items (transactions)
  - A default distribution should be generated
  - Tests further steer generation by layering constraints or selecting from pre-defined scenarios
  - This makes test short, readable, easier to write and maintain



How to Choose an Item

1. Item is an atomic entity that can be driven or monitored on interface, so choose an item which can model all possible transactions and scenarios
2. Choose the most complex (abstract, the biggest) item that satisfies the first requirement.



How would you choose an item in case of UART protocol?

SystemVerilog Data Item: UART Frame

```
class uart_frame;
  rand int delay;
  rand bit start_bit;
  rand bit [7:0] payload;
  rand bit [1:0] stop_bits;
  rand bit [3:0] error_bits;
  bit  parity;

  // utility functions
  extern function bit calc_parity( );
  extern function void print( );
  extern function bit compare(uart_frame rhs);
  …
endclass: uart_frame
```

Fields to be randomized

How will the test-writer request a bad parity frame?

parity is calculated using a function

Constraints

```
typedef enum bit {BAD_PARITY, GOOD_PARITY} parity_e;
class uart_frame;
  rand int delay;
  rand bit start_bit;
  rand bit [7:0] payload;
  rand bit [1:0] stop_bits;
  rand bit [3:0] error_bits;
  bit parity;
  // control fields
  rand parity_e parity_type;

  constraint default_delay { delay  >=  0;
                             delay < 20; }

  constraint default_parity_type {parity_type dist {
                      GOOD_PARITY:=90, BAD_PARITY:=10};}
  // utility functions
   …
endclass: uart_frame
```

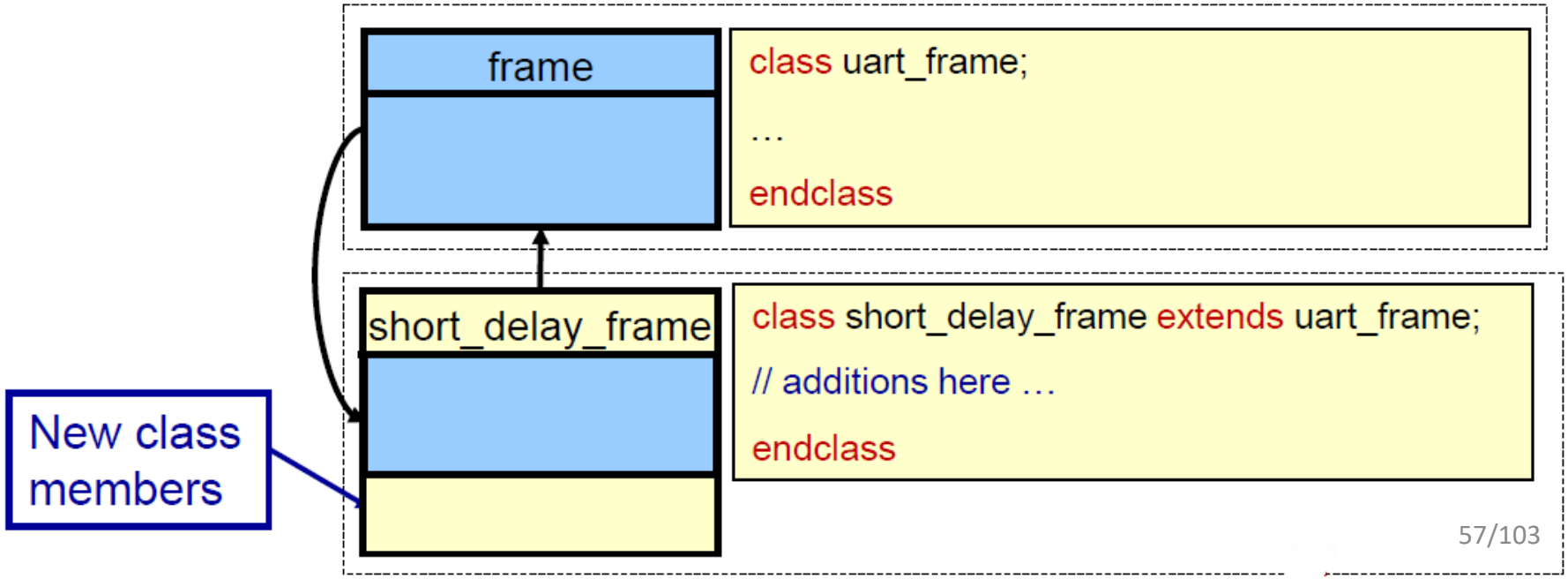
By default we don't want a huge delay

Can you think of another possible default constraint?

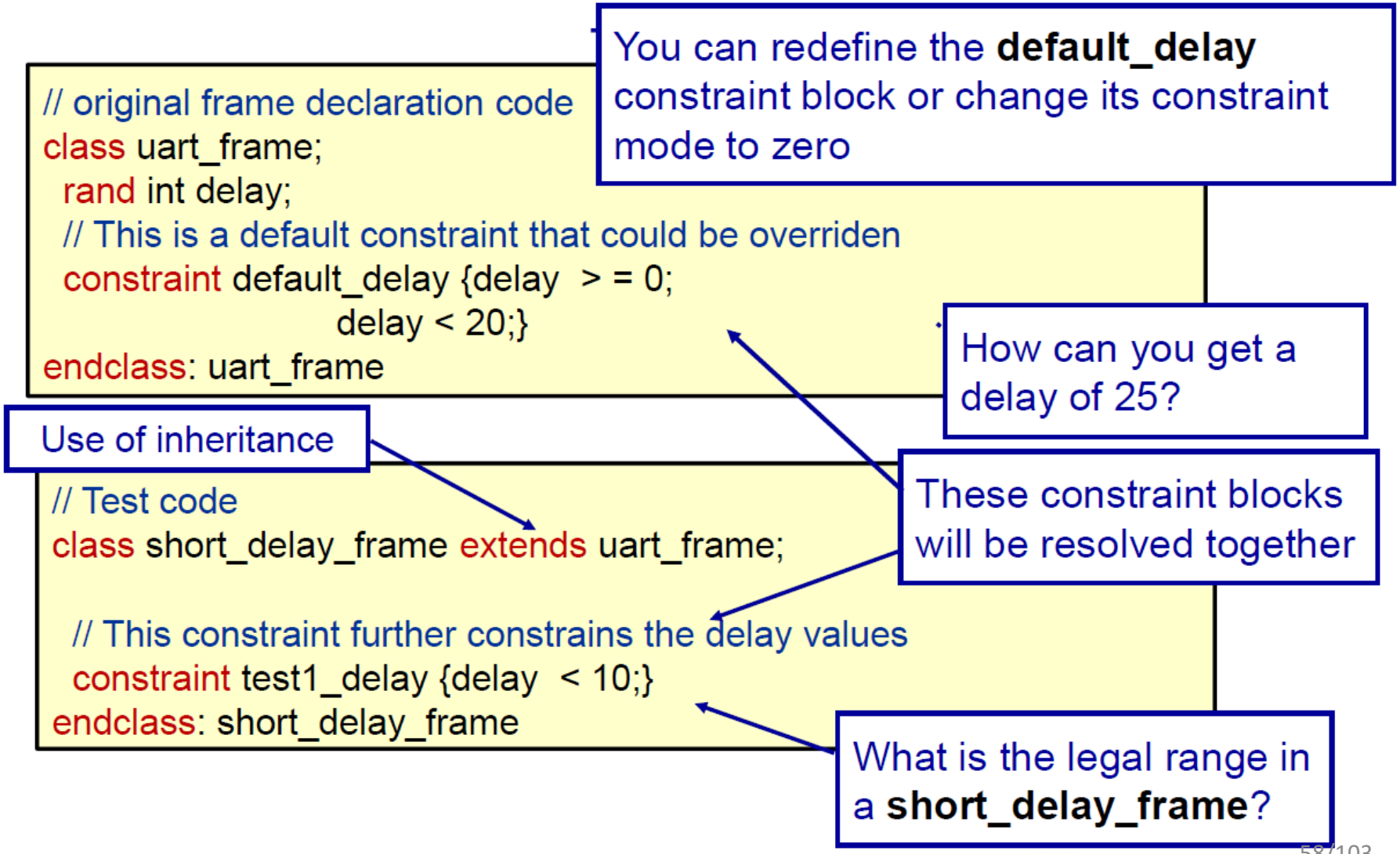
In typical traffic, most frames are legal

Layering Constraints

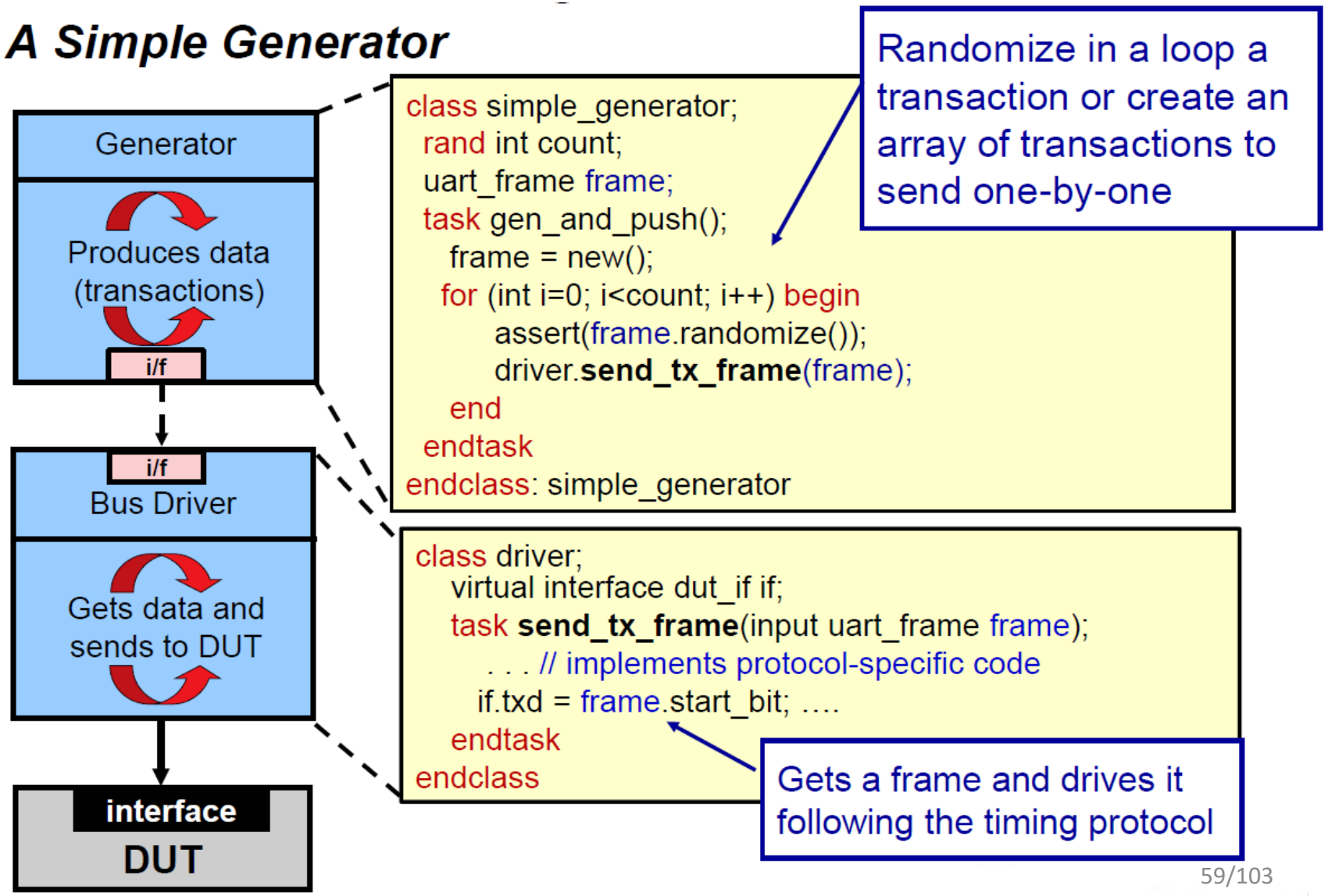
- In a specific test user may want to:
  - Adjust the randomness
  - Disable existing constraints
  - Further change the generation using new constraints
- This is achieved using class inheritance
- Create a new class that inherits from the parent class



Layering Constraints: Example

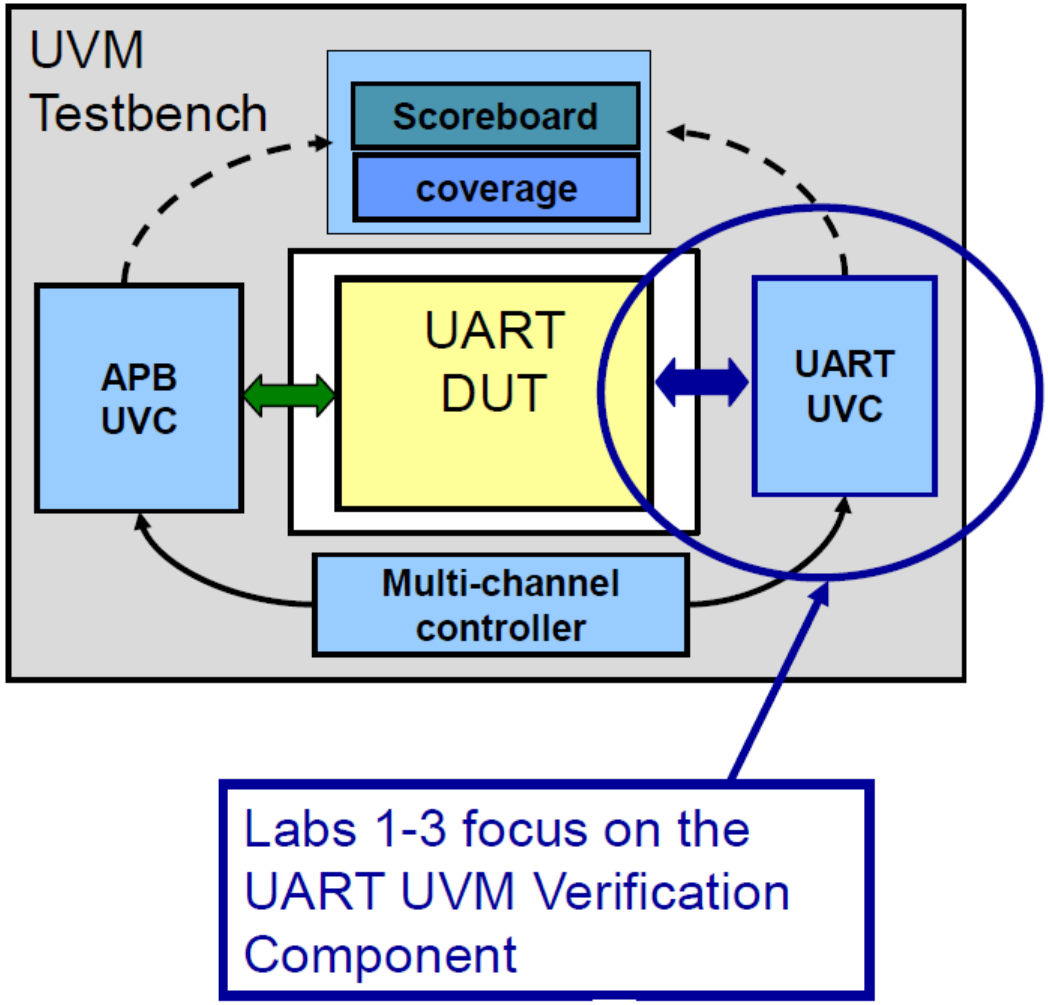


How Are Data Items Generated?



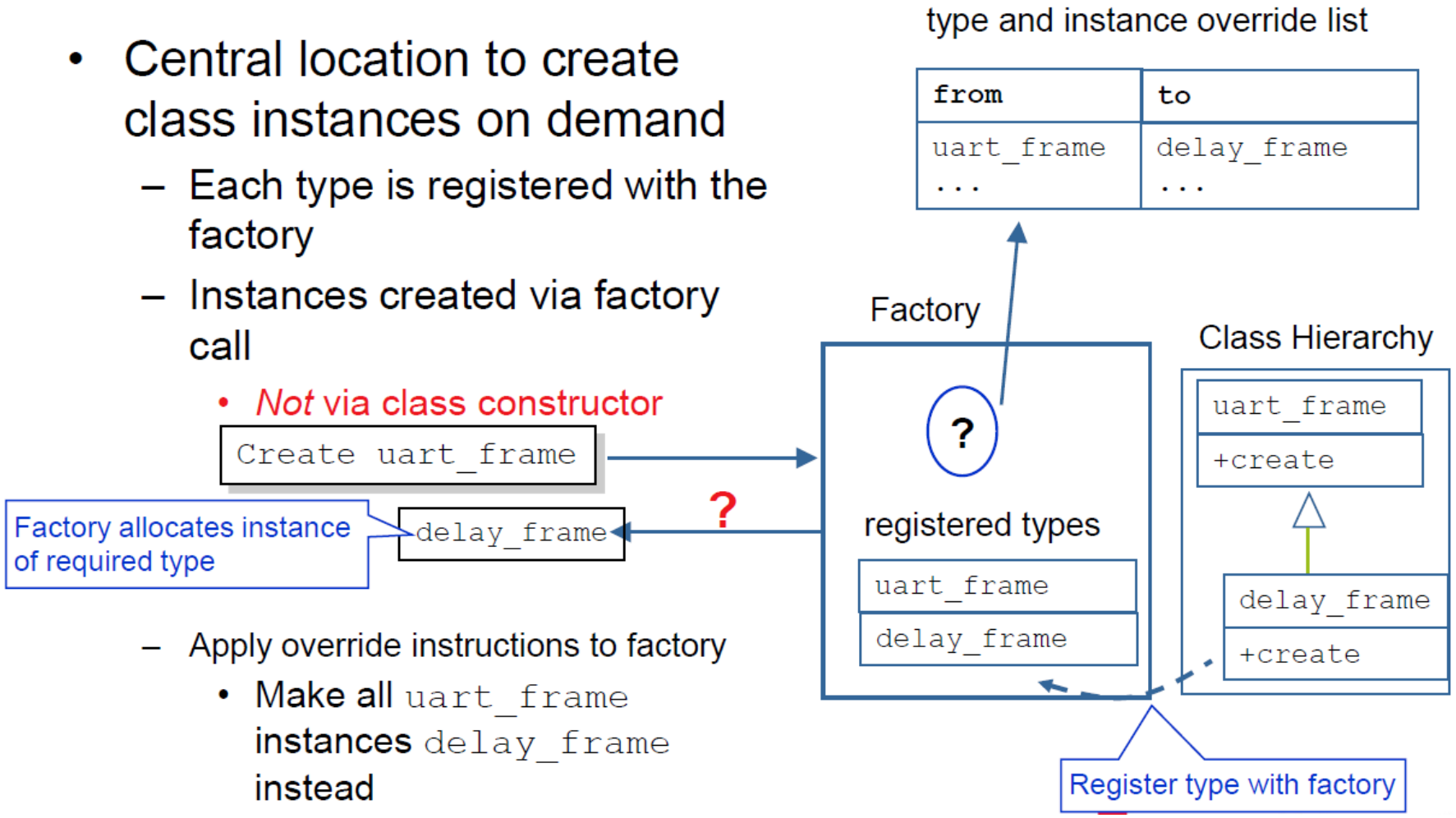
It's Lab Time – Lab 1

- Data item modeling:
  - UART serial data interface – frames
- Objectives:
  - Transaction generation and constraint layering
  - UVM automation
  - Use UVM message facility



What is UVM Factory?

- Central location to create class instances on demand
  - Each type is registered with the factory
  - Instances created via factory call
    - *Not* via class constructor



  - Apply override instructions to factory
    - Make all `uart_frame` instances `delay_frame` instead

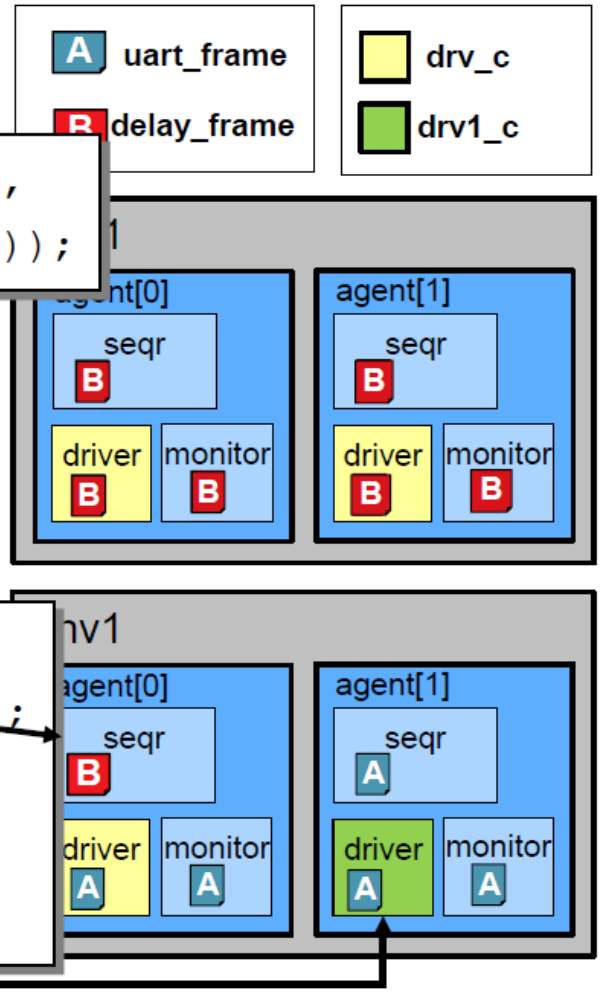
Factory Overrides

- Type override replaces ALL instances:-
  - For a specific test, replace all `uart_frame` packets with `delay_frame` packets

```
set_type_override_by_type(uart_frame::get_type(),
                          delay_frame::get_type());
```

- Instance override replaces specific instances:
  - Replace `uart_frame` in agent[0] sequencer
  - Replace `agent[1]` driver `drv_c` with `drv1_c`

```
set_inst_override_by_type("env1.agent[0].seqr",
  uart_frame::get_type(),delay_frame::get_type());

set_inst_override_by_type("env1.agent[1]",
  drv_c::get_type(), drv1_c::get_type());
```



Using Factory

```
class test1_frame extends uart_frame;
  constraint default payload { payload <= 8'h3f; }
  `uvm_object_utils(test1_frame)
  function new(string name="test1_frame");
     super.new(name);
  endfunction : new
endclass : test1_frame
```

macro registers the object with the factory

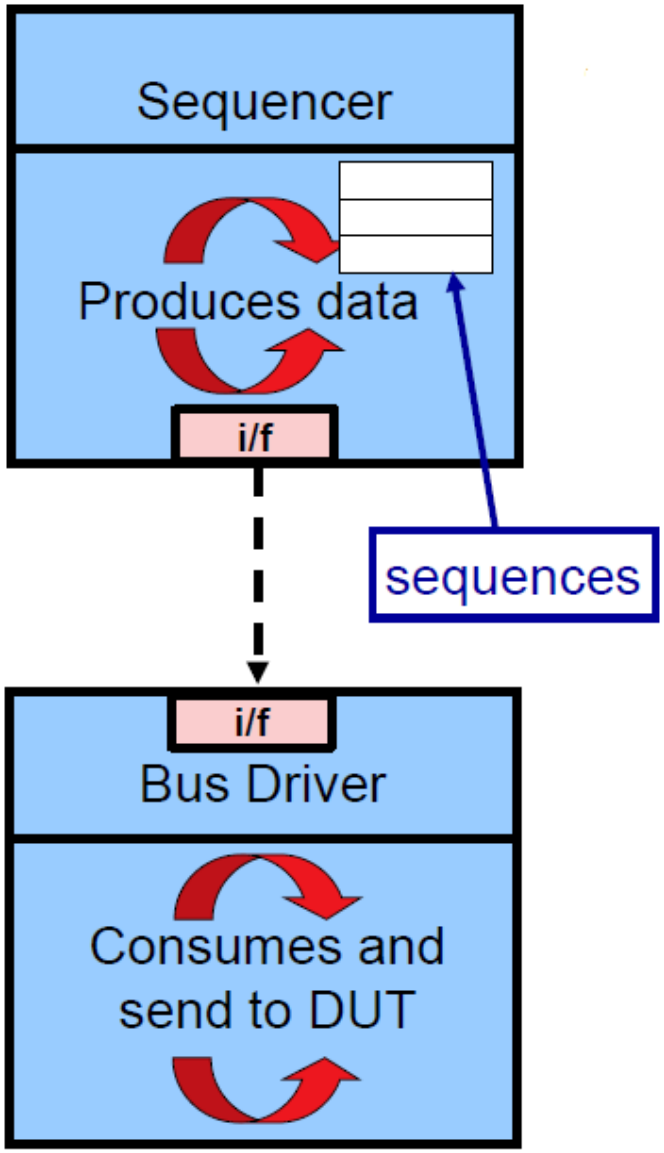
```
class frame_generator;
    uart_frame frame;
    …
    task gen_and_push();
     // replace frame = new(); with the following:
     frame = uart_frame::type_id::create("frame");
     …
  endclass : frame_generator
```

```
module lab1_top;
…
 frame_generator generator;
  …
 initial begin
    uart_frame::type_id::set_type_override (test1_frame::get_type());
    generator = new();
    …
endmodule : lab1_top
```

Scenario Creation: Sequences

- Many times, single items cannot capture high-level intention
  - Ordered streams of transactions are required
  - Examples: configuration of a device, program generation
- A sequence is a set of transactions
  that accomplish a defined task
- Sequences are provided as part of a reusable component
  - They capture important scenarios that should be exercised
- UVM provides an advanced sequence generation mechanism

UVM Sequencer: Advanced Generator



- By default works exactly like a generator
  - Generates random transactions on request
- Addresses all randomization requirements
  - random by-default,
  - reactive generation,
  - system-level ready,
  - modeling timing, etc
- User-defined sequences of transactions are created and executed
  - For example, a user creates a sequence that interleaves legal and illegal frames
- Test writer sets the sequencer to execute a specific sequence

Sequences: Example

```
// Send one BAD_PARITY frame followed by a GOOD_PARITY
//  frame with the same payload
class retry_seq extends uvm_sequence #(uart_frame);
  // uart_frame req;    // built-in data item field
  `uvm_object_utils(retry_seq)

  rand bit [7:0] pload;    // randomizable sequence parameter
  . . .
  virtual task body ( );   // sequence behavior
    `uvm_do_with(req, {payload == pload; parity == BAD_PARITY;} )
    `uvm_do_with(req, {payload == pload; parity==GOOD_PARITY;} )
  endtask : body
endclass
```

extend from uvm_sequence

Data item

Registers the retry_seq with the factory

body() can also include time consuming statements, fork…join, function calls, etc

The ‘uvm_do’ Operation

```
…
virtual task body ();   // sequence behavior
    `uvm_do_with(req,  {payload == pload; parity == BAD_PARITY;} )
endtask
```

Reactive generation

- Translates into:
  - Wait till item is requested
  - Create items using a factory
  - Randomize (with constraints or not)
  - Return the item to the requester
  - Block code execution till item_done()

• Combines procedural and declarative (constraint layering) directives
• Factory support out-of-the-box

Inline constraints for additional control

• Allows modeling timing
• e.g. scenario that calls for

• built-in TLM to allow reuse and support multi-language

New and expert users deploy all key randomization concepts!

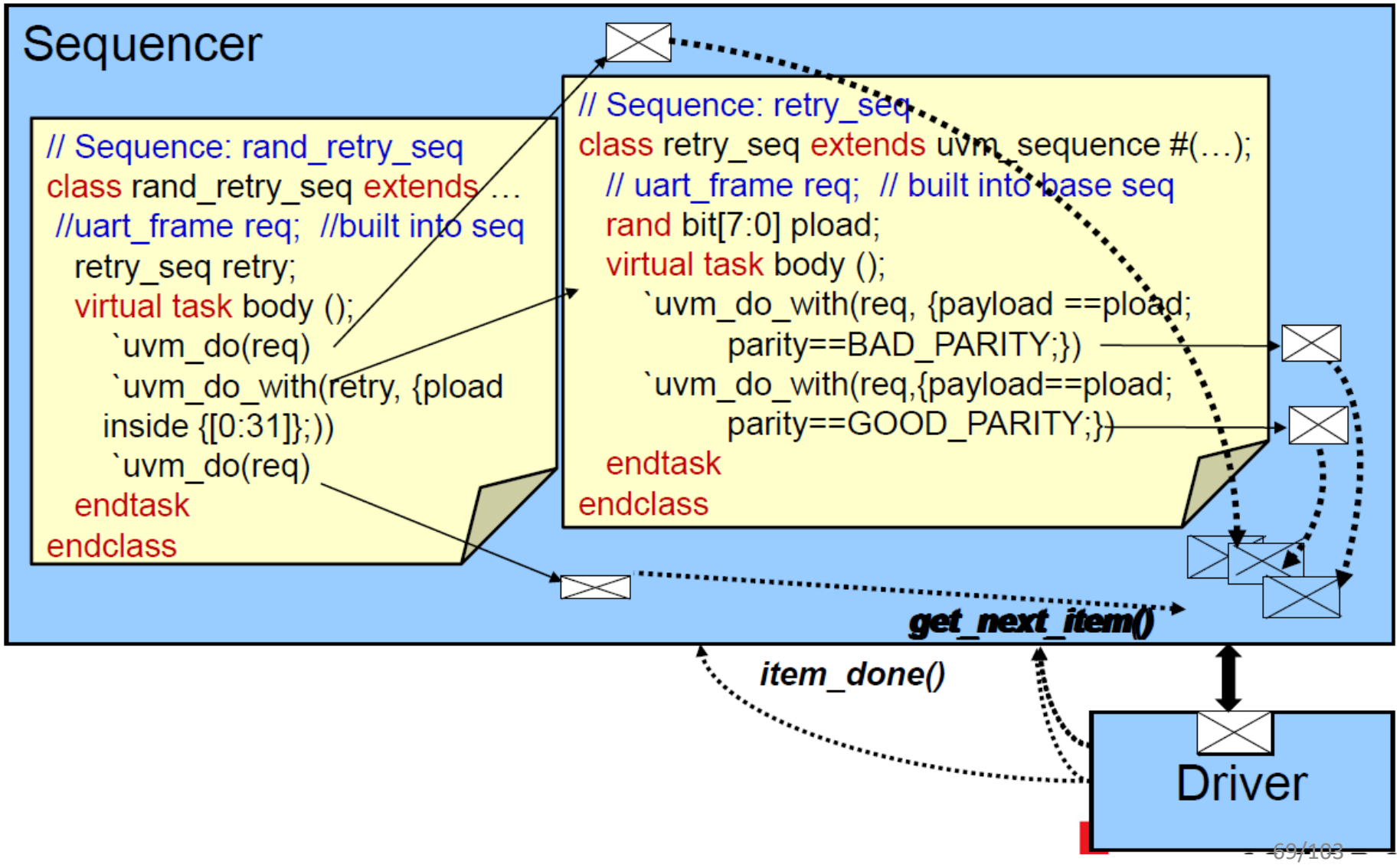
Nesting Sequences

```
// call retry sequence wrapped with random frames
class rand_retry_seq extends uvm_sequence #(uart_frame);
  `uvm_object_utils(rand_retry_seq)

  retry_seq retry_sequence;

  virtual task body (); // executable sequence behavior
    `uvm_do (req)
    `uvm_do_with(retry_sequence , {pload inside {[0:31]};})
    `uvm_do(req)
  endtask
endclass
```

Using a previously defined sequence

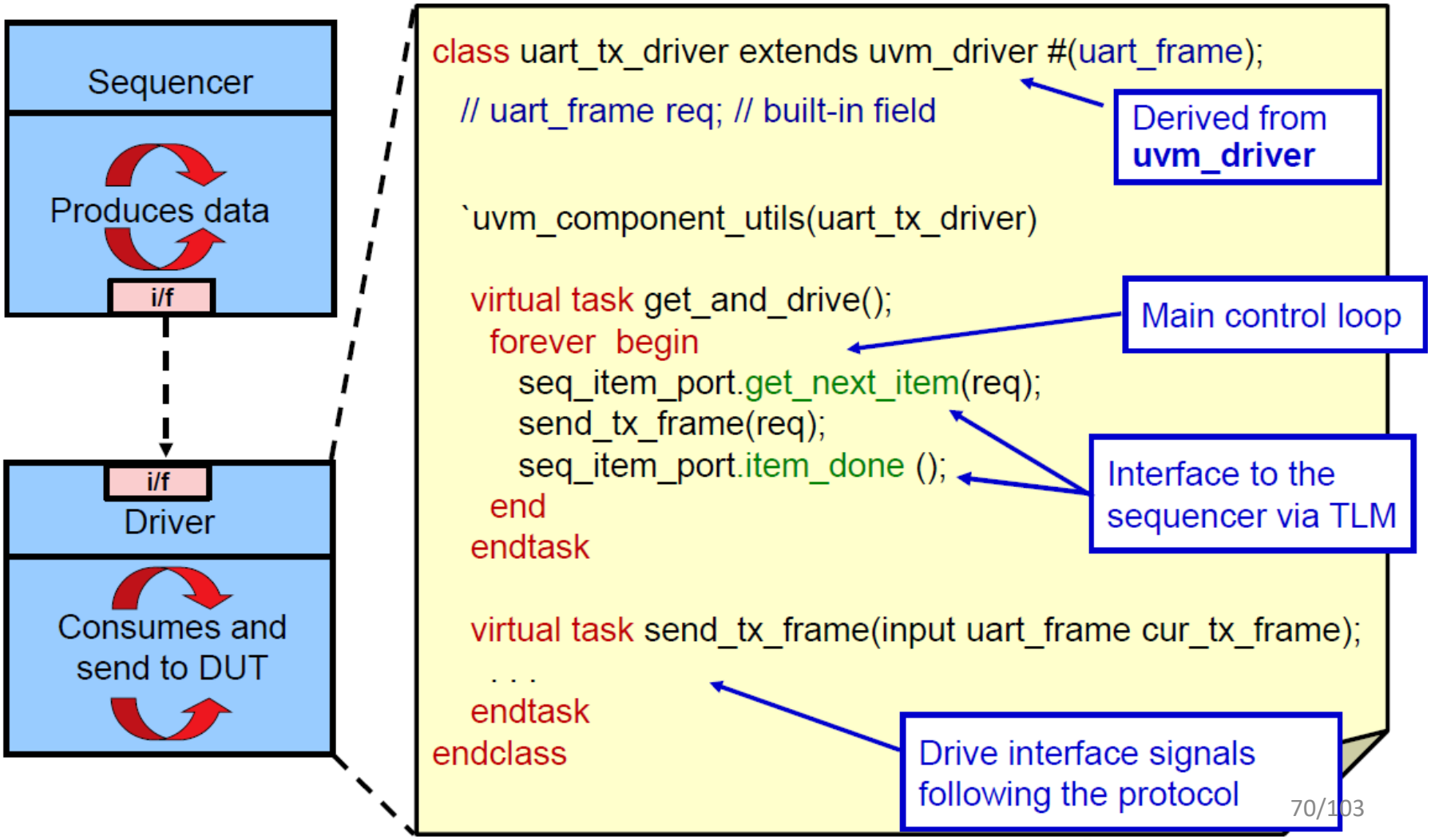
Create and randomize a retry_seq sequence and call it's body

Sequence Execution



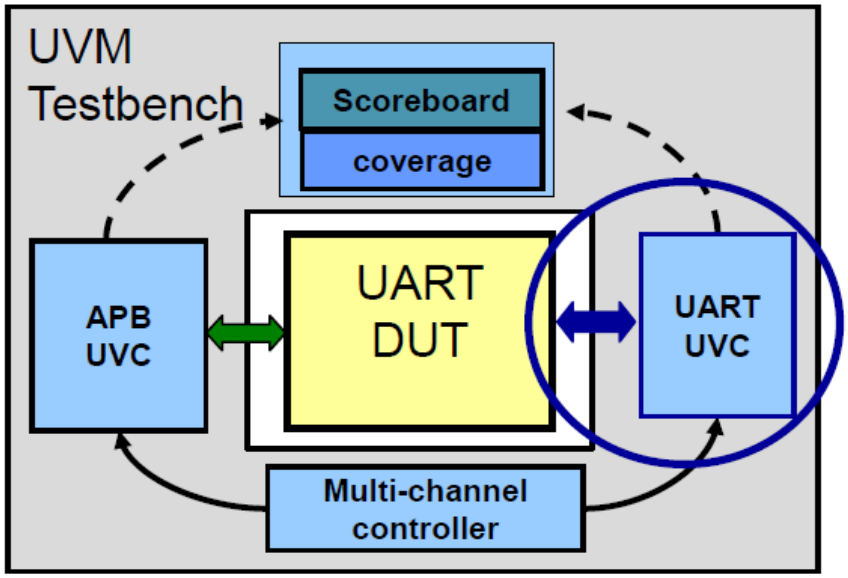
The UVM Driver

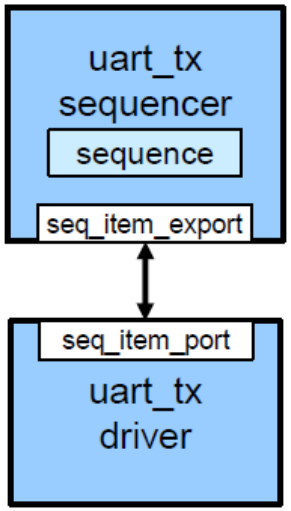
- Pulls the transactions from the sequencer and drives the DUT interface



It’s Lab Time – Lab 2

- Objectives:
  - Explore the driver and sequencer interaction
  - Review the sequencer default behavior
  - Execute a specific sequence
  - Write a new sequence





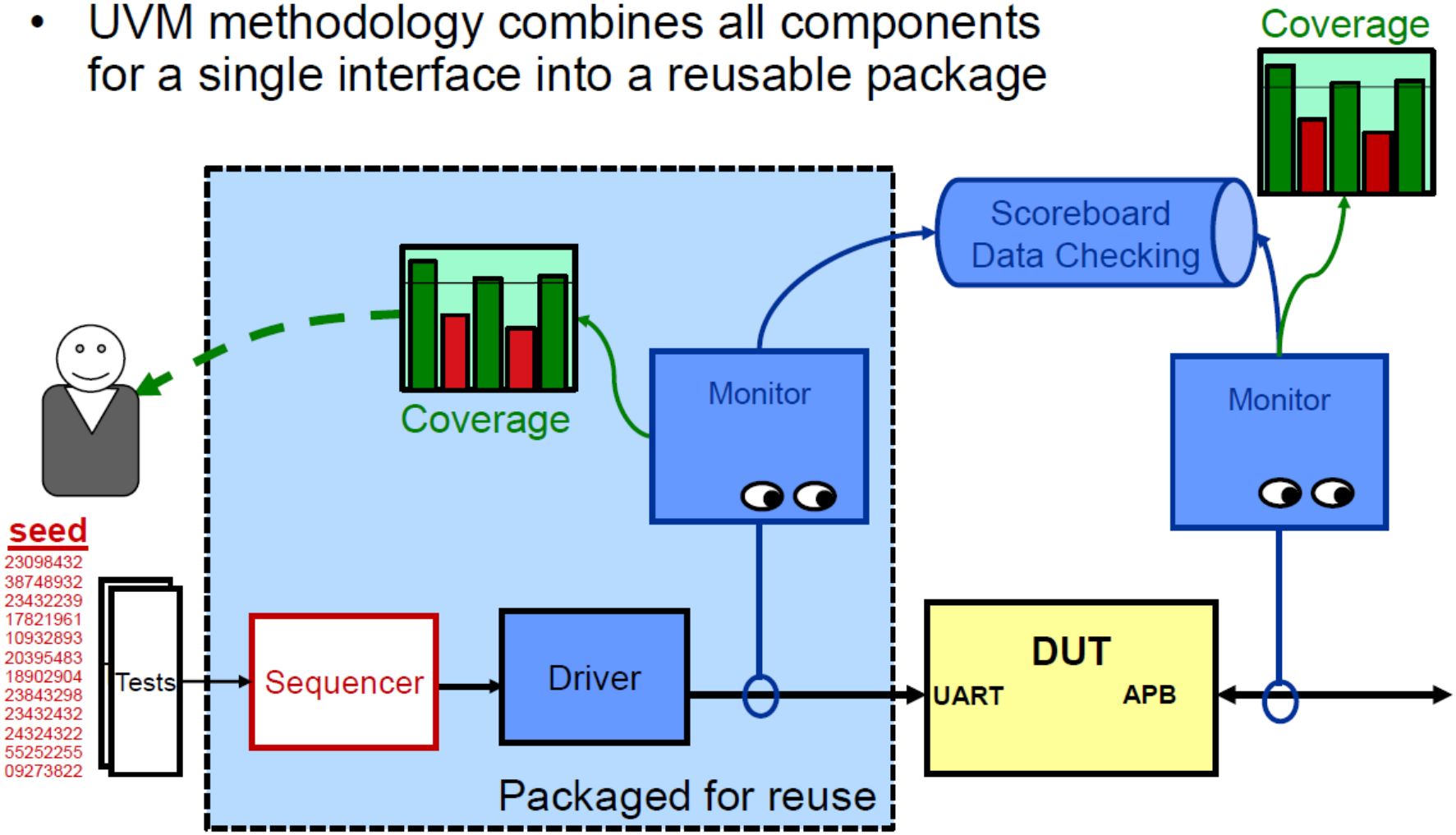
Sequencer:
- Creates and executes “sequences” of transactions
- Sends UART frames to the Driver

Sequence :
- Creates randomized UART frames

Driver :
- Gets UART frames from the Sequencer
- Converts to signal-level details (it just prints the frame in this lab)
- Indicates it’s ready for another frame

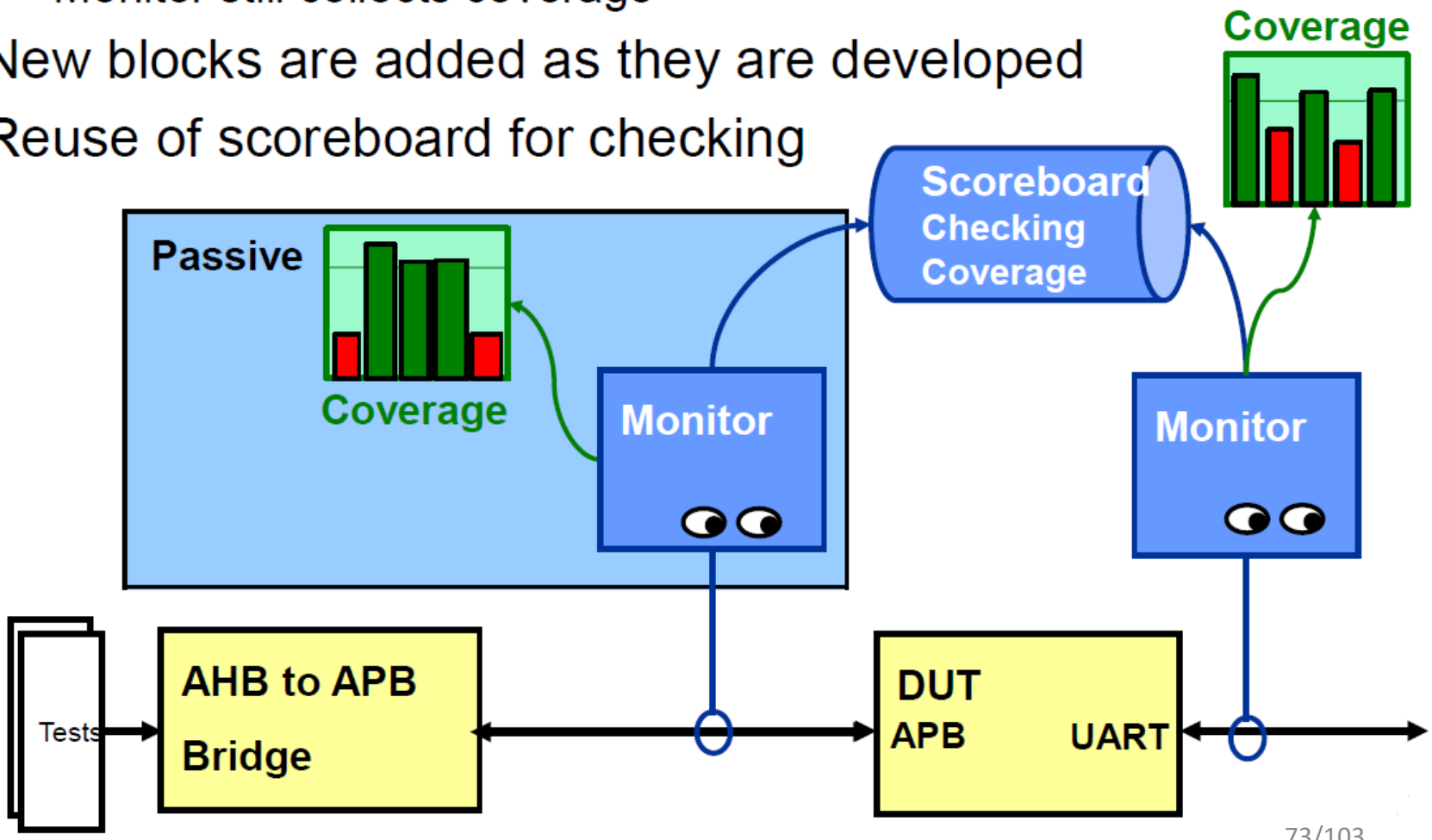
Packaging UVM Components for Reuse

- UVM methodology combines all components for a single interface into a reusable package



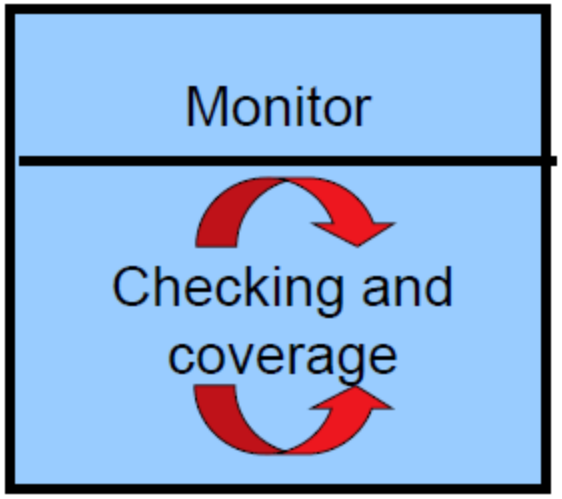
Reuse at the System Level

- Configure APB UVC as passive:
  - Monitor still collects coverage
- New blocks are added as they are developed
- Reuse of scoreboard for checking



Monitor

- Monitor
  - A passive component that collects information on a monitored interface
  - Builds abstract transactions
  - Contains events, status, checkers and coverage
- Protocol specific
- Passive entity
  - Never drives signals!
- Monitor is independent of the driver!
  - May need to monitor information driven by the testbench



UVM Monitor: Example

```
class uart_monitor extends uvm_monitor;
  `uvm_component_utils(uart_monitor)
   uart_frame  frame;     // data item to be collected

  covergroup uart_trans_frame_cg;
    NUM_STOP_BITS : coverpoint  frame.nbstop {
                bins ONE = {0};
                bins TWO = {1};
                }
    DATA_LENGTH : coverpoint  frame.char_length { …}
    PARITY_MODE : coverpoint  frame.parity_mode { … }
  endgroup
   function new (string name = "", uvm_component parent = null);
       super.new(name, parent);
       uart_trans_frame_cg = new();
   endfunction: new
   task collect_frame ( );
     // Collect uart data from interface
     uart_trans_frame_cg.sample();
   endtask: collect_frame
endclass: uart_monitor
```

Derived from uvm_monitor and registered with the factory

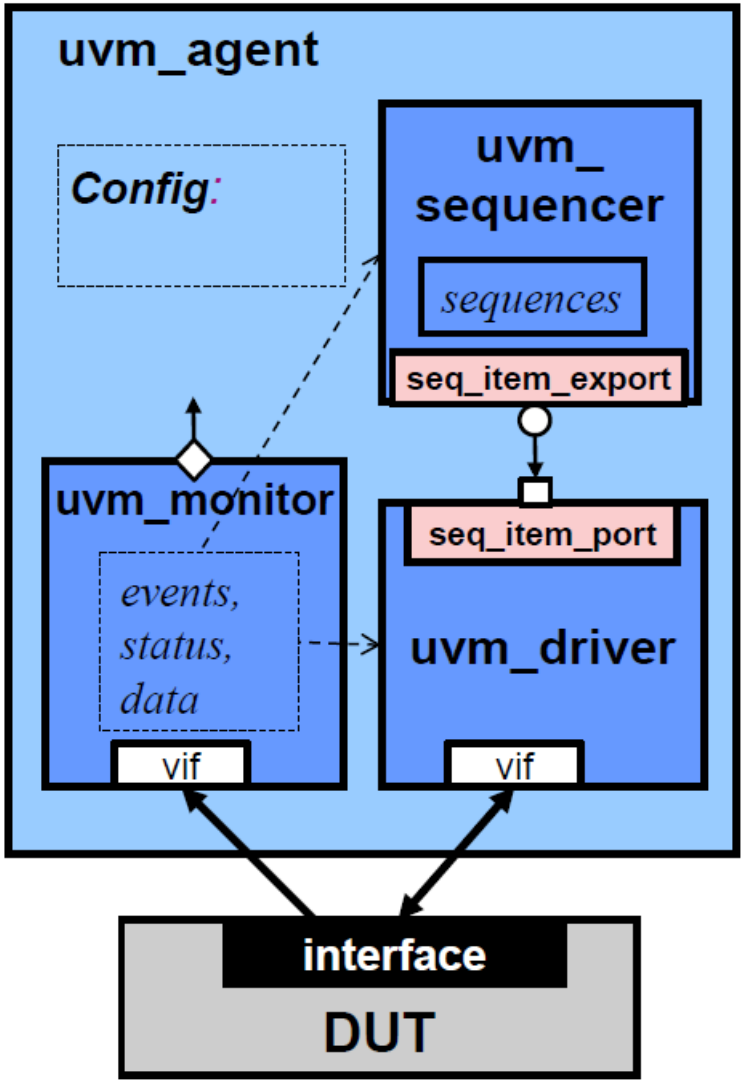
Implement coverage model based on your verification plan

new() covergroup in the constructor

Task to collect data items from the DUT interface

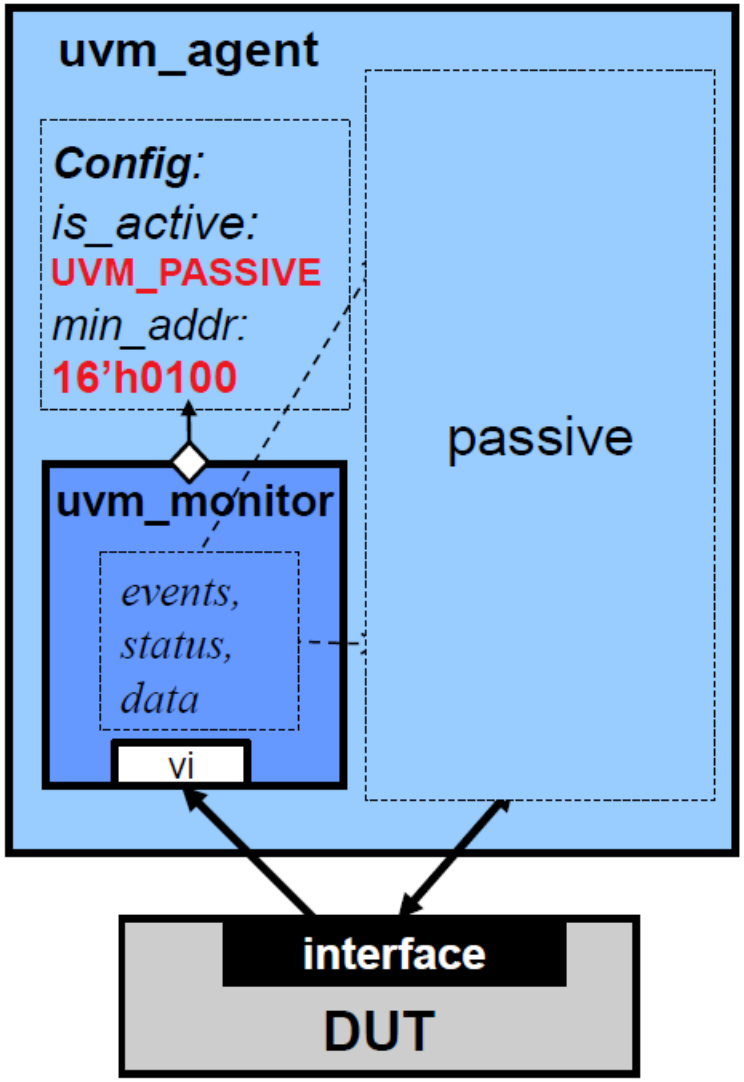
Sample coverage when frame is collected

Agent



- Agent provides all the verification logic for a device in the system
- A standard agent has
  - Sequencer for generating traffic
  - Driver to drive the DUT
  - Monitor
- Note that the monitor is independent of the driver

Agent Configuration



- A standard agent is configured using an enumeration field: **is_active**
- UVM_ACTIVE
  - Actively drive an interface
  - Driver, Sequencer and Monitor are allocated
- UVM_PASSIVE
  - Only the Monitor is allocated
  - Still able to do the checking and collect coverage
- Other user defined configuration parameters can also be added

Agent build_phase() and connect_phase()

```
class master_agent extends uvm_agent;
  . . .
  // agent sub component instances
  master_driver        driver;
  master_sequencer  sequencer;
  master_monitor      monitor;
  . . .
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(uvm_phase phase);
  // create sequencer, driver and monitor
    monitor = master_monitor::type_id::create( "monitor", this);
    if (is_active == UVM_ACTIVE)  begin
      driver = master_driver::type_id:: create("driver", this);
      sequencer = master_sequencer::type_id::create("sequencer", this);
    end
  endfunction
  virtual function void connect_phase(uvm_phase phase);
      driver.seq_item_port.connect(sequencer.seq_item_export);
   endfunction
endclass
```

```
class v2_mdriver extends master_driver;
   // add attributes + override virtual methods
endclass
// syntax for introducing the new driver – in test
master_driver::type_id::set_type_override (
     v2_mdriver::get_type());
```

Use the factory to override the driver for a specific test

Call super.build_phase(…)

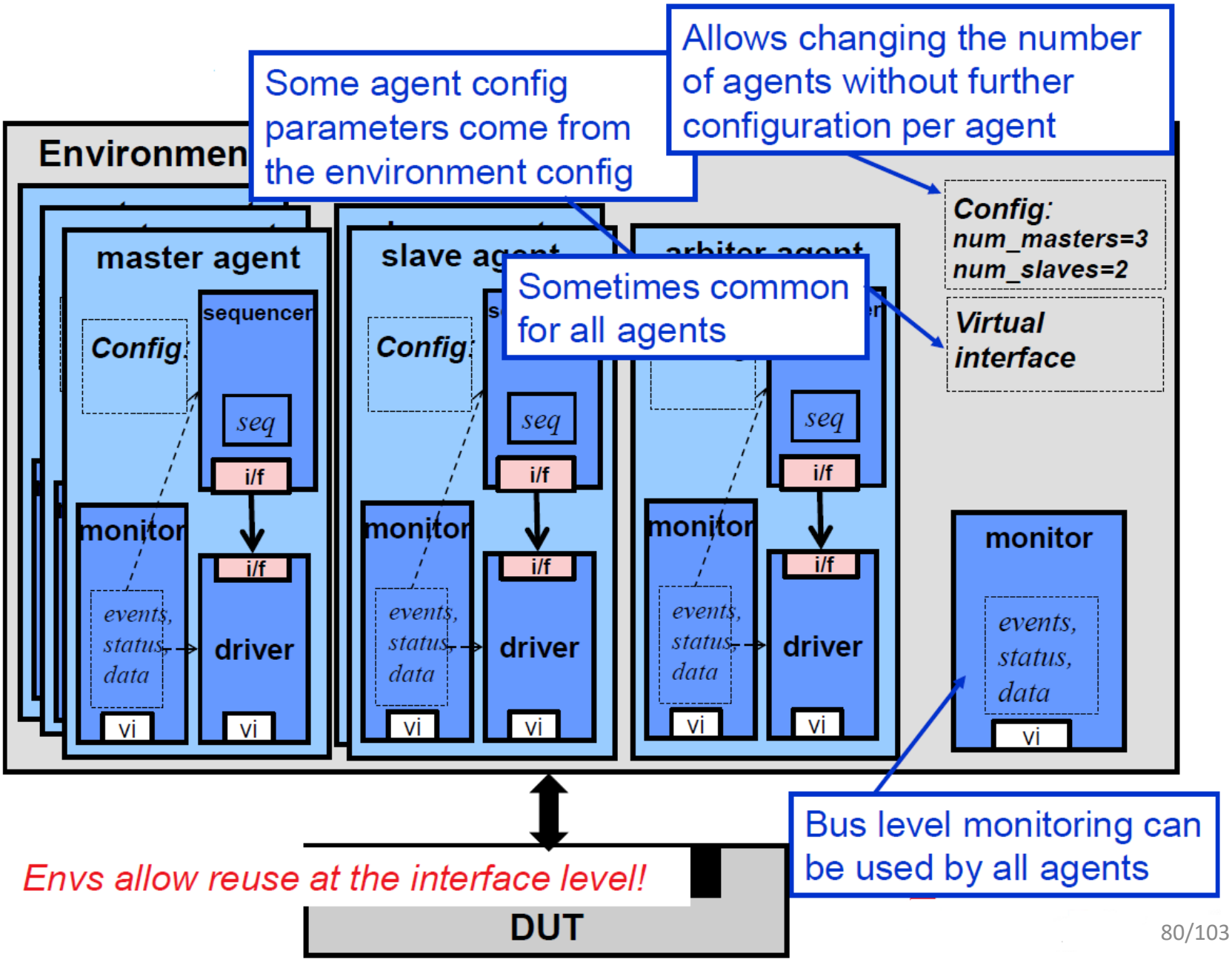
Use **create()** for factory allocation

Standard TLM mechanism for connecting the driver and sequencer

Verification Environment

- Many protocols have a variable number of devices
- These are represented as agents in UVM
  - Agents share a common configuration or common signals
    Example: bus speed
  - Agents communicate with each other
    directly or via a central component
    Example: A single bus monitor can serve multiple agents
- Environment classes (envs) are introduced
  to encapsulate and configure multiple agents
  - Contain all the reusable logic for an interface
  - Also referred to as UVM Verification Components (UVCs)
  - Usually have two or more agents

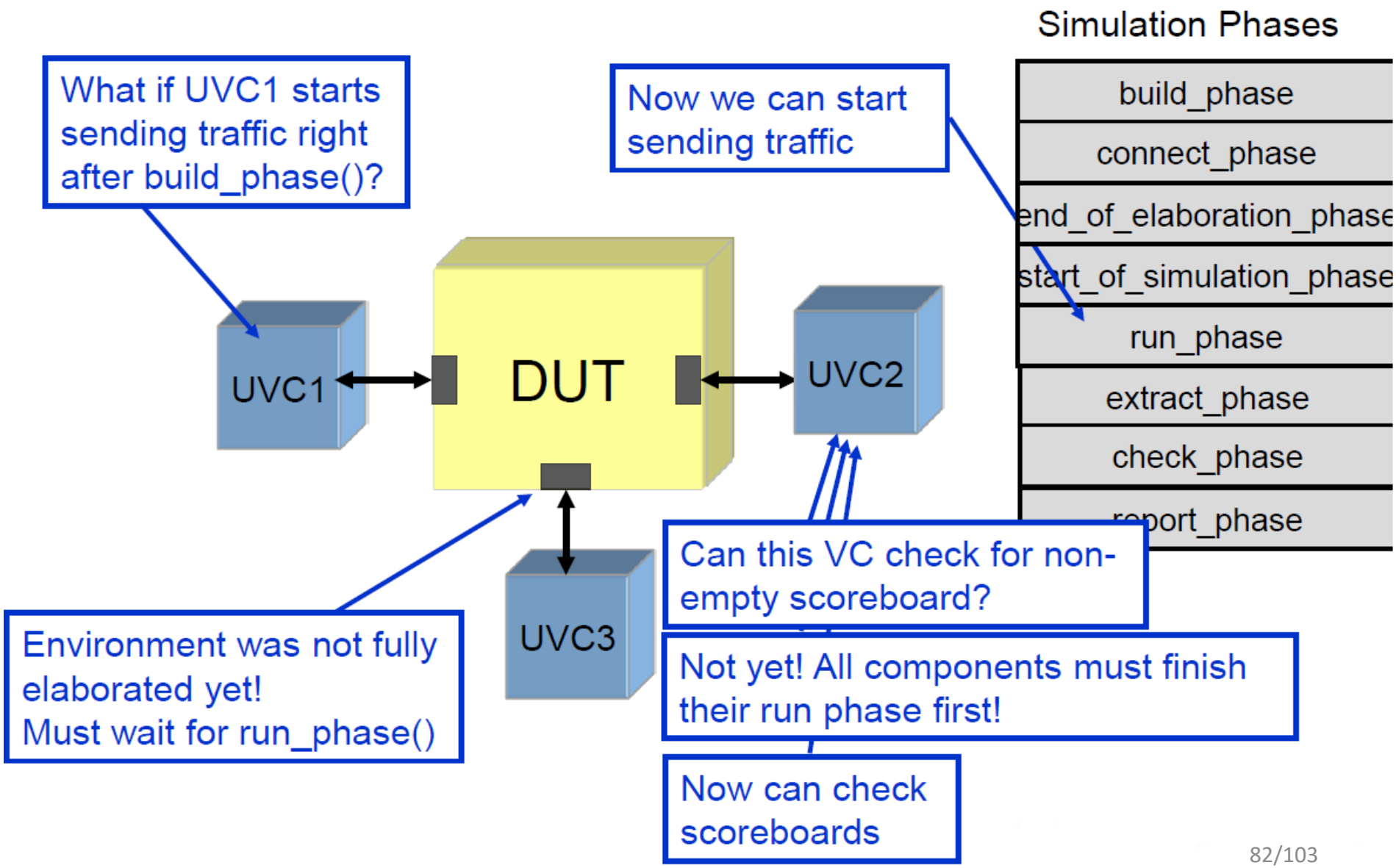
Environment Example



Simulation Phases

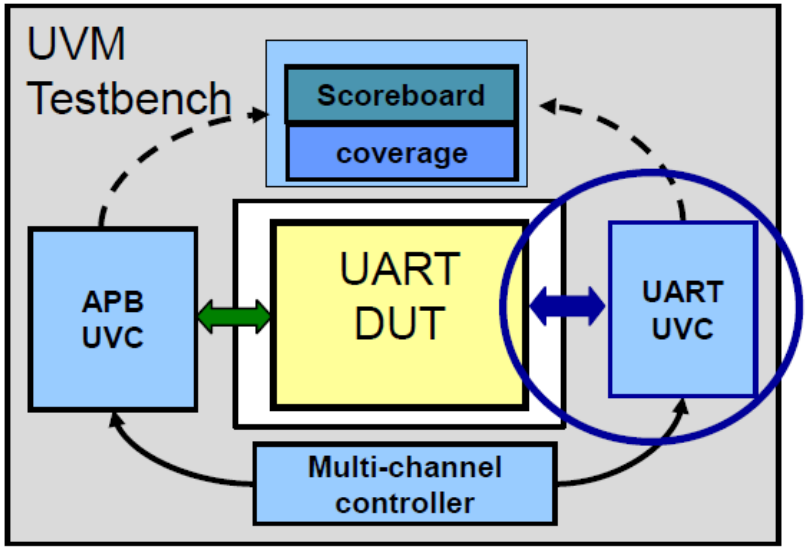
- When moving to classes you need to manage environment creation at run-time
- Test execution is divided to phases
  - Configuration, testbench creation, run-time, check, etc
- Unique tasks are performed in each simulation phase
  - Set-up activities may be performed during “testbench creation” while expected results may be addressed in “check”
  - Phases run in order – next phase does not begin until previous phase is complete
- A set of standard phases enables VIP plug & play

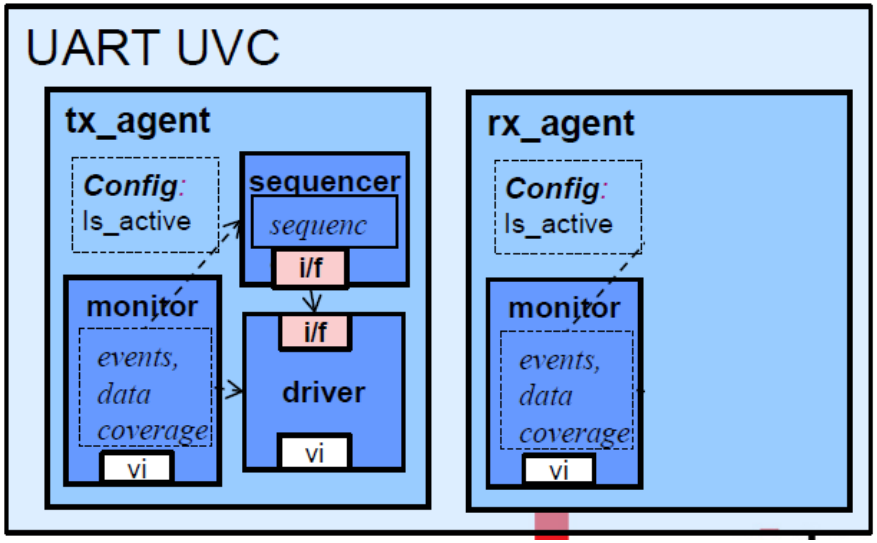
Simulation Phases



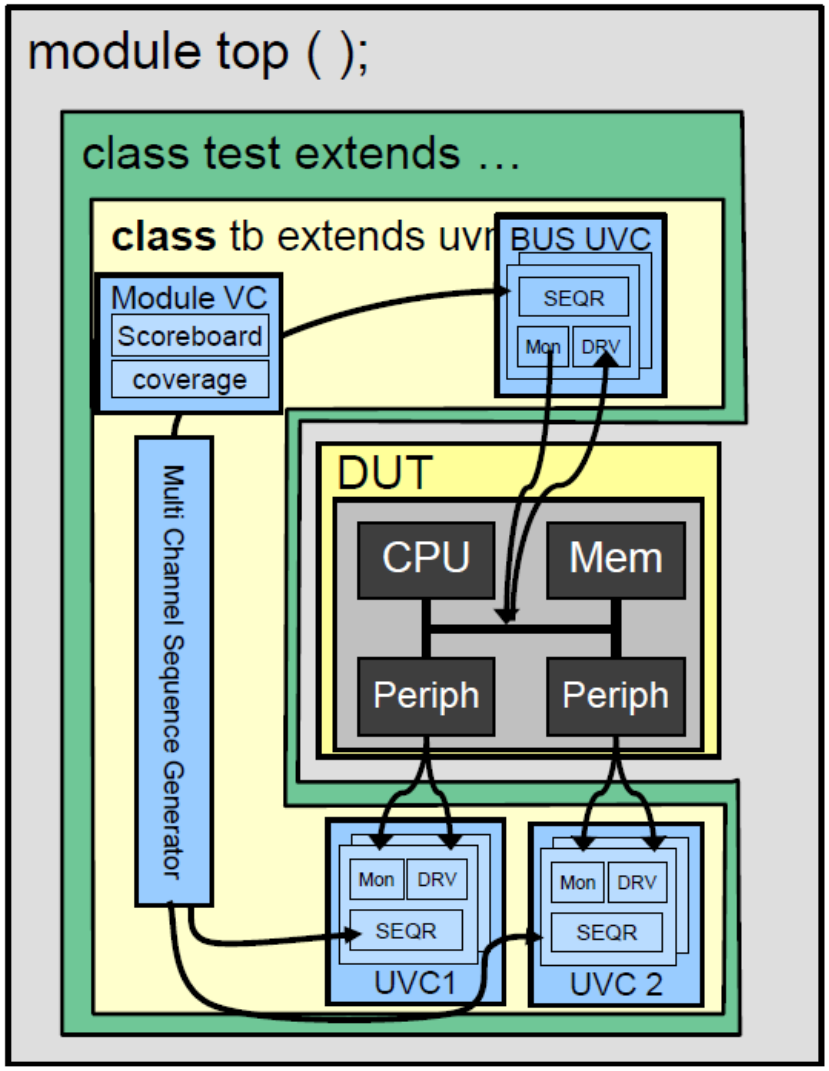
It's Lab Time – Lab 3

- Reusable Environment Topology:
- Objectives:
  - Review the UVM Verification Component architecture: sequencer, driver, monitor, agent, env
  - Understand how to make an agent PASSIVE





Testbench Environment



- Separate the env configuration and the test
  - TB class instantiates and configures reusable components
- Specific tests are built on top of testbenches (tbs)
  - Specify the nature of generated traffic
  - Can modify configuration parameters as needed
- Benefits:
  - Tests are shorter, and descriptive
  - Less knowledge to create a test
  - Easier to maintain – changes are done in a central location

Testbench Example

```
class uart_ctrl_tb extends uvm_env;
  `uvm_component_utils(uart_ctrl_tb)

 apb_env apb0;    // APB UVC
  uart_env uart0;    // UART UVC
  uart_ctrl_env uart_ctrl0;  // Module UVC

  virtual function void build();
       super.build();
       uvm_config_db#(uvm_bitstream_t)::set(this,"a
    "is_active", UVM_ACTIVE);
       uvm_config_db#(uvm_bitstream_t)::set(this,"uart0.tx", "is_active",
    UVM_ACTIVE);
       uvm_config_db#(uart_ctrl_config)::set(this,"uart_ct
       apb0 = apb_env::type_id::create("apb0", this);
       uart0 = uart_env::type_id::create("uart0", this);
       uart_ctrl0 = uart_ctrl_env::type_id::create("uart_ctrl0", this);
    endfunction
endclass
```

Extends from uvm_env

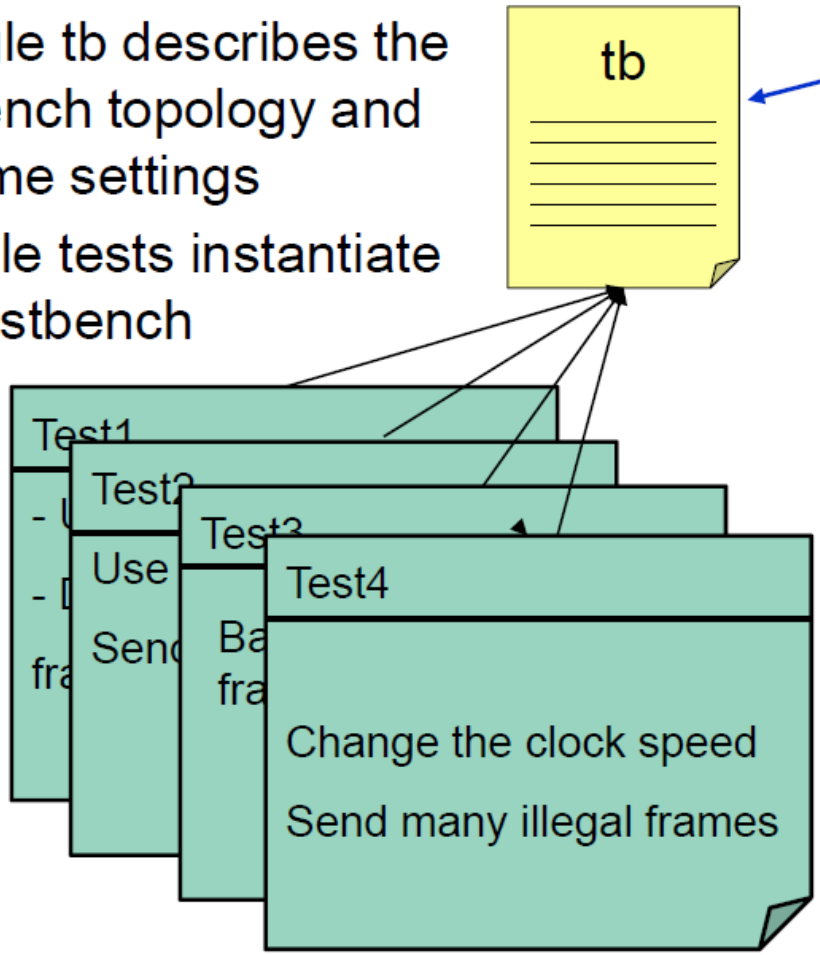
Configure using wildcards

Minimal effort and component knowledge required

Create and build using standard mechanism

Using a Testbench for Multiple Tests

- A single tb describes the testbench topology and run-time settings
- Multiple tests instantiate this testbench



- Tests determine the nature of the generated traffic

How a Test uses a Testbench

```
class apb_uart_rx_tx extends uvm_test;
 `uvm_component_utils(apb_uart_rx_tx)

 uart_ctrl_tb  uart_ctrl_tb0;    //testbench
virtual function void build_phase(uvm_phase pha
   super.build_phase(phase);
   // Set the default sequence for master
uvm_config_db#(uvm_object_wrapper)::set(this,“uart_ctrl_tb0.apb0.master.
     sequencer.run_phase",“default_sequence",
     "read_modify_write::type_id::get()");
   //Construct  and build the uart_ctrl_tb0 environment
   uart_ctrl_tb0 = uart_ctrl_tb::type_id::create(“uart_ctrl_tb0”, this);
 endfunction: build_phase

virtual task run_phase(uvm_phase phase);
   uvm_top.print_topology();
    …
  endtask
endclass: apb_uart_rx_tx
```

Extends from uvm_test

Configuration information specific for this test

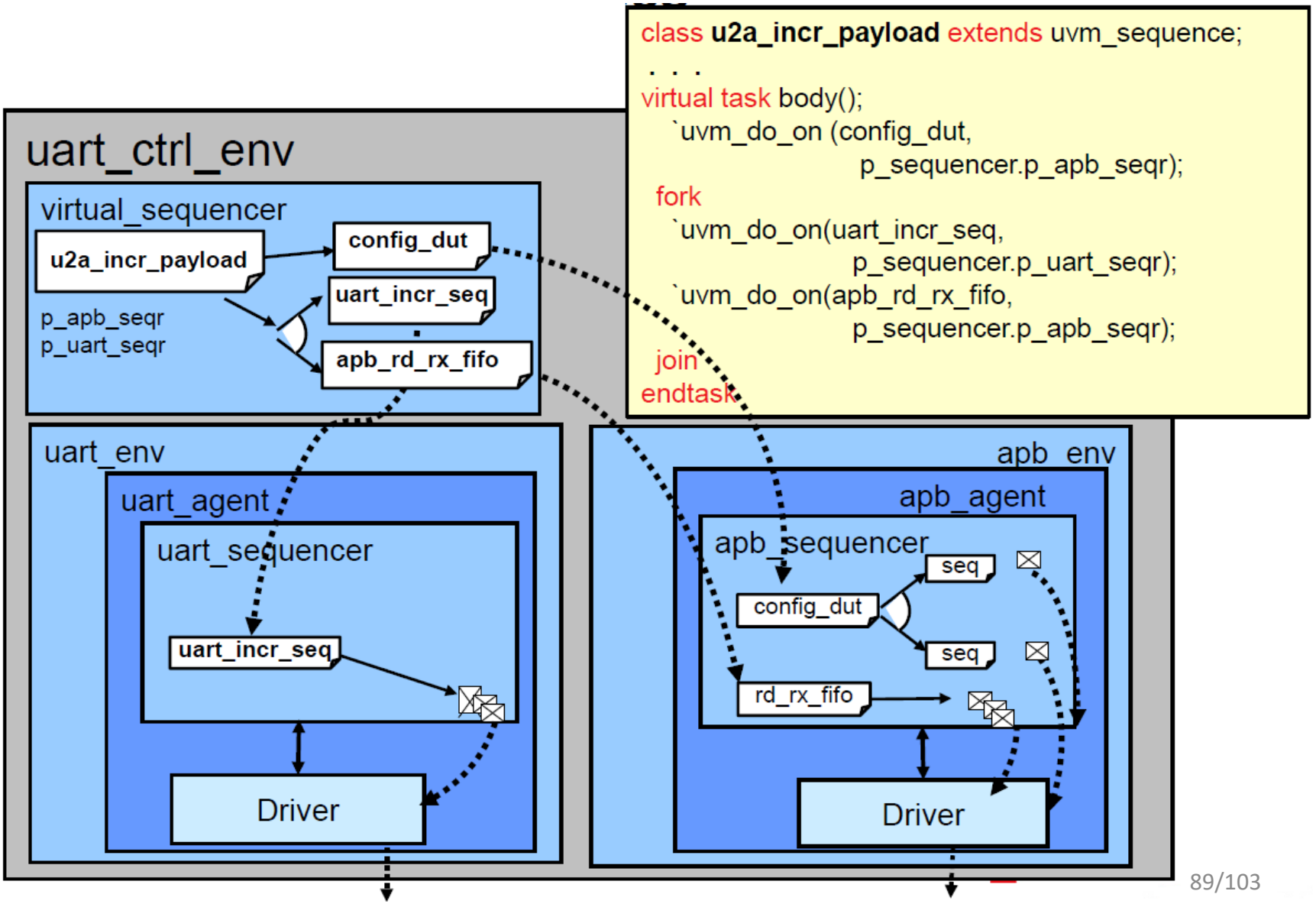
Test creates and builds the testbench

run() task is optionally used for run-time test control or status. e.g., print dynamic testbench topology

Virtual Sequences

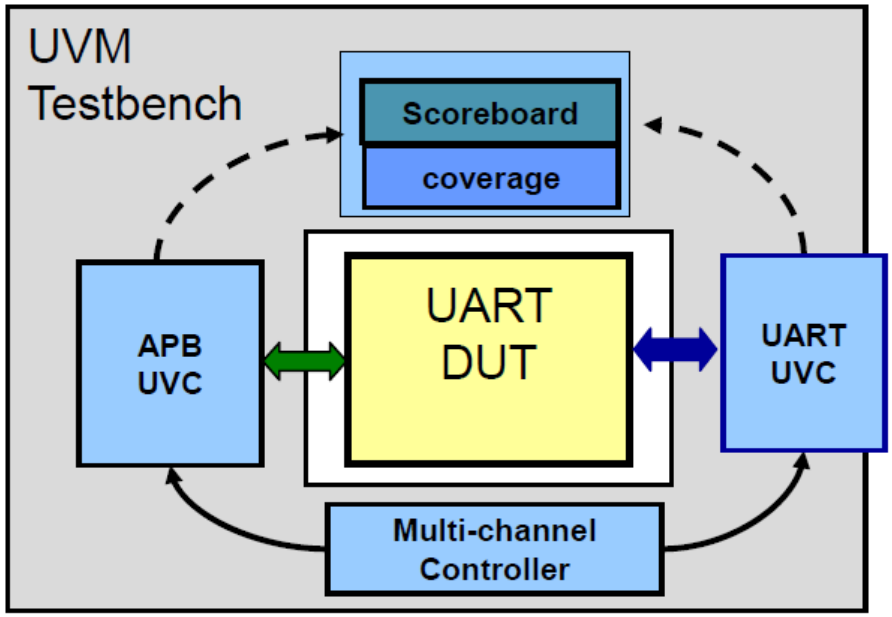
- So far, we have seen how to control stimulus for a single channel (interface)
  - Create a sequencer
  - Create and register sequences
- In a verification environment, we need to coordinate traffic on multiple interfaces in parallel
  - Coordinate data and time
  - “After finishing configuring the device send Ethernet traffic”
- Multi-channel sequences are called virtual sequences
  - Single procedural thread to control multiple interfaces
  - Can create reusable system-level sequences

Virtual Sequences: Example



It’s Lab Time – Lab 4

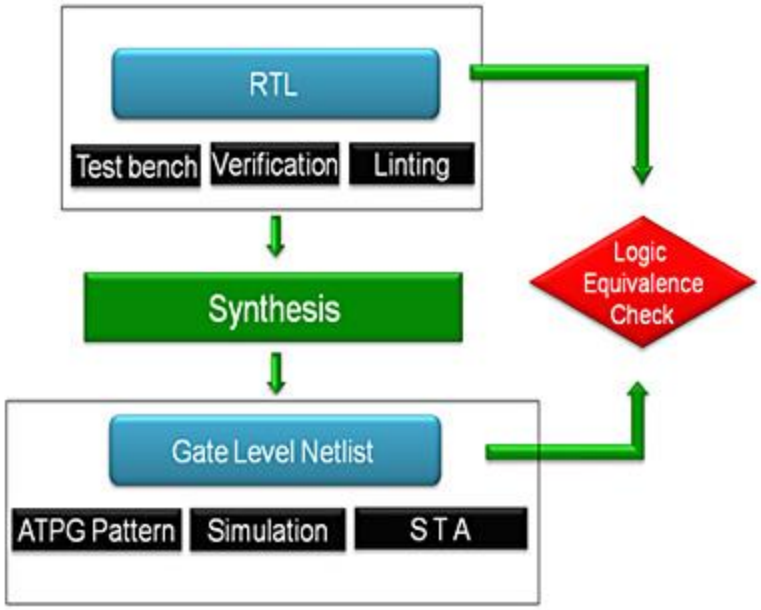
- Test Creation Using Reusable Components
- Objectives:
  - Examine test creation and control using the UART tb
  - Examine an existing test and observe how it instantiates an environment
  - Control environment behavior from the test
  - Show how to control test execution and execute multiple tests without recompiling/re-elaborating the design



VERIFICATION TECHNIQUES

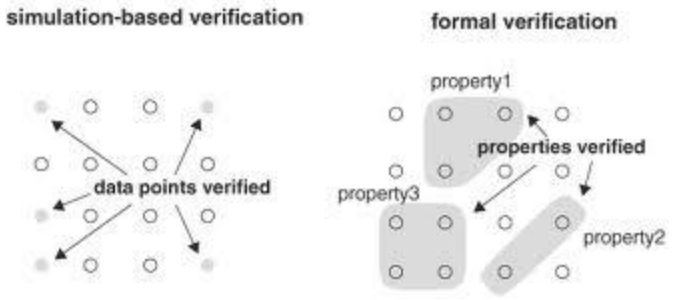
Gate Level Simulations

- Synthesis is the process of taking a design written in a hardware description language, such as VHDL, and compiling it into a netlist of interconnected gates
- We run the simulations on netlist because that is the only way to catch some types of bugs (synchronization…)
- GLS differ from RTL simulations because the timings (delays) in GLS are realistic, while in RTL delays are zero.
- These timings are defined in SDF – standard delay format file



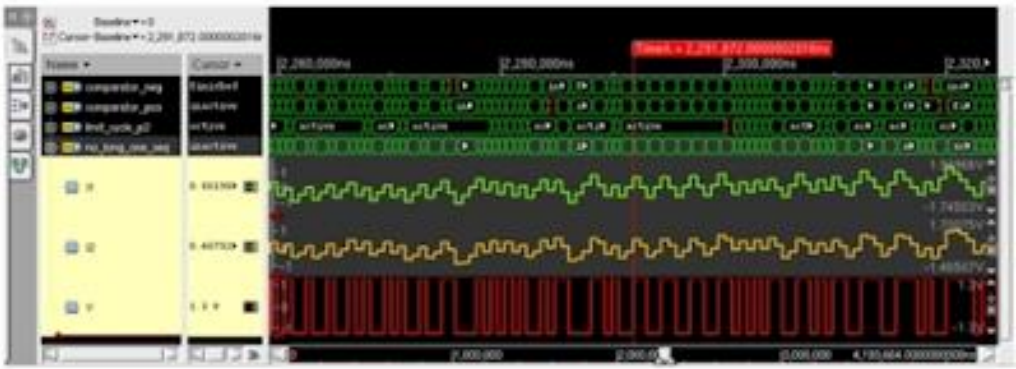
Formal Verification

- Formal verification is the process of proving that the design is functionally correct
  with the use of mathematical techniques.
- Simulation vs. Formal = Dynamic vs. Static
- In simulation we generate input vectors and check the output.
- In formal verification we define the properties of design, and formal tool tries to prove it or disprove it.
- Limitations of formal verification:
  - Only possible for small designs
  - Still not possible to completely define all functionality formally.



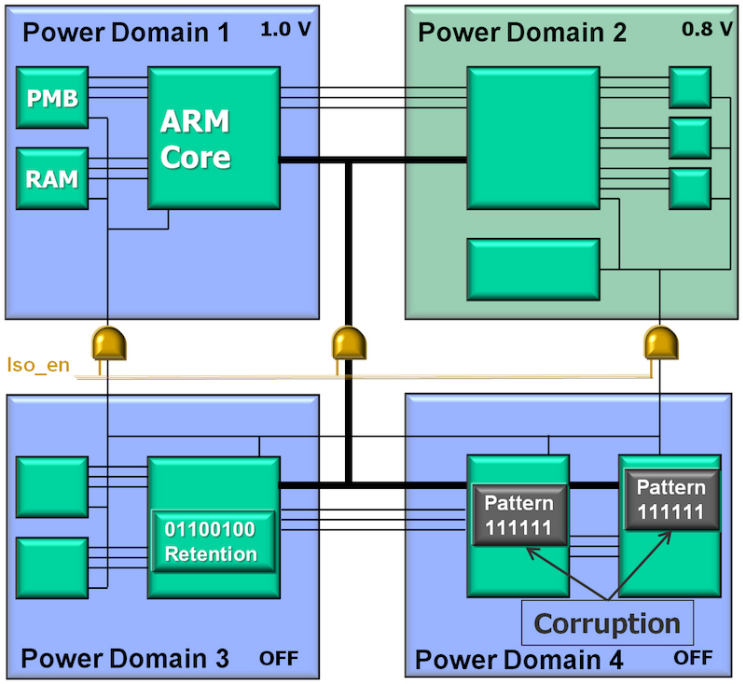
Mixed Signal Verification

- Modern SoCs (System on Chips) are built from both digital and analog circuits.
- Simulation of both types of circuits together is necessary to ensure the correctness.
- There are several types of mixed signal verification:
  - Digital model - fastest and least precise
    Analog circuits are modeled as digital circuits
  - Analog model – slower and more precise
    Analog circuits are modeled with special languages (ex. Verilog AMS)
  - SPICE –
    The most precise simulation of analog circuits,
    and the most time consuming



Low Power Verification

- In real world mobile applications,
  it is essential to save power in parts of the chip that are not in use.
- Techniques to save power:
  - Power domains – divide the system in blocks which can be shut off independently
  - Retention – memory and register values
    can be restored after shutdown.
  - Isolation cells are put between blocks
    so that signals from powered-down block
    does not corrupt a powered-up block
  - Level shifters – cells that transform voltage
    from different voltage domains
- UPF – Unified Power Format
  File which specifies power intent –
  all of the above techniques
- Additional tests have to be created
  that will verify various power scenarios.



Resources

- UVM
  https://verificationacademy.com/
  http://www.accellera.org/community/uvm/
- System Verilog LRM
  http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
- Specman e
  LRM: http://www.ieee1647.org/downloads/prelim_e_lrm.pdf
  Free course: https://www.udacity.com/course/cs348
- www.google.com ☺