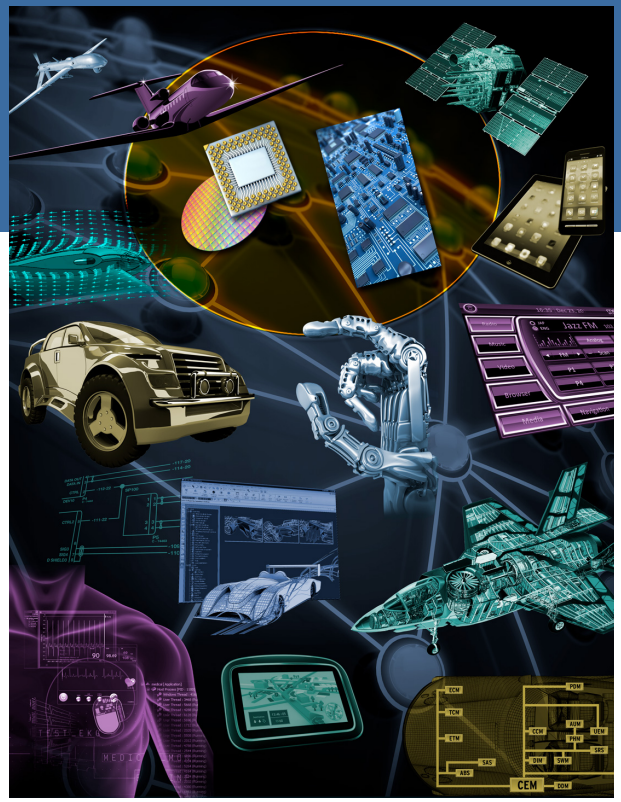


Advanced Verification

Bob Oden

UVM Field Specialist, Mentor

Instructor, ECE Department, NCSU

November 27, 2017

Mentor®
A Siemens Business

Agenda - Monday

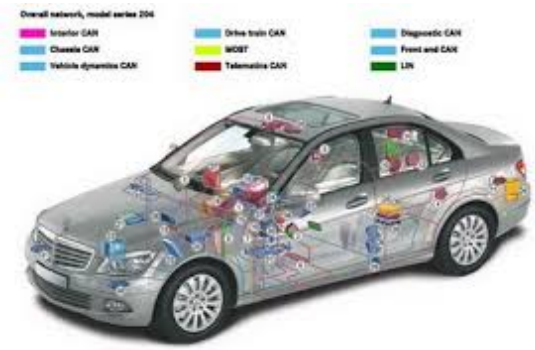
- The circumstances: Electronic design
- The problem: Project risk
- The solution: Systematic verification
- The industry: Trends in verification technology
- SystemVerilog: Standard language for verification
- UVM: Standard methodology for verification

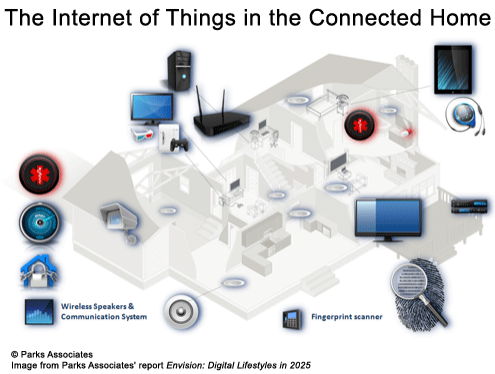
Agenda - Wednesday

- Block level environments
- Golden models
- Chip level environments
- Emulation
- Reuse – Horizontal, vertical, platform
- Simulation and emulation in regression testing
- Verification management for closing coverage

The Circumstances

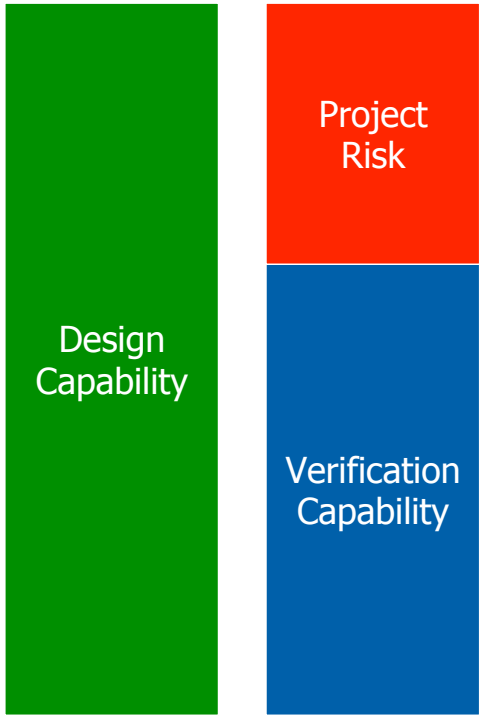
- Ever increasing design size and complexity
  - Cell phone in your pocket
  - Electronics in your car
  - Connected devices in your home





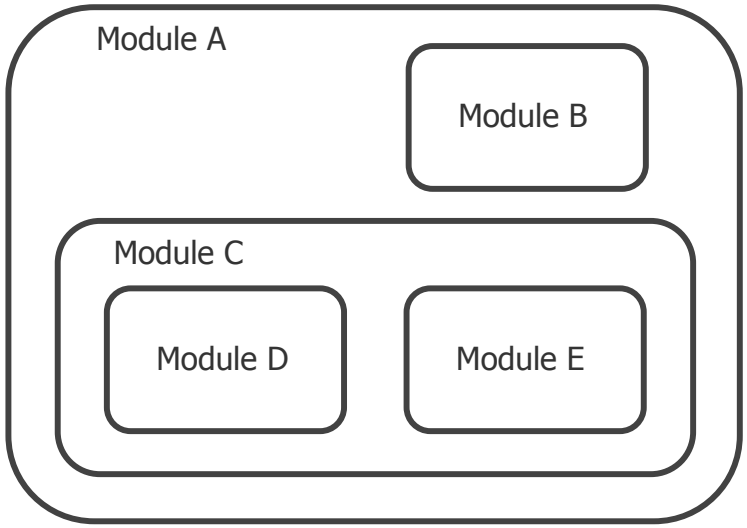
The Problem

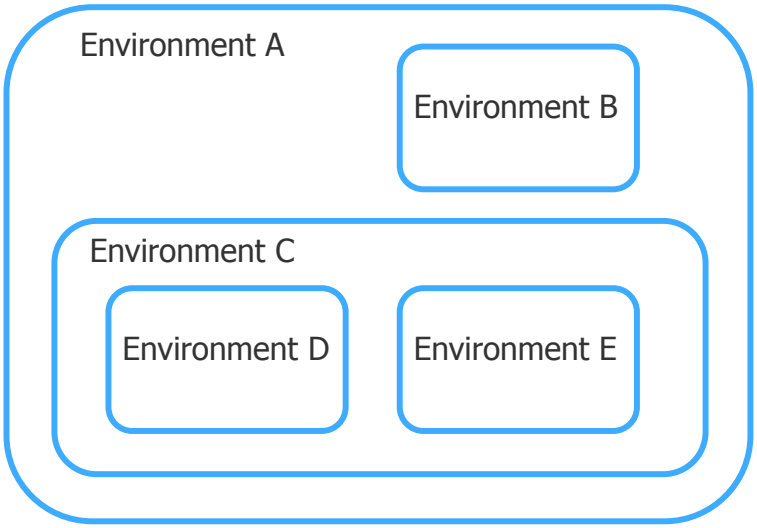
- Project risk is directly proportional to the gap between design capability and verification capability



The Solution

- Systematic verification planning and execution
  — parallels design planning and execution
- Abstract verification language

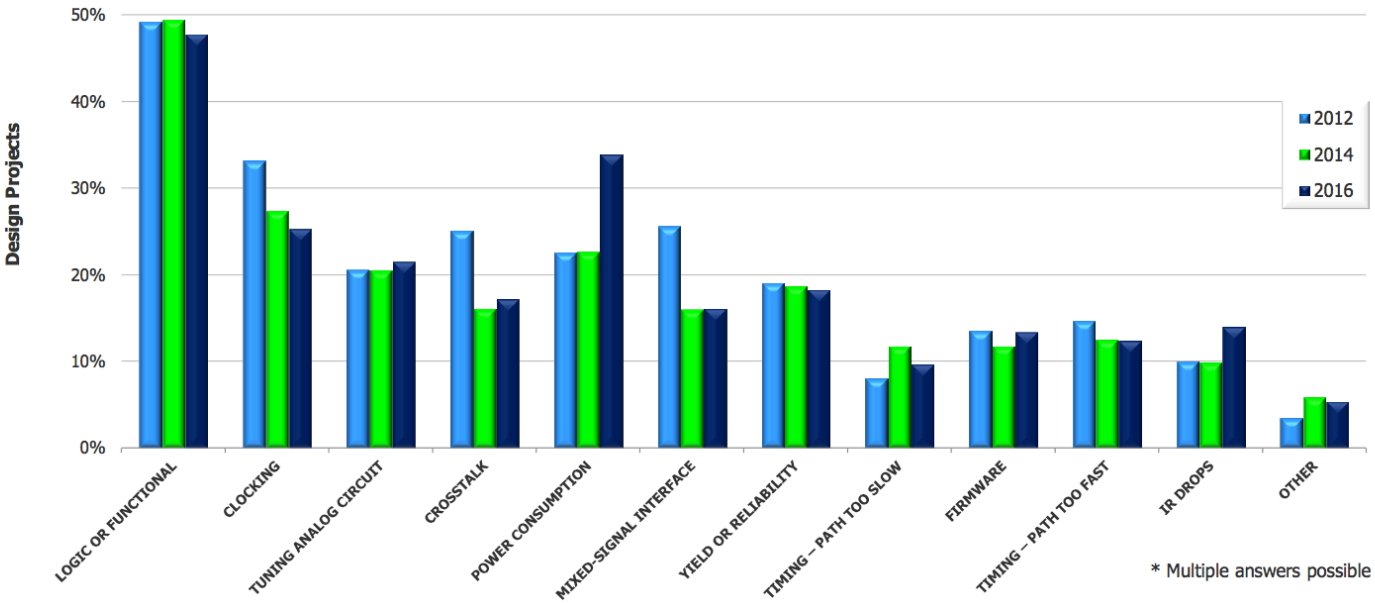




INDUSTRY TRENDS IN VERIFICATION

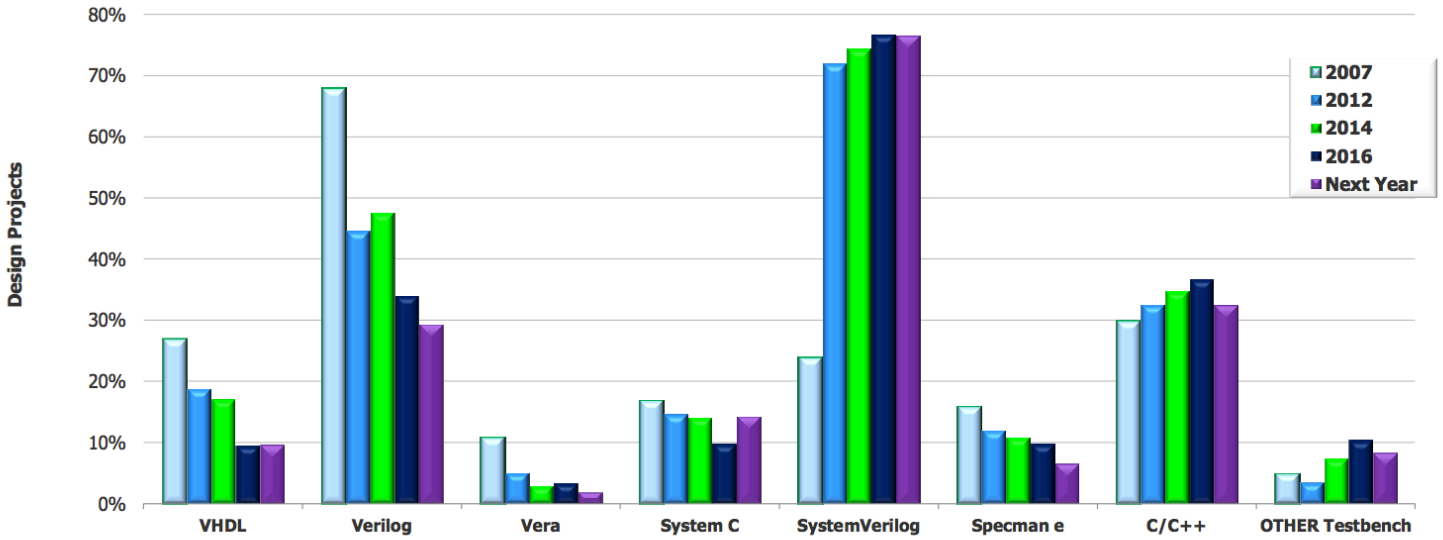
Industry Trends - Flaws

Flaws Contributing to ASIC/IC Respins



Industry Trends – Verification Languages

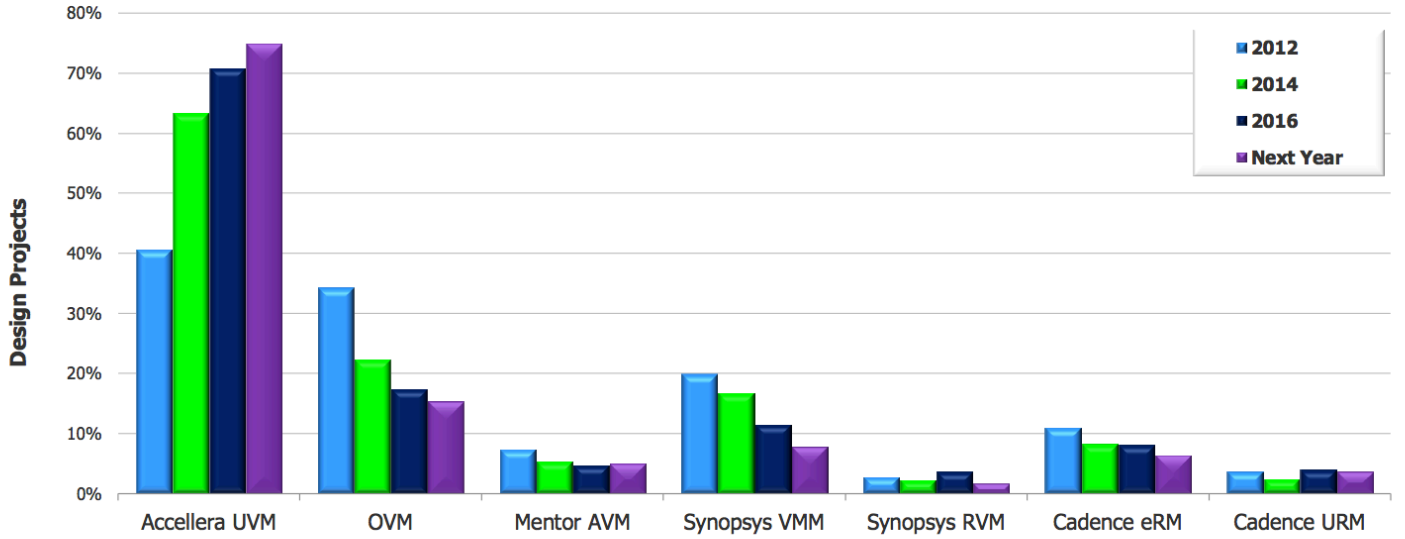
ASIC/IC Verification Language Adoption Trends



* Multiple answers possible

Industry Trends – Verification Methodologies

ASIC/IC Testbench Methodology Adoption Trends



* Multiple answers possible

VERIFICATION PLANNING

Verification Architecture Document

- Identifies testing goals at each simulation level
  - Feature verification at block level
  - Integration and performance at top level
  - Traffic patterns to be tested
- Identifies environment(s) to be developed
  - Block level environments required
  - Upper level environments required
  - Which block level environments are required in upper level simulations
  - Interface pacakges required
- Documents architecture for each environment
  - Required prediction models identified

Verification Architecture Document

- Resource and schedule planning
  - Engineering resources estimated
    - People, tools, expertise,
  - Milestones determined
    - Verification requirements identified
    - Environment architectures defined
    - Order of environment development
    - Verification performed without environments
    - Delivery of prediction models
    - First simulation dates
    - RTL drop dates
    - Coverage closure

The Universal Questions of Verification

- "How do we know when were done?"
- "How do we get there?"
- "Are we there yet?"

Verification Test Plan – What is tested

- "How do we know when were done?"
  - What needs to be tested?
  - Everything can't be tested
    - Manage risk with informed decisions

- Test Plan
  - Identifies what needs to be tested
  - Start by identifying what without consideration of how

Verification Test Plan – How it's tested

- "How do we get there?"
  - How do we complete each item in the Test Plan?
  - Identify best way of achieving each item
- Verification Test Plan
  - Identifies what needs to be tested
  - Identifies how each Test Plan item will be completed
    - Code coverage
    - Functional coverage
    - Assertion coverage
    - Directed test
    - Prediction

Determining Verification Completion?

- "Are we done?"
  - If not, when will we be done?

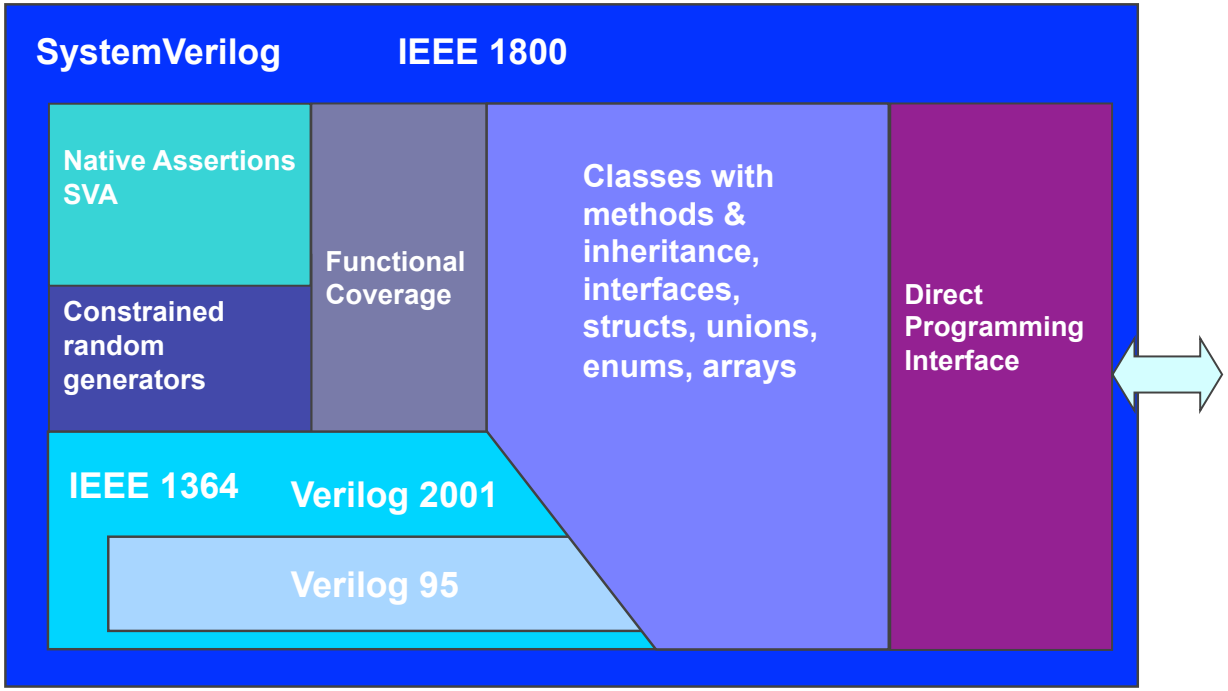
- Verification loop
  1. Add test scenarios
  2. Collect coverage during test regression
  3. Merge test coverage results
  4. Rank test/seed pairs
  5. Generate custom coverage reports
  6. Identify coverage holes
  7. Identify coverage closure trends

Test Plan Example

	A	B	C	D	E	F	G
1	***Verification Plan for AHB to WB Bridge***						
2							
3	**#**	**Section**	**Description**	**Link**	**Type**	**Weight**	**Goal**
4		Simulation					
5	1	AHB_Bus				1	100
6	1.1	AHB Protocol checks	Verify the signaling between:hsel and hready, hsel and haddr, hwrite and hwdata	assert__ahb_hready_follows_hsel assert__ahb_address_stable_throughout_transfer assert__ahb_wdata_stable_throughout_write	Assertion Assertion Assertion	1	100
7	1.2	AHB Transaction Covergroup	All coverpoints within the AHB transaction covergroup	ahb_transaction_cg	CoverGroup	1	100
8	1.3	AHB Address Transitions	Ensure the following address series: to_upper, acc_seq	ahb_transaction_cg::addr_transitions	Coverpoint	1	100
9	1.4	AHB RW Access	Ensure all addresses written to and read from	ahb_transaction_cg::op_x_addr	Cross	1	100
10	2	WB_Bus				1	100
11	2.1	WB Transaction Covergroup	All coverpoints within the WB transaction covergroup	wb_transaction_cg	CoverGroup	1	100
12	2.2	WB Address Transitions	Use all possible delay values	wb_transaction_cg::delay	Coverpoint	1	100
13	2.3	WB RW Access	Ensure all addresses written to and read from	ahb_transaction_cg::op_x_addr	Cross	1	100
14	3	Compulsory_Tests				1	100
15	3.1	Base test	Run test_top	test_top.*	test	1	100
16	3.2	Random test	Run ahb_random_test	ahb_random_test.*	test	1	100
17	3.3	Bridging test	Test bridge mapping range	ahb_wb_bridge_range_test	test	1	100
18	4	Code_Coverage				1	100
19	4.1	RTL Core	"Ensure that all design units have 100% statement coverage 100% branch coverage"	/hdl_top/ahb2wb	Instance	1	100

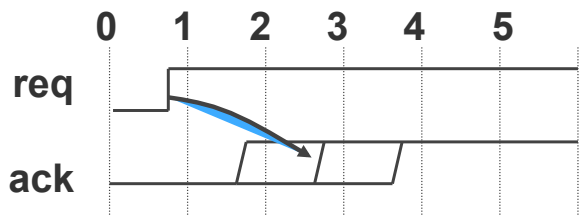
SYSTEMVERILOG

SystemVerilog - Overview



SystemVerilog Assertions

A concise description of [un]desired behavior

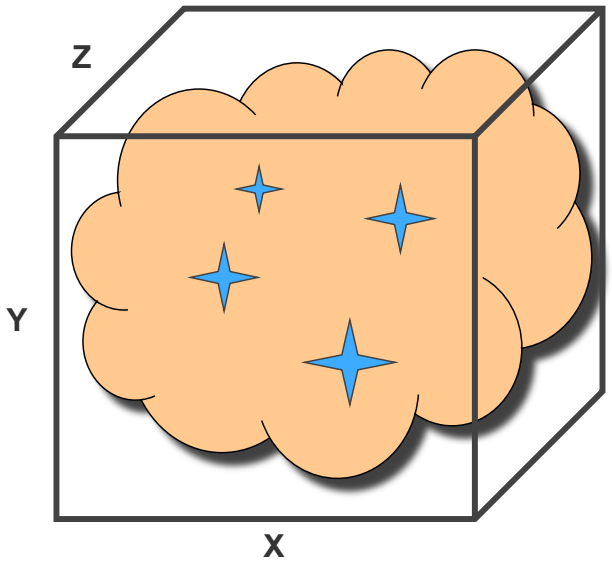


Example intended behavior

"After the request signal is asserted, the acknowledge signal must come 1 to 3 cycles later"

SystemVerilog Constrained Random

- Declare a set of random variables – X, Y, Z
- Declare a set of constraints – $Y < 42$, $X \leq Y \leq Z$
- Find the set of values that meet the given constraints
- Randomly pick solutions



Class Based Randomization

- Randomization within SV is object based
  - All SV data types randomizable
  - Variables preceeded with rand or randc keywords randomized
  - Objects instantiated within the object are not automatically randomized
  - Constraints applied when variables randomized

```
class Bus;
   rand bit [15:0] addr;      // Randomized
   randc int data;            // Randomized
   employee busDriver;    // Not randomized

   constraint word_align { addr[1:0]==2' b0;}

endclass : Bus

…
Bus bus = new;
assert(bus.randomize());
assert(bus.data.randomize() ); // Only randomize data
…

// Overlay constraint on constraints inside class
assert(bus.randomize() with { addr[2] == 1' b0;});
```

Randomization Distribution

- rand: Uniform distribution of random values with possible repeating
- randc: Uniform “cyclic” values cover all possible values before repeating
- Applies to
  - SV data type variables
  - All elements of arrays
  - All elements and size of dynamic and associative arrays

Constraint Examples

packet.sv (C:\projects\Demonstration\easy_tlm\tb) - GVIM

File Edit Tools Syntax Buffers Window Help

```
13 class packet;
14
15    // The following properties are visible to the DUT
16    rand bit [7:0]  destinationAddress;
17    rand bit [7:0]  sourceAddress;
18         bit [7:0]  packetData[$];
19
20    // The following properties are used to influence
21    // packet generation, error conditions, etc.
22    rand bit [31:0] packetId;
23    rand bit [ 7:0] packetSize;
24    rand bit        mopError;
25    rand bit [ 7:0] mopErrorLocation;
26
27    // Constraints on packet generation
28    constraint packetSize_c { packetSize != 0 ;
29                              packetSize dist {[  1:  3]:/6,
30                                               [  4: 47]:/3,
31                                               [ 48:191]:/2,
32                                               [192:255]:/1 }; }
33    constraint destinationAddress_c { destinationAddress inside {0,1,2,3};}
34    constraint mopErrorLocation_c   { mopErrorLocation < packetSize-1; }
35    constraint external_c;
                                                          32,35          16%
```

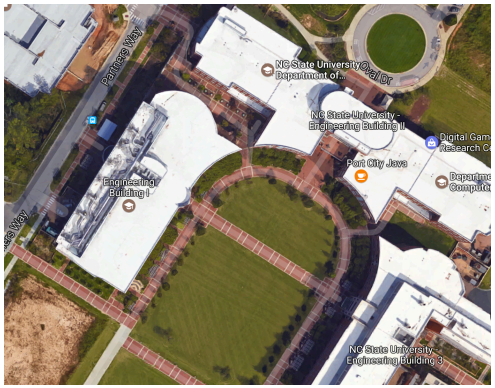
Declarative vs. Procedural

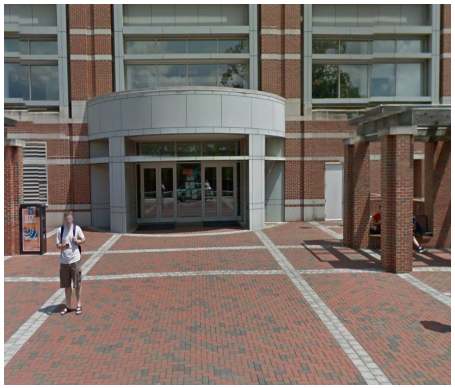
- Contained in any class
- pre_randomize(): Automatically called before object randomization
  - Override to perform initialization or set preconditions, etc
- post_randomize(): Automatically called after object randomization
  - Override to cleanup, report, set post conditions, etc.

```
class packet;

…
function void pre_randomize();
   length = data.size();
endfunction : pre_randomize

function void post_randomize();
   dataSum = data.sum;
endfunction : post_randomize

endclass : packet
```

Coverage – Improving Your Understanding

- Increasing angles of observation increases understanding
  - Each coverage type is a different view of verification effort
  - Each coverage type is complimentary and provides sanity against each other





Coverage Options

- Code Coverage
  - Code was executed
- Functional Coverage
  - Data and scenario
- Assertion Coverage
  - Signaling

- Note on coverage measurement during simulation run
  - Test should not run until coverage achieved
  - Use test ranking to optimize

Code Coverage

- **Statement coverage** — counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch coverage** — counts the execution of each conditional "if/then/else" and "case" statement and indicates when a true or false condition has not executed.
- **Condition coverage** — analyzes the decision made in "if" and ternary statements and can be considered as an extension to branch coverage.
- **Expression coverage** — analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle coverage** — counts each time a logic node transitions from one state to another.
- **FSM coverage** — counts the states, transitions, and paths within a finite state machine.

Functional Coverage

- **Covergroup -** encapsulates the specification of a coverage model and may include: clocking event, coverpoints, bins, cross coverage, transition coverage, coverage options.
- **Coverpoint** – specifies an integral expression to be covered. Values of variables within the class. Can be divided and described using bins.
- **Bins** – separate collections of variable values from among all possible variable values
- **Cross** – coincident values of one or more coverpoints
- **Transition** – Tracks sequential values or value groups

Assertion Coverage

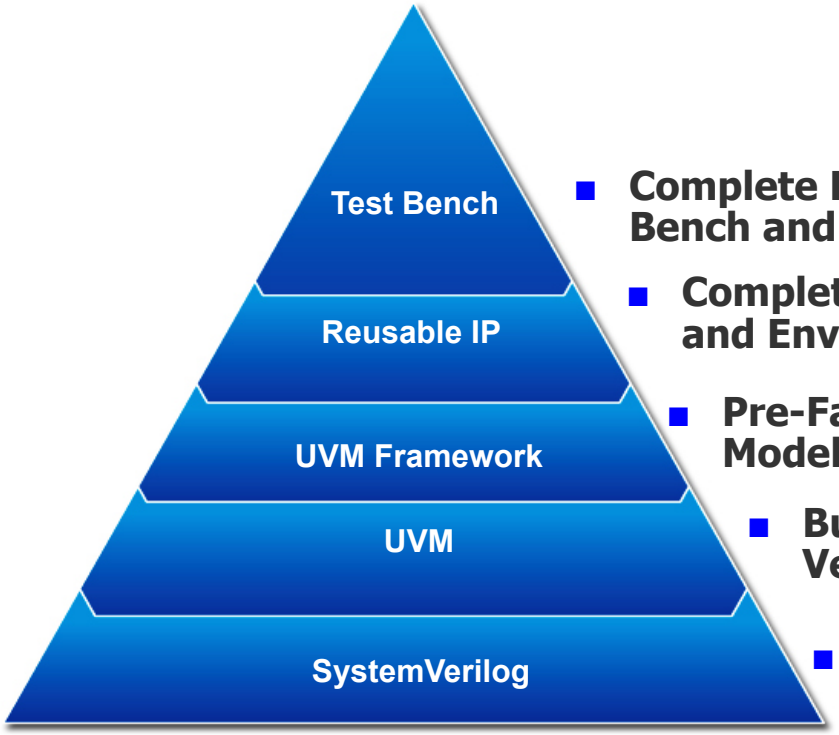
- **Cover directive**
  - Identifies observed signal relationships
  - Applied to properties or sequences
  - Reports attempt and success count
  - Useful for identifying vacuous passes
  - Signal sequences can trigger other operations
    - Coverage sampling
    - Event notification

UNIVERSAL VERIFICATION METHODOLOGY

UVM

Mentor
A Siemens Business

UVM Reuse Structure



- Complete Home: DUT Specific Bench and Stimulus
- Completed Rooms: Interface and Environment Packages
- Pre-Fabricated Rooms: UVM Use Model and Reuse Methodology
- Building Supplies: Common Verification Building Blocks
- Raw Materials: Abstract Programming Language

Common TB Features

- Core capabilities of every verification environment
  - Component hierarchy
  - Moving data between components
  - Managing test flow
  - Generating messages
  - Synchronizing activities
  - Sharing resources
  - Generating stimulus
  - Checking results
  - Creating test cases

All simulation benches do the same things…differently.

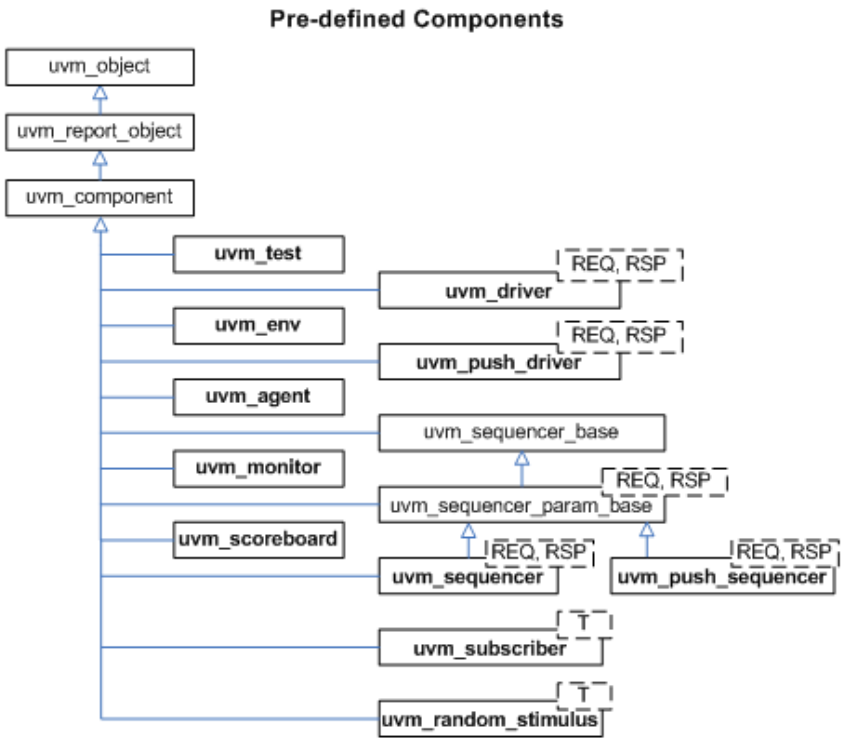
UVM Features

- Core capabilities of every verification environment
  - Component hierarchy
  - Moving data between components
  - Managing test flow
  - Generating messages
  - Synchronizing activities
  - Sharing resources
  - Generating stimulus
  - Checking results
  - Creating test cases

UVM provides a standardized implementation…Freedom from choice!

UVM Components

- UVM_component
  - Encapsulates common data and functionality of all components
    - Hierarchy
    - Phasing
    - Objections
    - Factory
    - Recording
- Basis for environment hierarchy



UVM Stimulus

- Transaction object sent to environment through the sequencer
- Represents bus operation
- Untimed
- Contains sequence ID and sequence item ID

uvm_sequence_item

The base class for user-defined sequence items and also the base class for the uvm_sequence class.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_transaction
- **uvm_sequence_item**

Class Declaration

```
class uvm_sequence_item extends uvm_transaction
```

new	The constructor method for uvm_sequence_item.
set_id_info	Copies the sequence_id and transaction_id from the referenced item into the calling item.
set_sequencer	Sets the default sequencer for the sequence to sequencer.
get_sequencer	Returns a reference to the default sequencer used by this sequence.
set_parent_sequence	Sets the parent sequence of this sequence_item.
get_parent_sequence	Returns a reference to the parent sequence of any sequence on which this method was called.

UVM Reporting

uvm_report_object

The uvm_report_object provides an interface to the UVM reporting facility.

Class Hierarchy

- uvm_void
- uvm_object
- **uvm_report_object**

Class Declaration

```
class uvm_report_object extends uvm_object
```

new	Creates a new report object with the given name.

Reporting

uvm_report uvm_report_info uvm_report_warning uvm_report_error uvm_report_fatal	These are the primary reporting methods in the UVM.
set_report_verbosity_level	This method sets the maximum verbosity level for reports for this component.
set_report_id_verbosity set_report_severity_id_verbosity	These methods associate the specified verbosity with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_severity_action set_report_id_action set_report_severity_id_action	These methods associate the specified action or actions with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_severity_override set_report_severity_id_override	These methods provide the ability to upgrade or downgrade a message in terms of severity given *severity* and *id*.
set_report_default_file set_report_severity_file set_report_id_file set_report_severity_id_file	These methods configure the report handler to direct some or all of its output to the given file descriptor.
get_report_verbosity_level	Gets the verbosity level in effect for this object.
get_report_action	Gets the action associated with reports having the given *severity* and *id*.
get_report_file_handle	Gets the file descriptor associated with reports having the given *severity* and *id*.

UVM Common Phases

- Only components participate in phases
- Phase order
  - As shown in table
- Function phases
  - Build, connect, end_of_elaboration, start_of_simulation, extract, check, report
- Task phases (time consuming)
  - run

UVM Common Phases	The common phases are the set of function and task phases that all uvm_components execute together.
uvm_build_phase	Create and configure of testbench structure
uvm_connect_phase	Establish cross-component connections.
uvm_end_of_elaboration_phase	Fine-tune the testbench.
uvm_start_of_simulation_phase	Get ready for DUT to be simulated.
uvm_run_phase	Stimulate the DUT.
uvm_extract_phase	Extract data from different points of the verficiation environment.
uvm_check_phase	Check for any unexpected conditions in the verification environment.
uvm_report_phase	Report results of the test.
uvm_final_phase	Tie up loose ends.

- Components participate in phase if task/function of phase name exists
  - Automatically executed

Additional Time Consuming Phases in UVM

- Run-Time phases executed in parallel with run phase.

- Do Not Use
  - Run –Time phases
  - User defined phases
  - Phase jumping

UVM Run-Time Phases	The run-time schedule is the pre-defined phase schedule which runs concurrently to the uvm_run_phase global run phase.
uvm_pre_reset_phase	Before reset is asserted.
uvm_reset_phase	Reset is asserted.
uvm_post_reset_phase	After reset is de-asserted.
uvm_pre_configure_phase	Before the DUT is configured by the SW.
uvm_configure_phase	The SW configures the DUT.
uvm_post_configure_phase	After the SW has configured the DUT.
uvm_pre_main_phase	Before the primary test stimulus starts.
uvm_main_phase	Primary test stimulus.
uvm_post_main_phase	After enough of the primary test stimulus.
uvm_pre_shutdown_phase	Before things settle down.
uvm_shutdown_phase	Letting things settle down.
uvm_post_shutdown_phase	After things have settled down.

UVM Factory – Constructor Proxy

- UVM Class Reference
  - "The uvm_factory is used to manufacture (create) UVM objects and components. Only one instance of the factory is present in a given simulation"
- Used to determine the class type of an object being constructed
- Used to change the class type of an object being constructed

UVM Factory - Registration

- Register Classes with the Factory
  - Objects
    - `uvm_object_utils
    - `uvm_object_param_utils
  - Components
    - `uvm_component_utils
    - `uvm_component_param_utils
- Change Registry Table
  - set_type_override()
  - set_instance_override()
- Construct Classes with the Factory
  - Create() instead of new()

Factory Registry

Requested Type	Returned Type
ahb2wb_predictor	ahb2wb_predictor
wb2ahb_predictor	wb2ahb_predictor

UVM Factory – Overrides

Factory Registry Before

Requested Type	Returned Type
ahb2wb_predictor	ahb2wb_predictor
wb2ahb_predictor	wb2ahb_predictor

ab2wb_predictor::set_type_override(ahb2wb_dpi_predictor::get_type())

Factory Registry After

Requested Type	Returned Type
ahb2wb_predictor	ahb2wb_dpi_predictor
wb2ahb_predictor	wb2ahb_predictor

Set overrides BEFORE
object construction
Inside test class or using UVM CLI

Resource Sharing Within UVM

- Resources typically shared within a simulation
  - Configuration objects
  - Virtual interface handles
  - Sequencer handles

- Resource visibility limited by type, scope, name

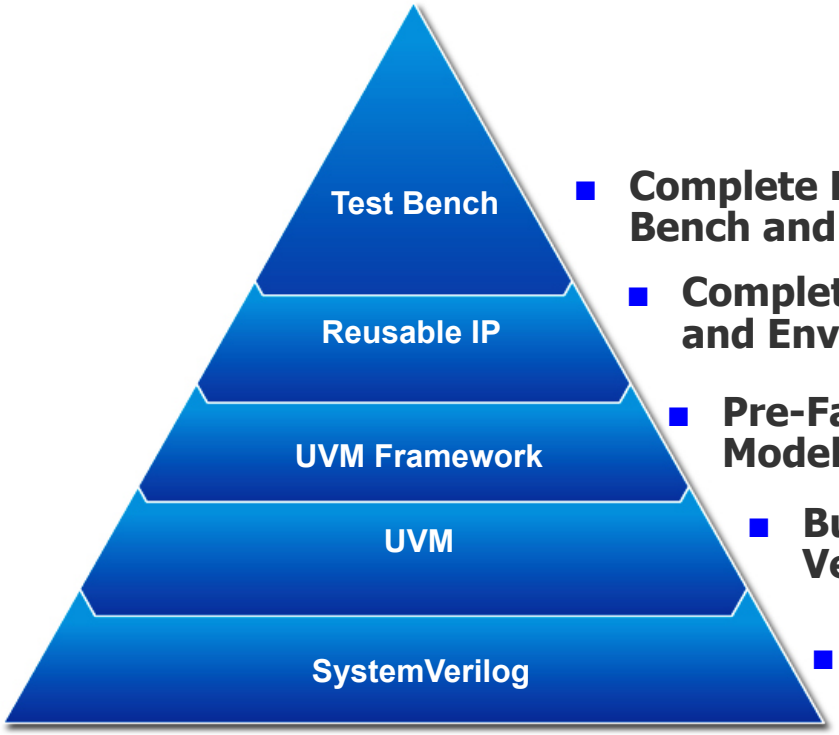
set

```
static function void set(uvm_component cntxt,
                         string        inst_name,
                         string        field_name,
                         T             value      )
```

get

```
static function bit get(      uvm_component cntxt,
                              string        inst_name,
                              string        field_name,
                        inout T             value      )
```

UVM Reuse Structure

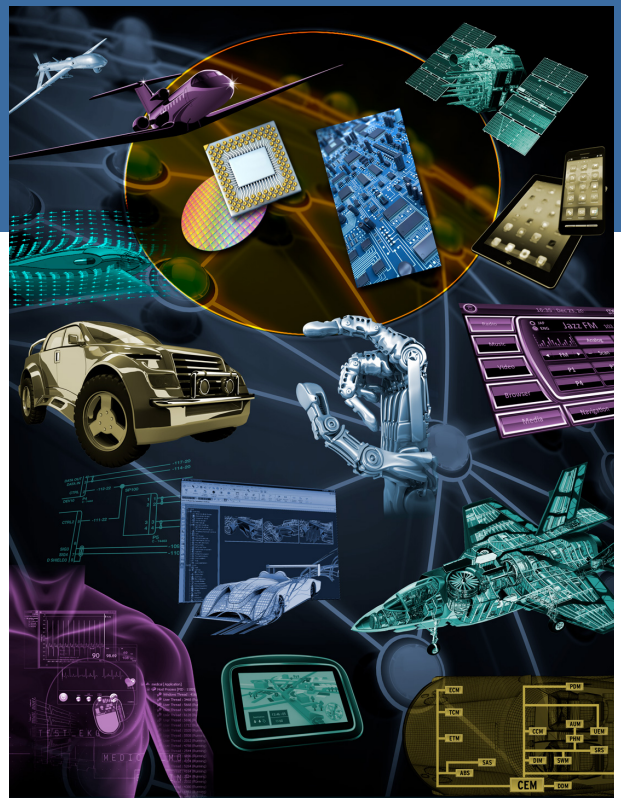


- Complete Home: DUT Specific Bench and Stimulus
- Completed Rooms: Interface and Environment Packages
- Pre-Fabricated Rooms: UVM Use Model and Reuse Methodology
- Building Supplies: Common Verification Building Blocks
- Raw Materials: Abstract Programming Language

Mentor®

A Siemens Business

www.mentor.com



Advanced Verification

Bob Oden

UVM Field Specialist, Mentor

Instructor, ECE Department, NCSU

November 29, 2017

Mentor®
A Siemens Business

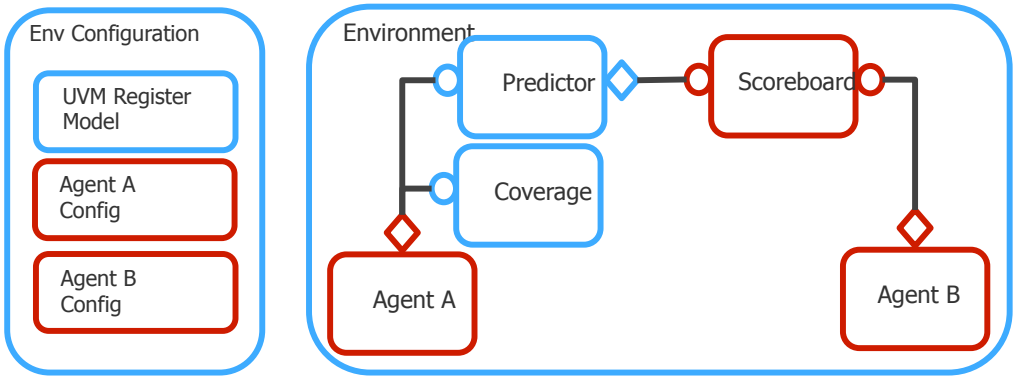
Agenda - Wednesday

- Block level environments
- Golden models
- Chip level environments
- Emulation
- Reuse
- Simulation and emulation in regression testing
- Verification management for closing coverage

BLOCK LEVEL ENVIRONMENTS

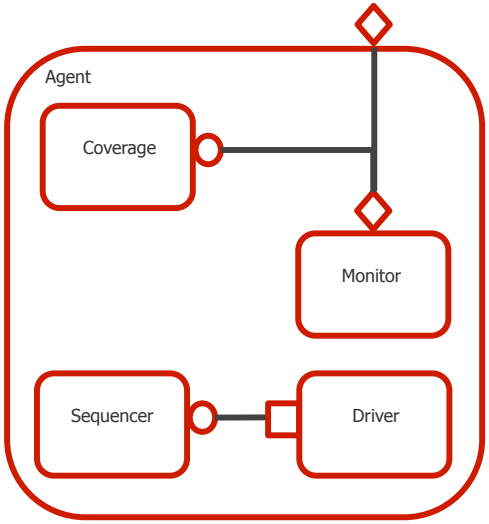
Block Level Environment

- Contents
  - Configuration
  - Agents
  - Predictors
    - Modeling options
  - Scoreboards
    - Verifying DUT output
    - End of Test checking
  - Coverage
    - Objective measurement of test results
  - Stimulus vs. Analysis
    - Independent for reuse



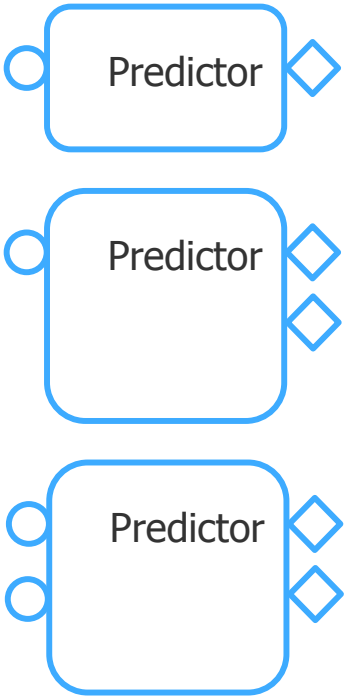
Agent

- Provides connection point between BFM signal level activity and environment transaction level activity
- Active agent
  - Provides data to driver BFM for signal activity
  - Receives data from monitor BFM of signal activity
- Passive agent
  - Receives data from monitor BFM of signal activity
- Broadcasts transactions to other components within environment
- No DUT specific operations



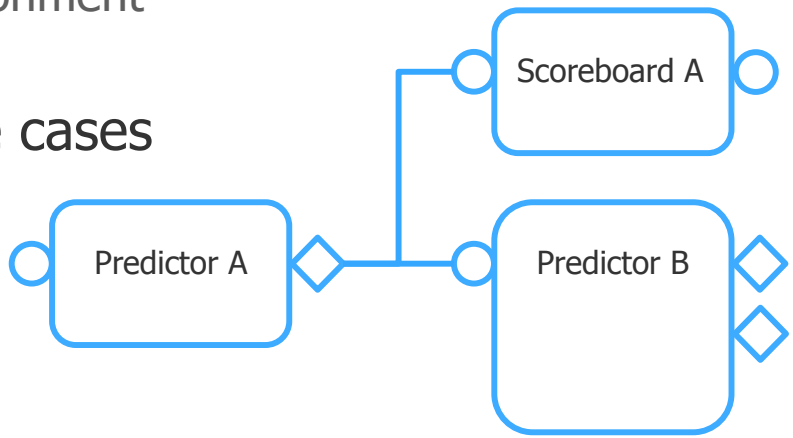
Predictor

- Models all or some of DUT operation
  - Single, all inclusive, golden model of DUT
  - Distributed golden model of DUT
  - Based on data flow through the design
  - Creates expected DUT output transactions
- Untimed
- Receives transactions from agents through analysis_export(s)
  - Creates expected DUT output transaction from
    - Input transaction
    - Prior transaction(s)
    - Current configuration or state



Predictor

- Sends transactions to scoreboards or other predictors through analysis_port(s)
- Any combination of analysis_exports and analysis_ports possible
  - Dependent on DUT data flow and environment architecture
- Can provide output checking in some cases
  - Memory checking
  - State flow
  - Etc.

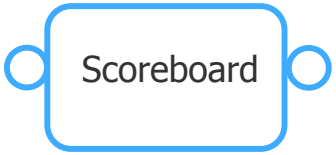


Prediction – Modeling Behavior

- Language options for modeling DUT behavior
  - SystemVerilog – Native
  - C/C++ - Use Direct Programming Interface, DPI, to call C/C++ functions from SystemVerilog and SystemVerilog from C/C++ functions
  - SystemC – Use UVM Connect to pass data from SystemVerilog and SystemC and from SystemC to SystemVerilog

Scoreboard

- Verifies DUT output
  - Compared against predicted value
  - Uses compare function in transaction class
- Transaction handles discarded after comparison
  - Allows for memory to be reclaimed through garbage collection
- Performs end of test operations
  - Ensure expected transaction storage is empty
  - Delay test completion until storage is empty
  - Ensure scoreboard received expected transactions
  - Output summary of scoreboard activity



Scoreboard – Storing Predicted Transactions

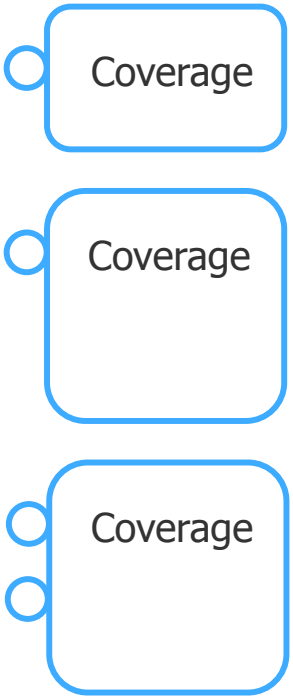
- Two basic mechanisms to handle two basic data flows
  - In order
    - DUT output transaction order predictable and guaranteed to be in order in relation to DUT input transactions
    - Use uvm_tlm_analysis_fifo to store expected transactions
  - Out of order
    - DUT output transaction order not predictable or ordered
    - Use SystemVerilog associative array to store expected transactions

Scoreboard – EOT Checking and Reporting

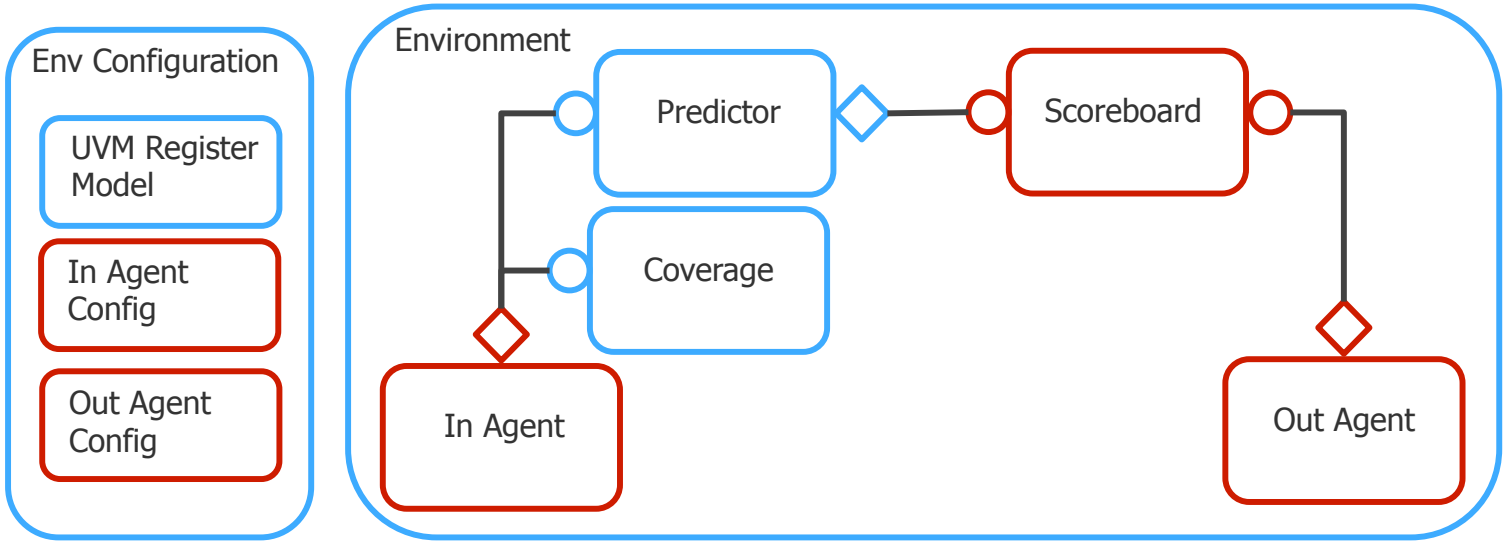
- Checks
  - Empty check
  - Activity check
  - Wait for drain
- Reports
  - Transaction counts
  - Remaining transaction display
  - summary

Coverage

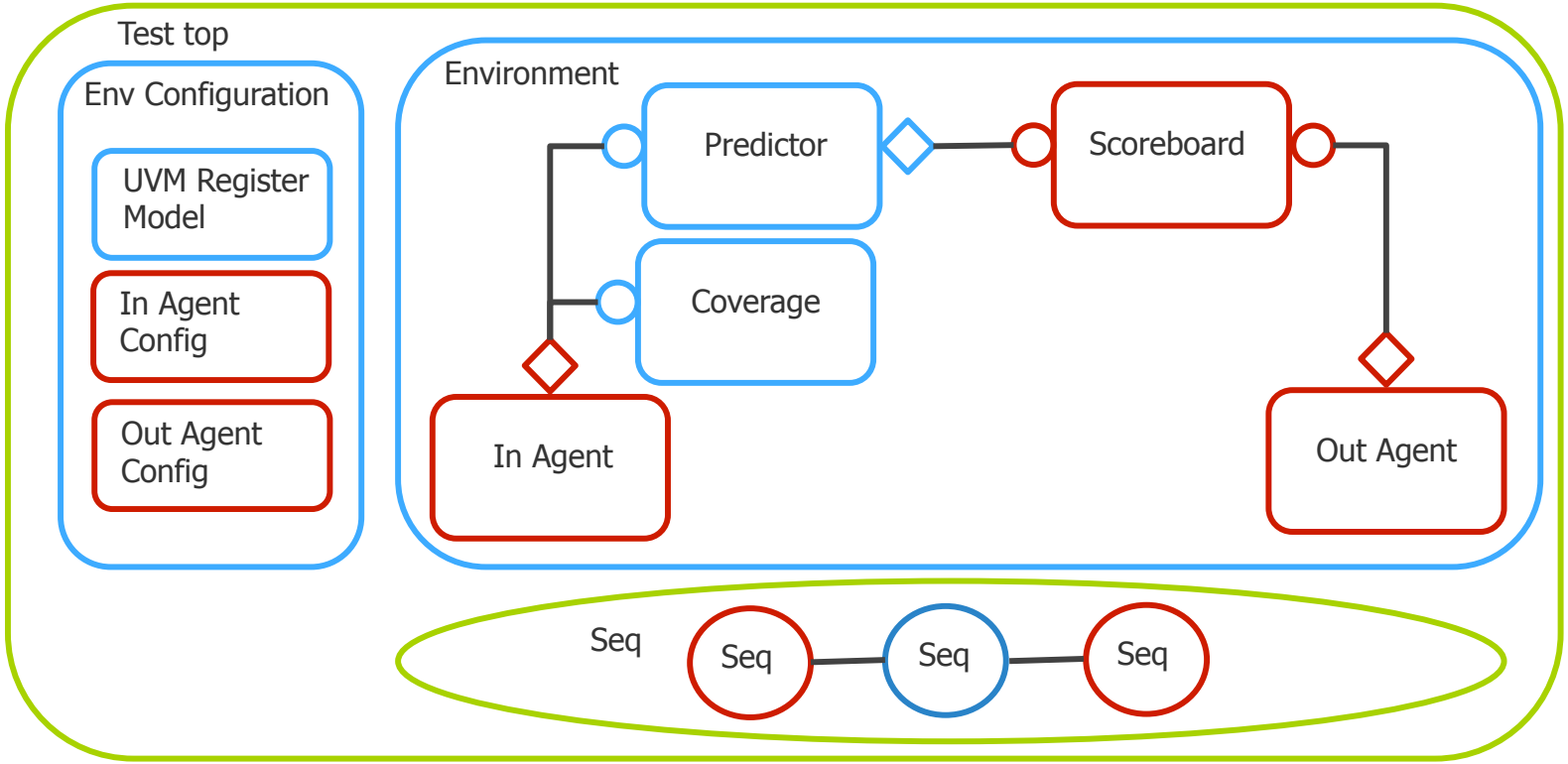
- Gathers and records data for functional coverage
  - Transactions
  - Configuration settings
  - Current state
  - Transitions
- Only receives transactions
  - Does not broadcast
- Contains
  - Covergroups
  - Coverpoints
  - Cross Coverage
  - Etc.



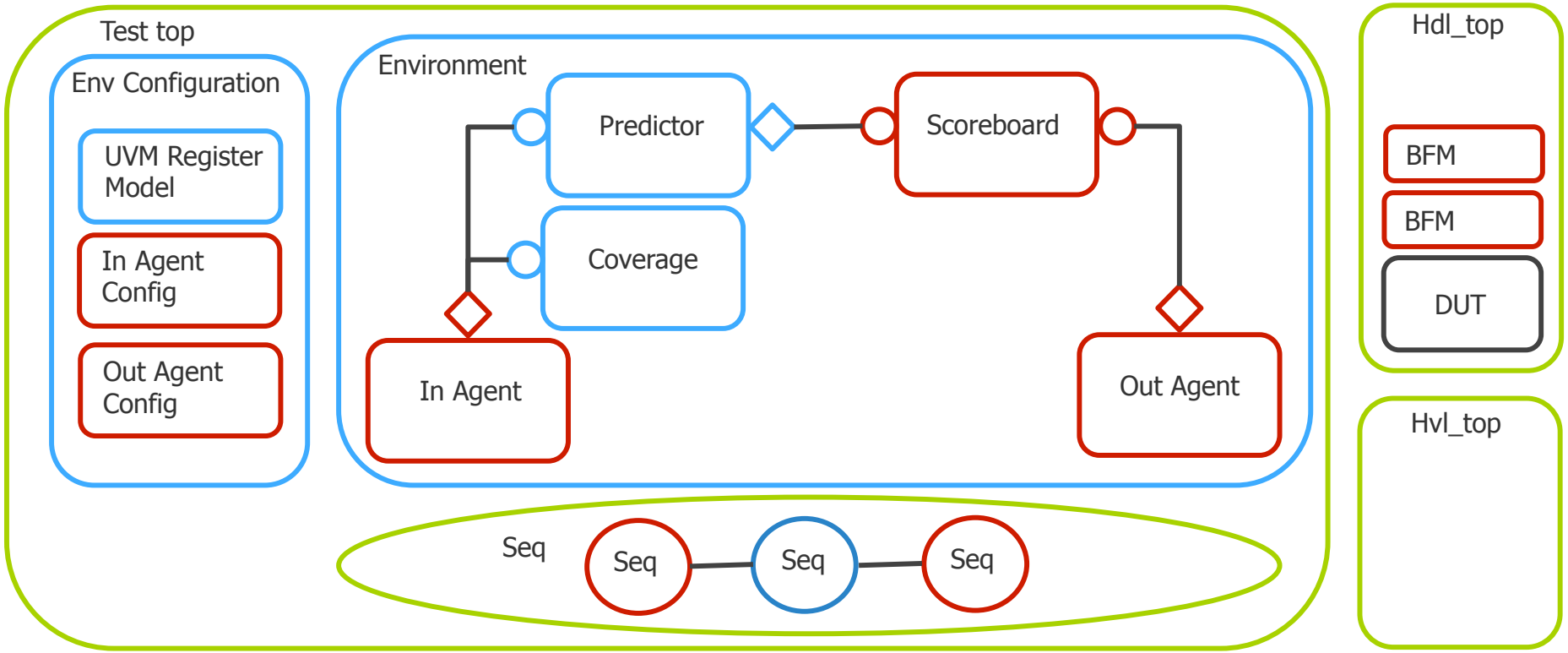
Block Level Environment Components



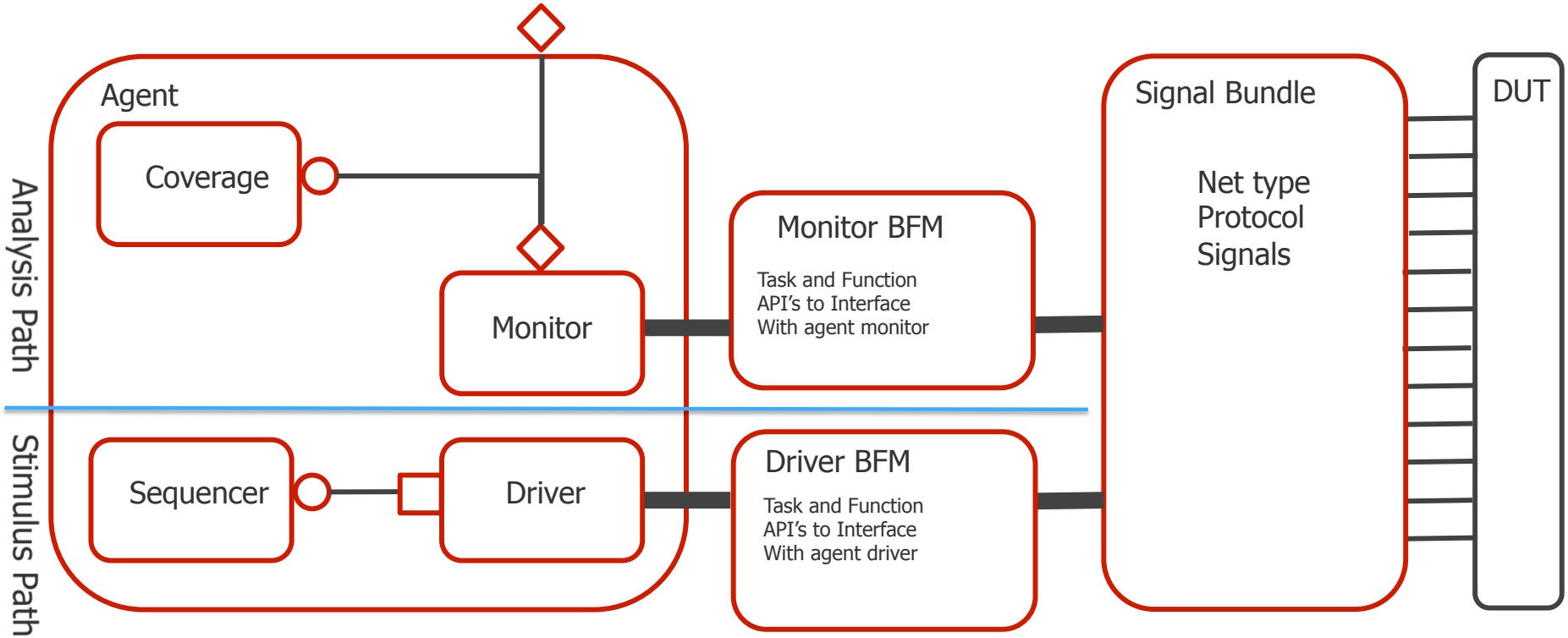
Block Level Test Component



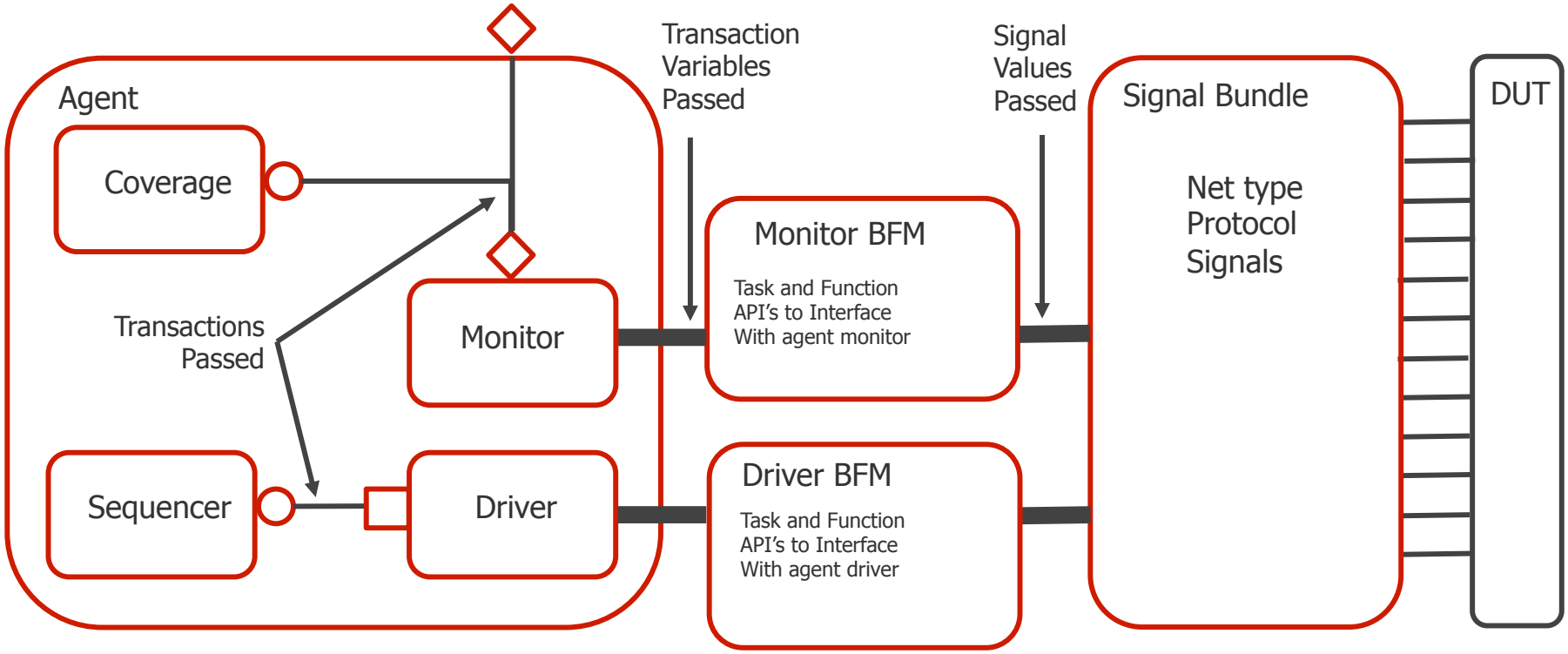
Block Level Test Bench Modules



Agent and Bus Functional Models

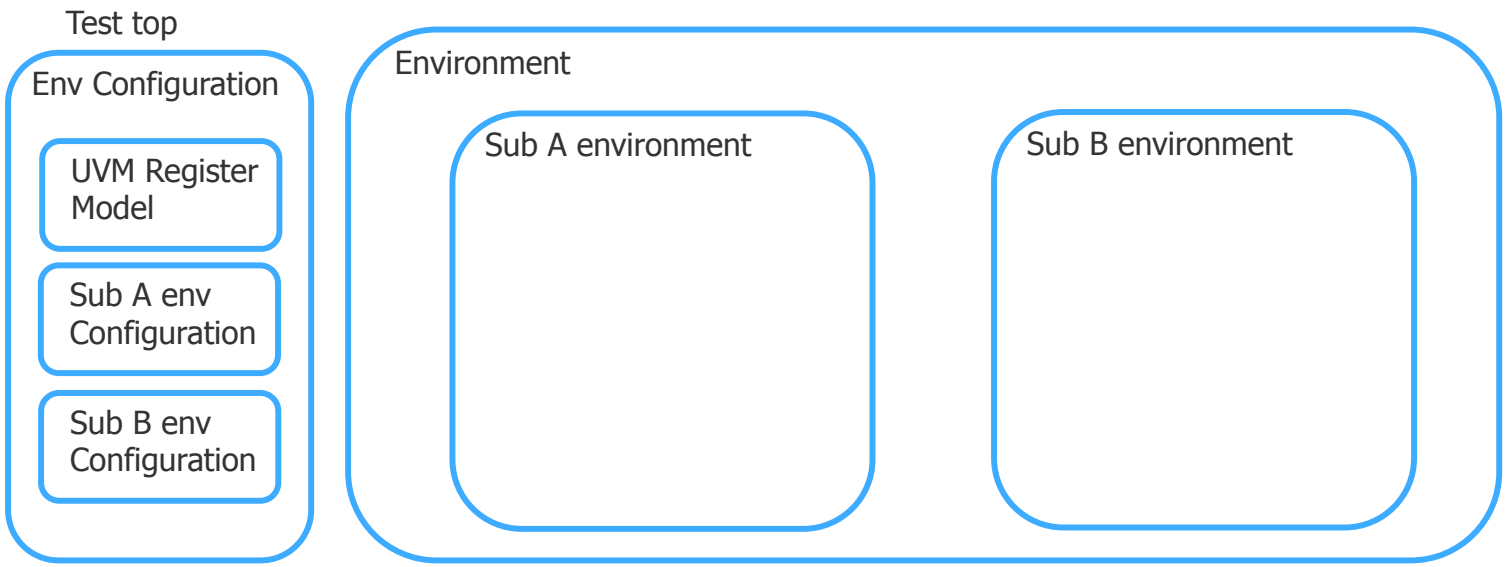


Agent and BFM – Data Flow

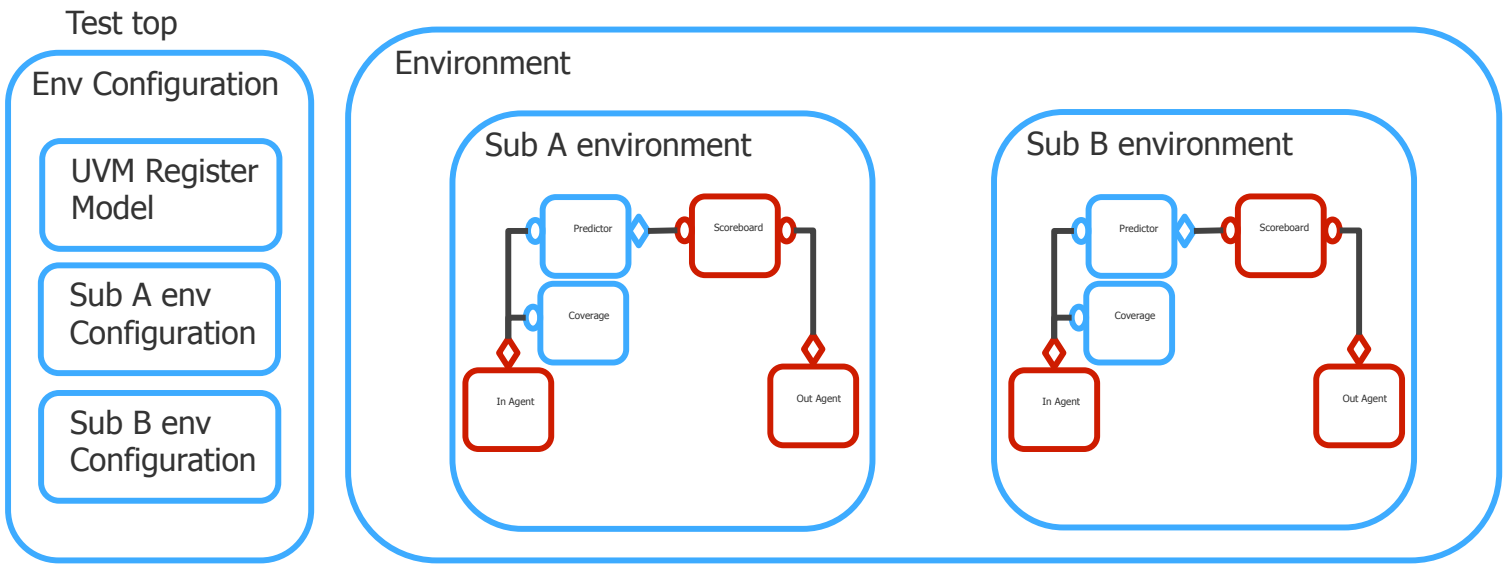


CHIP LEVEL ENVIRONMENTS

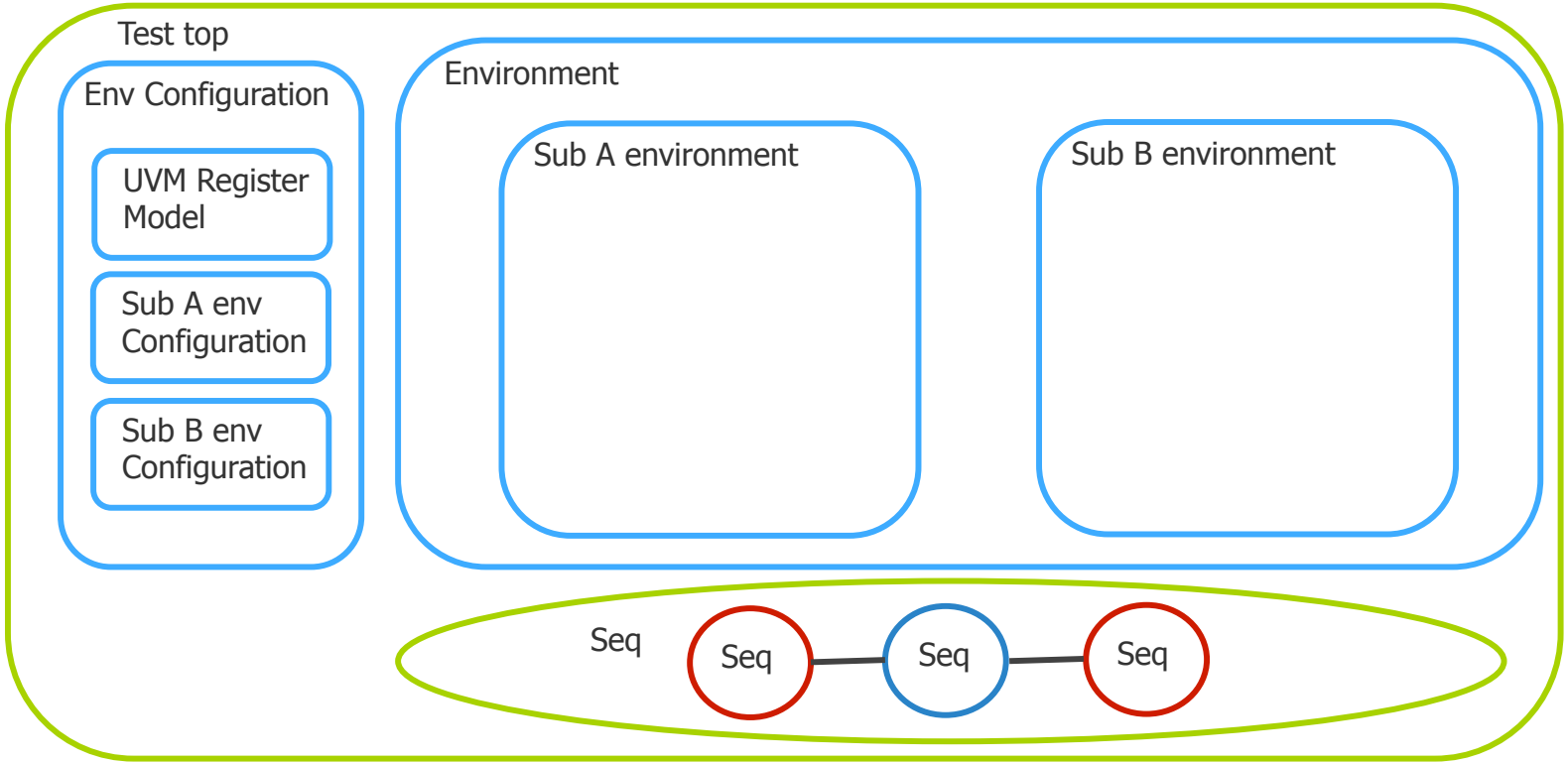
Chip Level Environment Components



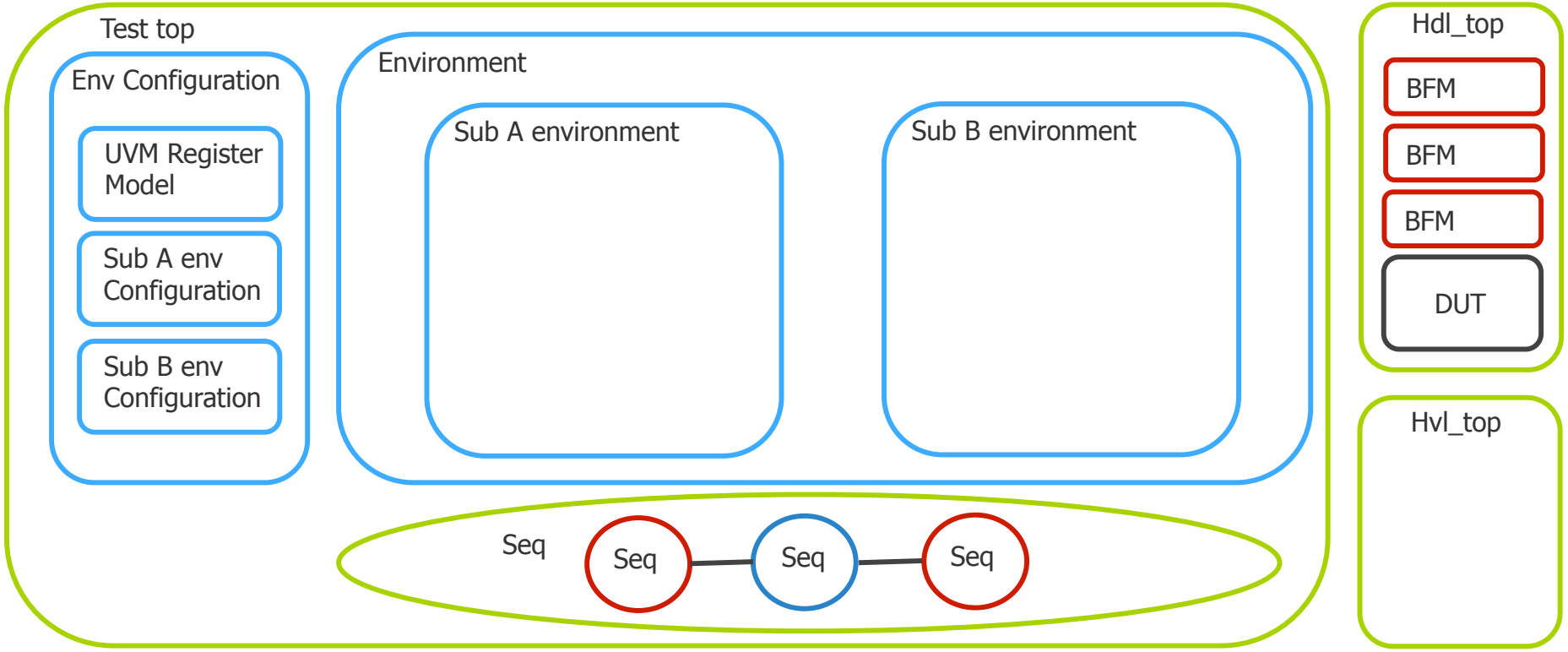
Chip Level Environment Components



Chip Level Test Components



Chip Level Test Bench Modules

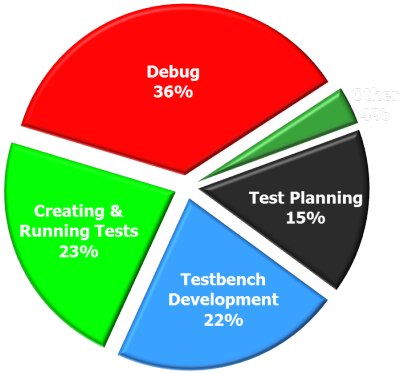


UVM & CO-EMULATION

Testbench-Driven Verification Productivity

SystemVerilog, UVM, Block-to-Top Reuse, Platform Portability

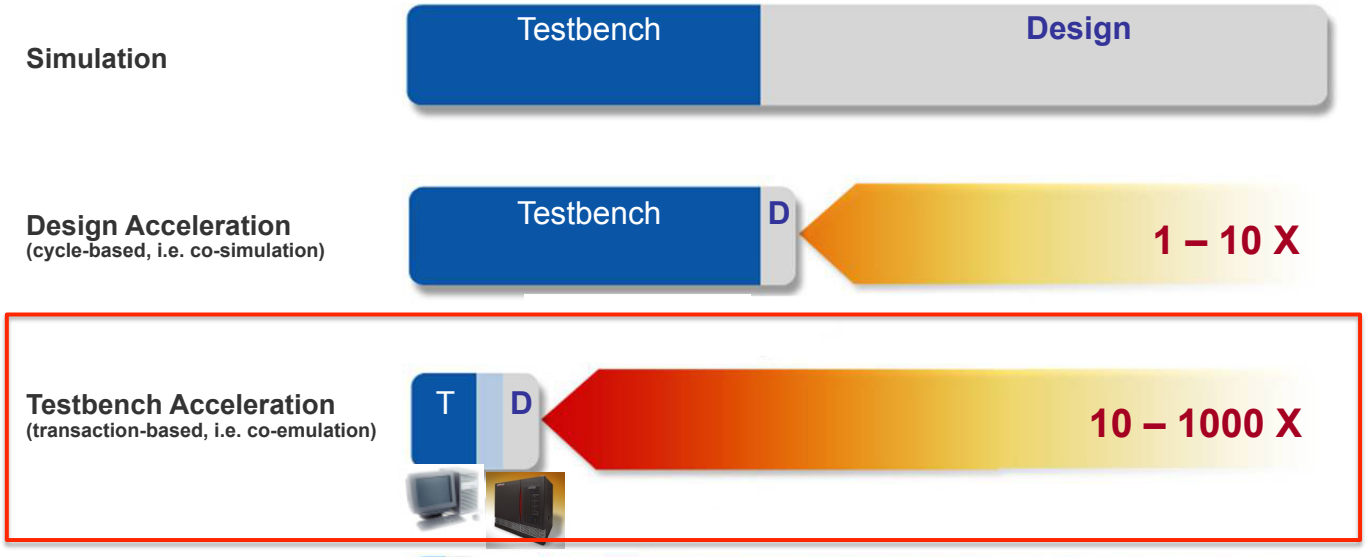
Mean time verification engineers spend in different tasks



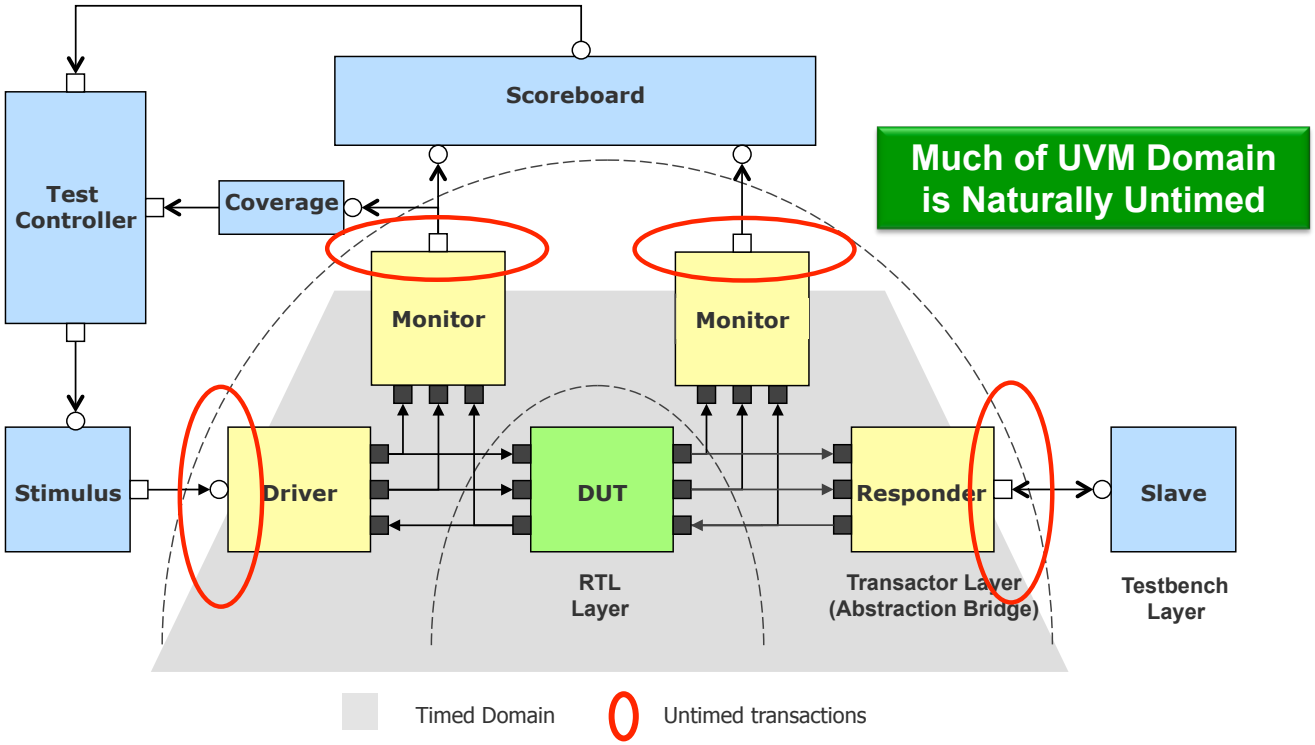
Wilson Research Group and Mentor Graphics, 2012 Functional Verification Study, Used with permission

- Electronics systems companies need large improvements in (simulation-based) verification productivity
- Adoption of SystemVerilog & UVM for increased productivity
  - Faster to create reusable verification components, testbenches and tests
  - Horizontal and vertical reuse:
    - Components, modules, libraries across projects
    - Block to sub-system to system level within a single project
- Veloce enables a 3rd dimension of reuse for accelerating SV/UVM
  - Platform portability:
    - Testbenches, ABV, CDV, VIP, etc. across engines/tools

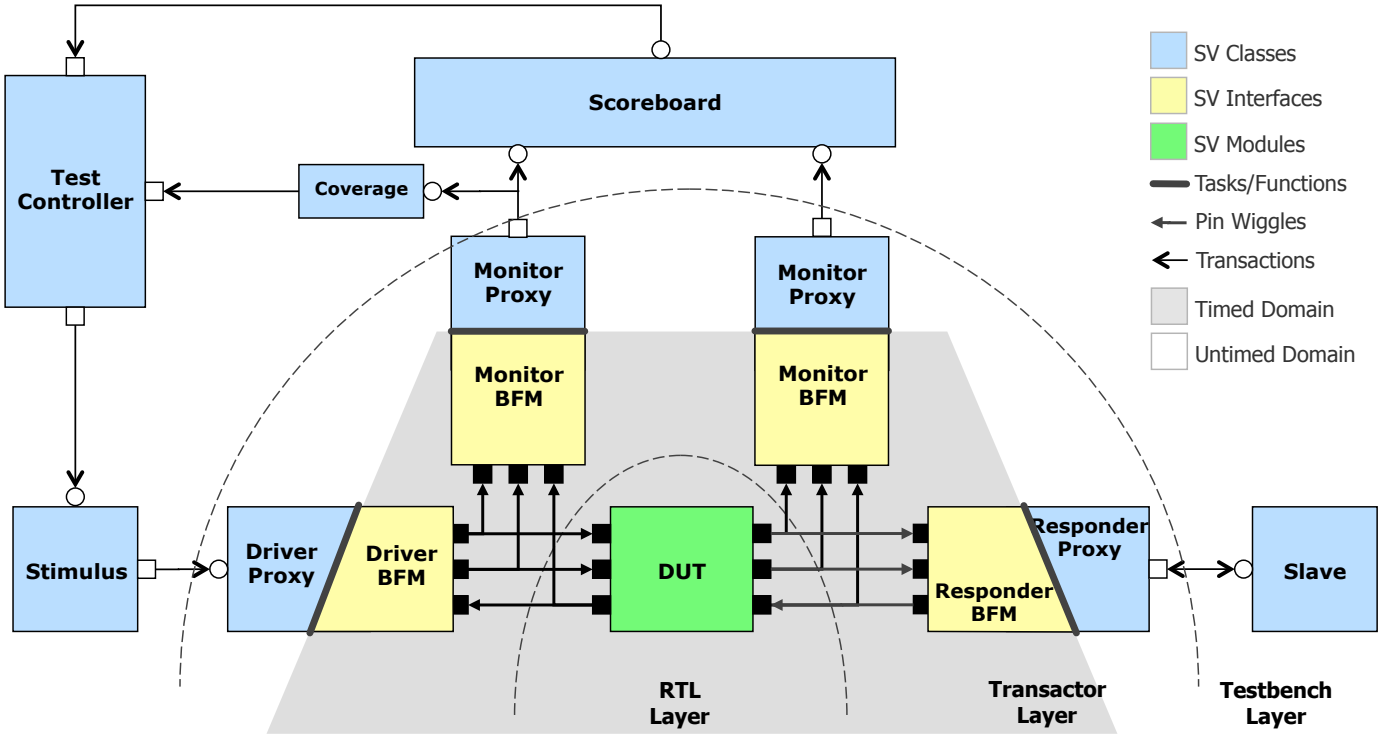
Testbench is Pivotal to Acceleration Speed-Up



UVM Layered Testbench

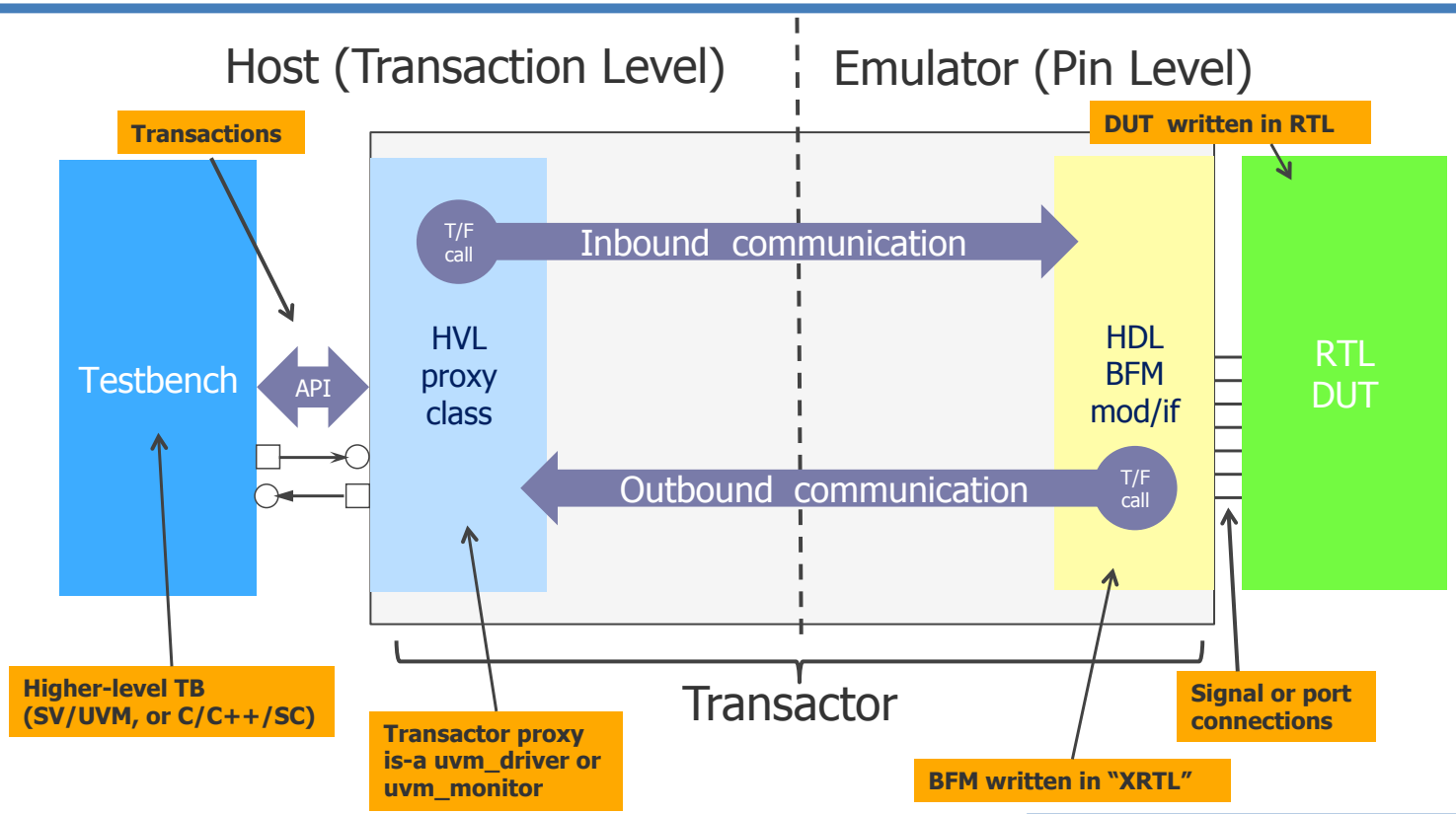


Unified UVM Simulation/Emulation Testbench

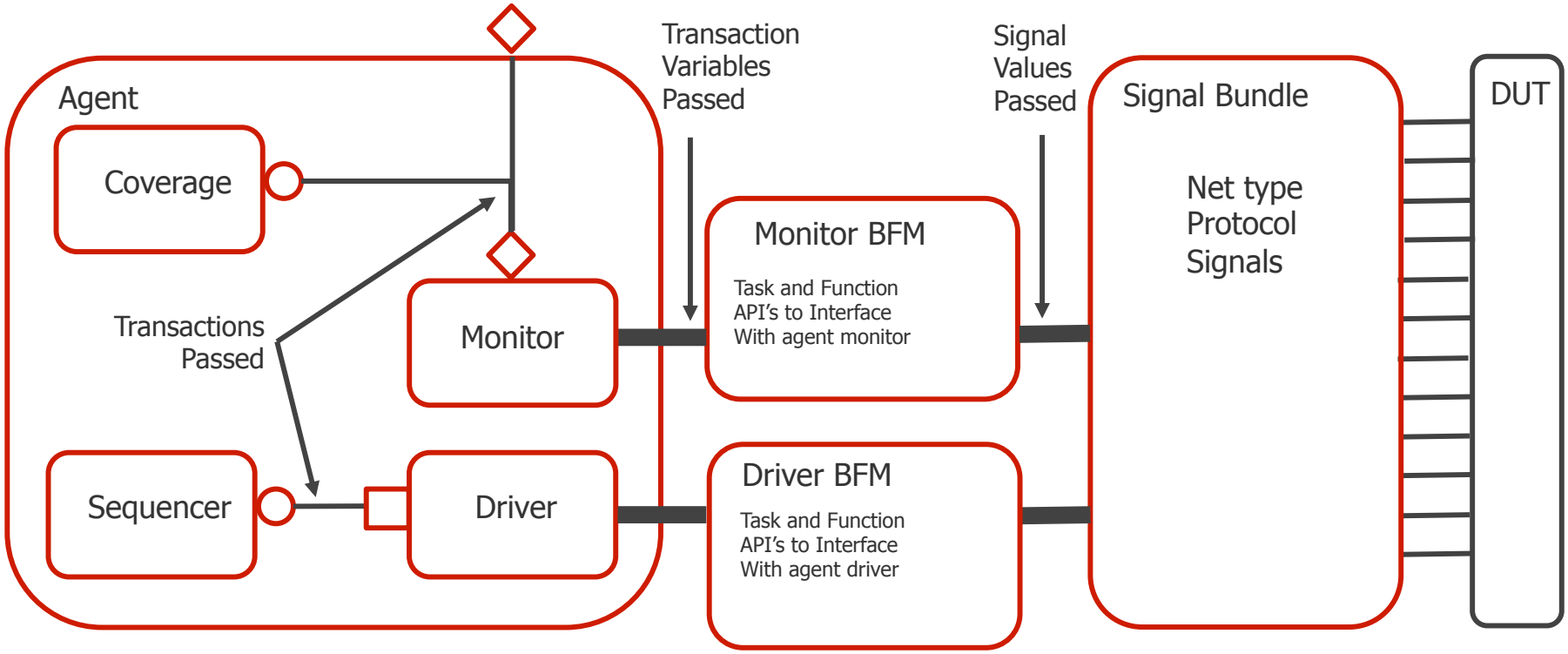


Transactors – Co-Emulation Building Blocks

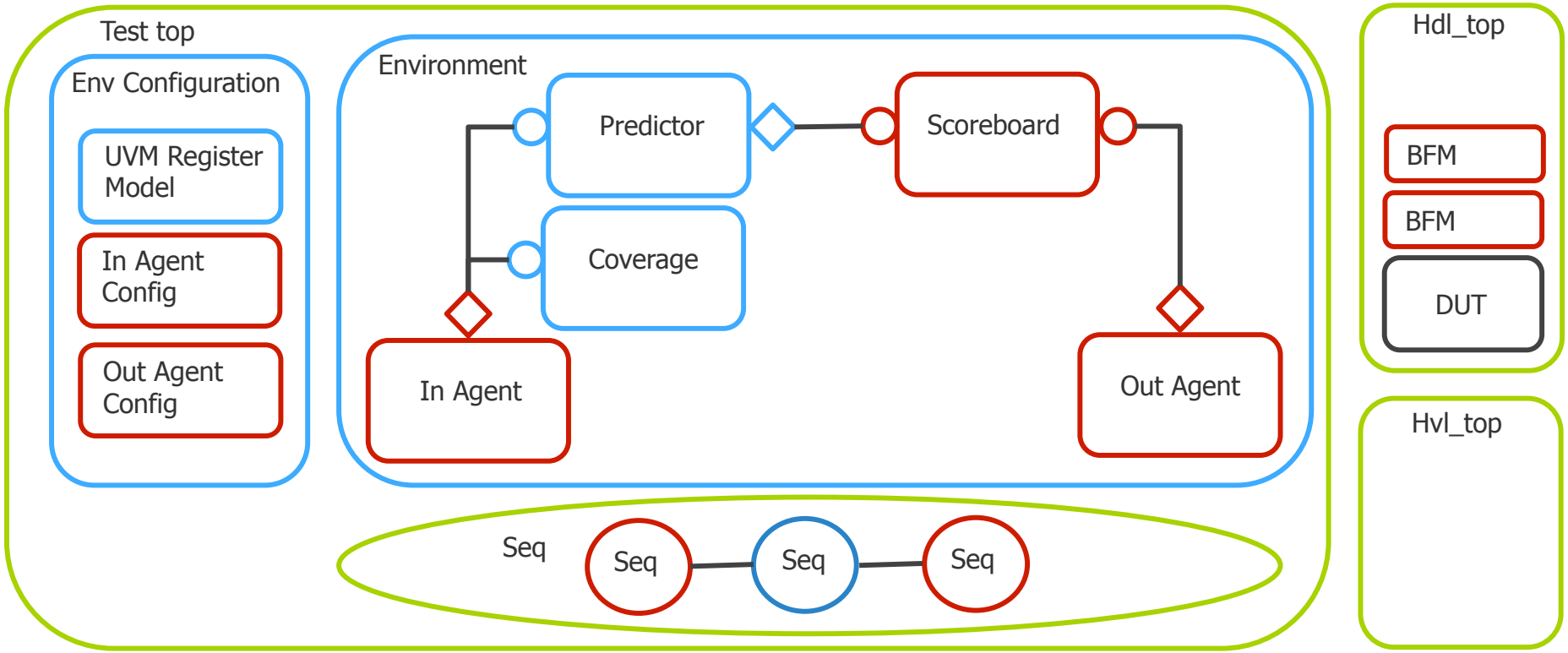
HDL BFM + HVL Proxy + HVL-HDL Channel



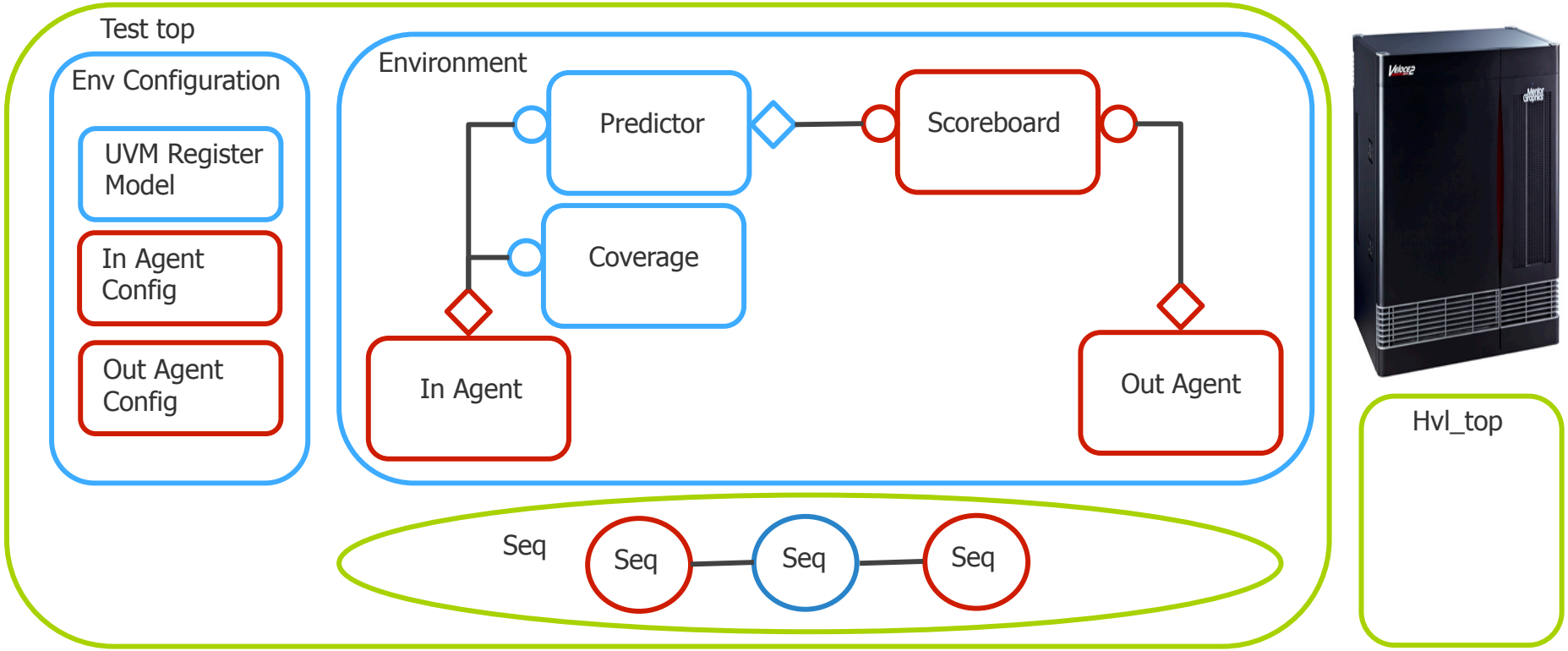
Agent and BFM – Data Flow



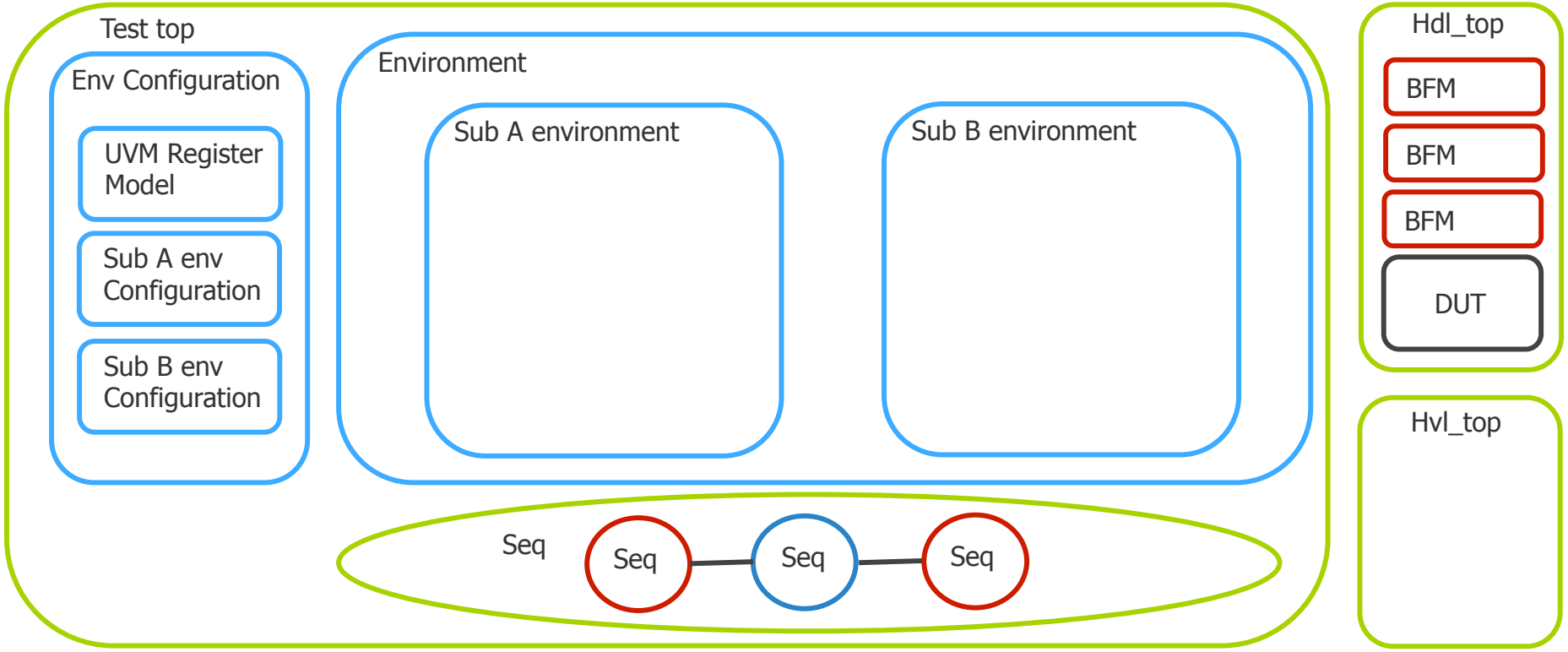
Block Level Test Bench Modules



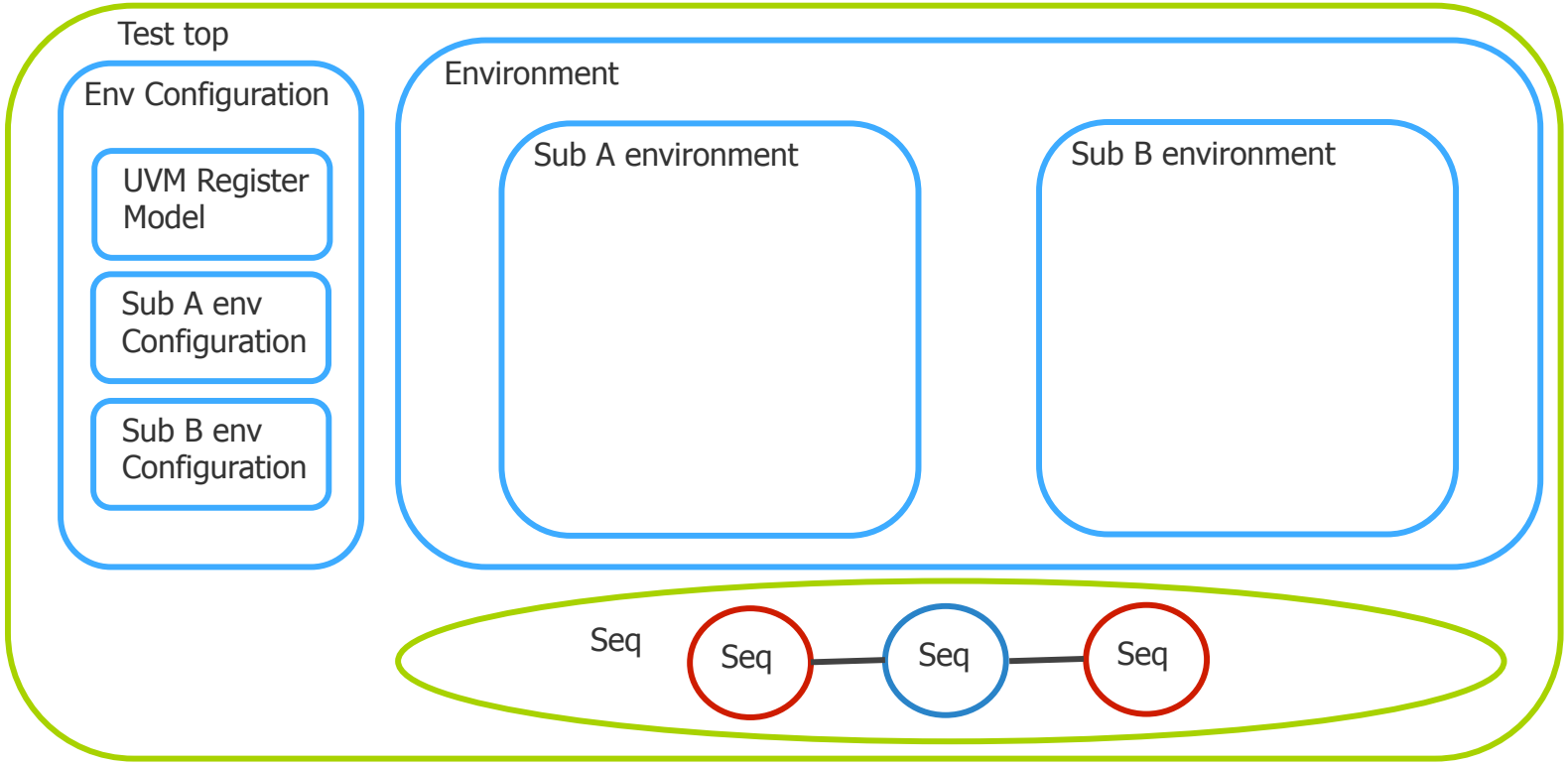
Block Level Test Bench Modules



Chip Level Test Bench Modules



Chip Level Test Bench Modules



Hvl_top

REUSE

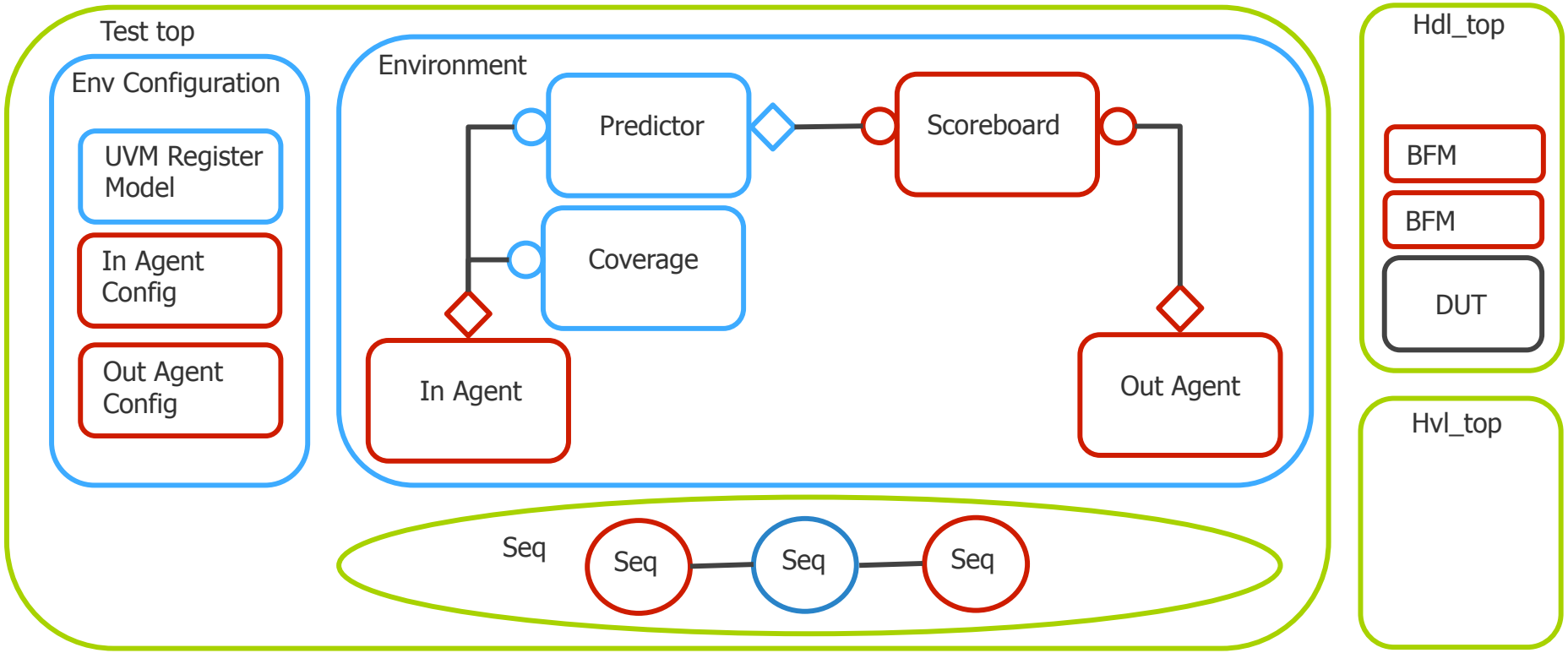
Characteristics of Reusable Components

- Consistent construction
  - Common constructor arguments
- Consistent initialization
  - Required information passed down through configuration hierarchy
- Recursive construction
  - Build their own sub-components
- Self-containment
  - Contain all required configuration and variables within self or sub-components
- Functionality compartmentalization
  - Group related functionality for reuse

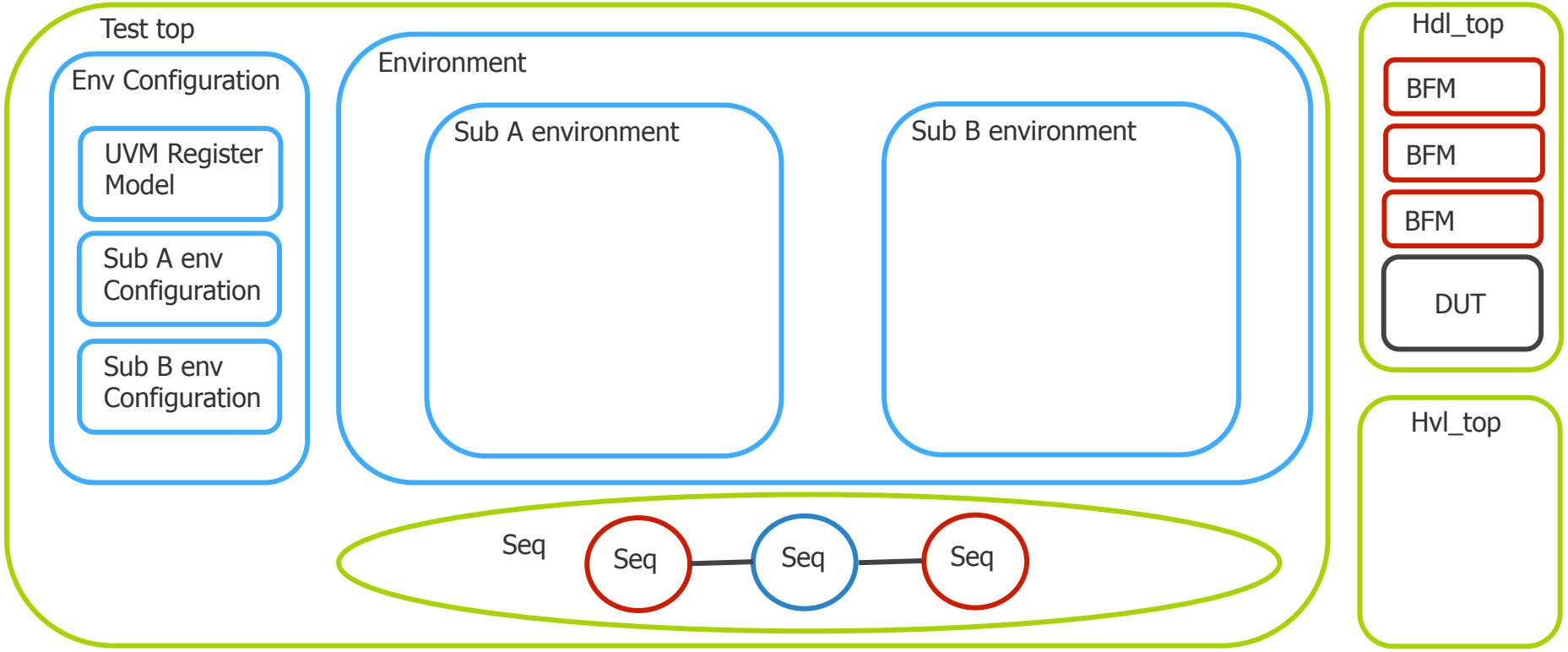
Types of Reuse

- Horizontal
  - Components and sequences across projects
  - Interface packages, utility packages
- Vertical
  - Components and sequences from block to top
  - Environment packages, utility packages
- Platform
  - Simulation and emulation
  - Same structure flow and stimulus

Block Level Test Bench Modules



Chip Level Test Bench Modules



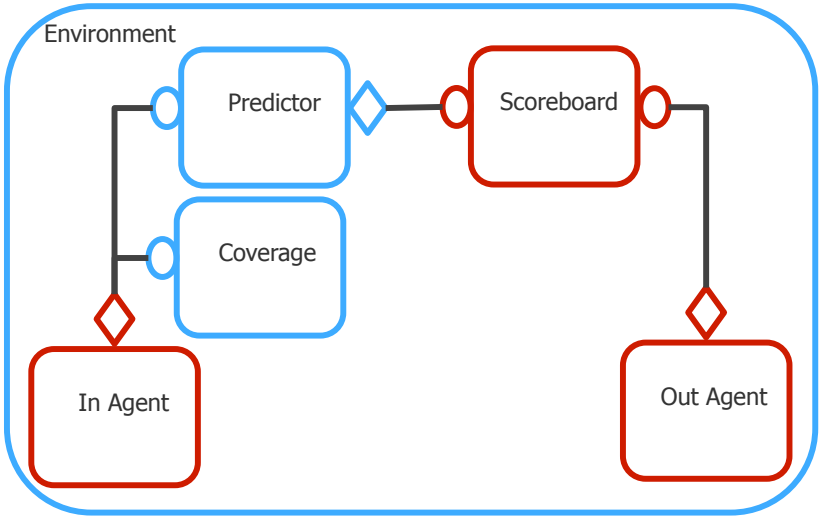
REGRESSION TESTING

Simulation and emulation in regression testing

- Each technology has strengths and weaknesses
  - Formal, CDC, simulation, emulation, etc
- Over reliance on any technology
  - Yields risk exposure due to its weakness
  - Increases project schedule
- Technologies are compilmentary
  - The strength of one covers the weakness of others
- Simulation and emulation are complimentary technologies
  - Speed, visibility, cost

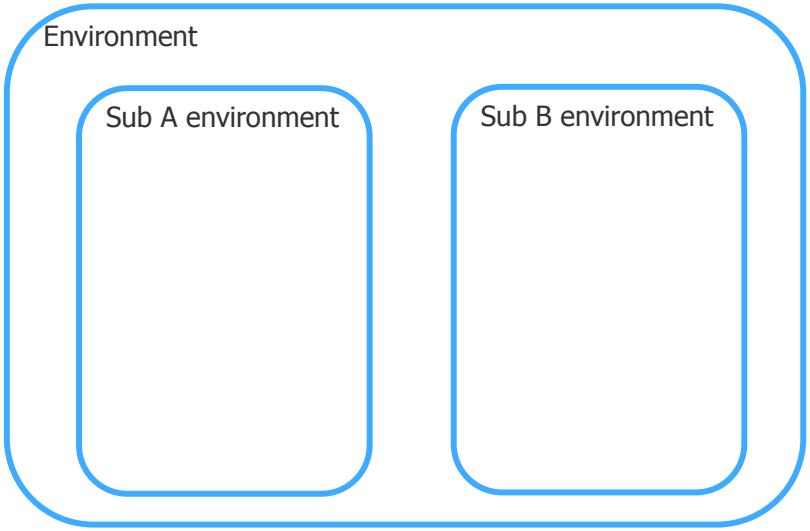
Block Level Testing for Feature Verification

- Simulation strength
  - Smaller design than full chip
  - Increased controlability of stimulus
  - Full visibility for debug
  - Native functional coverage support
- Transition to emulation
  - Development transitions from debug to regression



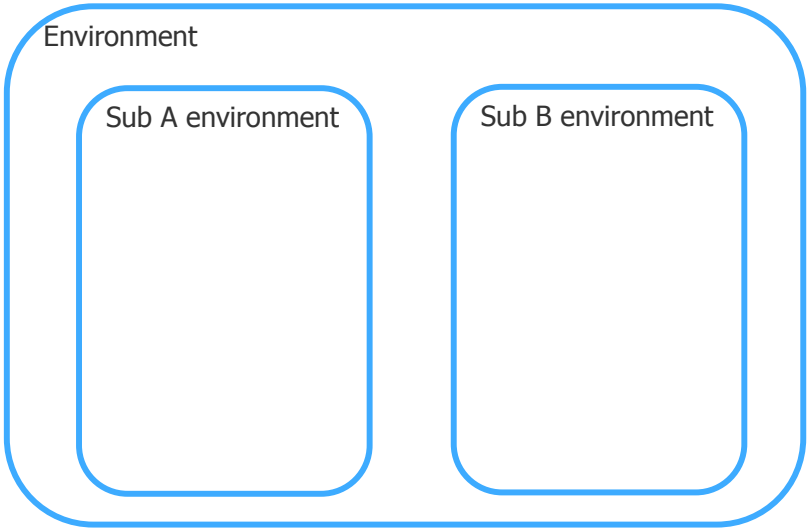
Chip Level Testing for Integration Verification

- Emulation strength
  - Large design leverages emulation concurrency
  - Broad stimulus
    - Traffic patterns testing sub-block integration
  - Only interconnect visibility required
- Transition to simulation
  - Reproduce errors discovered in sub blocks

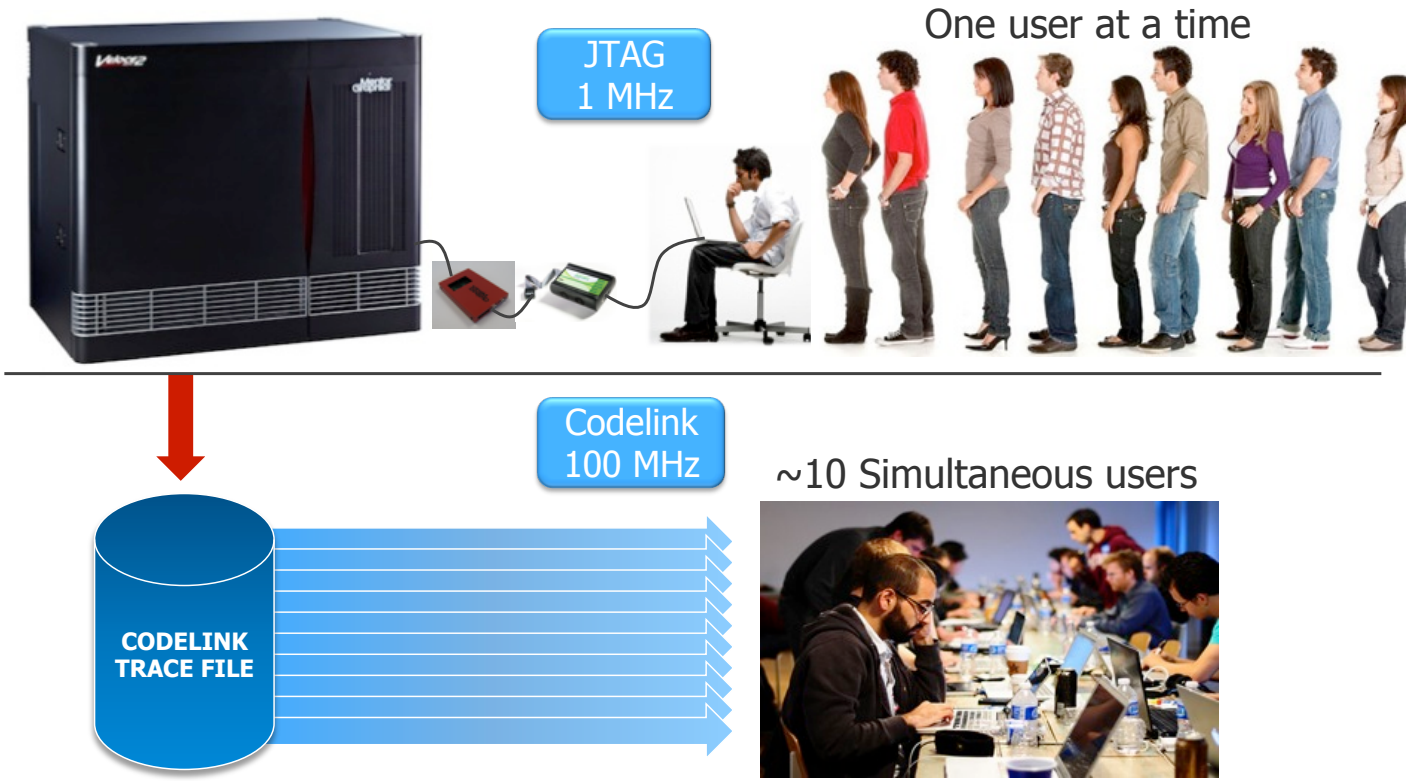


Chip Level Testing for Performance Verification

- Emulation strength
  - Emulation concurrency models design concurrency
  - Broad stimulus
    - Traffic patterns testing performance
    - Identify bottlenecks
  - Only interconnect visibility required
- Transition to simulation
  - Reproduce performance bottlenecks discovered in sub blocks

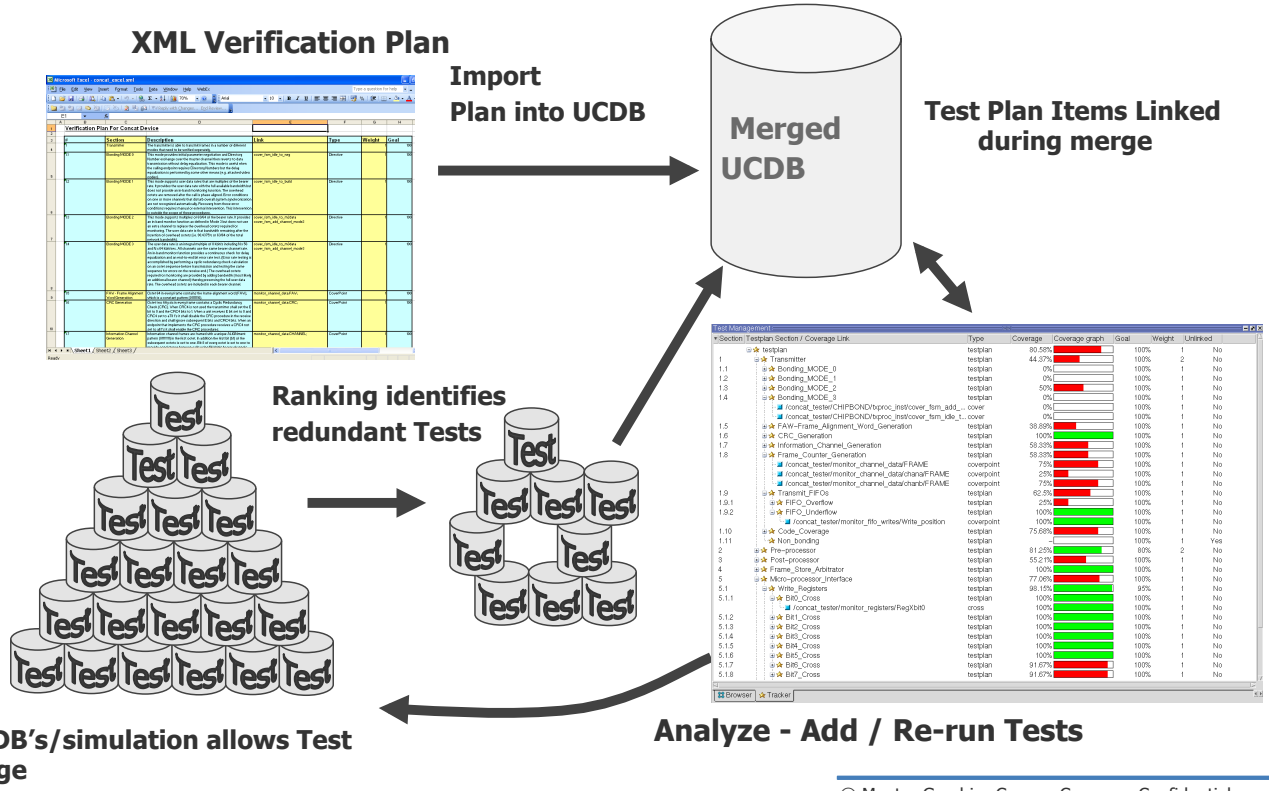


Chip Level Testing for SW Integration Verification



VERIFICATION MANAGEMENT

Verification management for closing coverage



Mentor®

A Siemens Business

www.mentor.com