



Universal Verification Methodology (UVM)

Verifying Blocks to IP to SOCs and Systems

Organizers:
Dennis Brophy
Stan Krolikoski
Yatin Trivedi



San Diego, CA
June 5, 2011

Workshop Outline

10:00am – 10:05am	Dennis Brophy	Welcome
10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
11:25am – 11:40am	Break	
11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A

Workshop Outline

✓ 10:00am – 10:05am	Dennis Brophy	Welcome
10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
11:25am – 11:40am	Break	
11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A



UVM Concepts and Architecture

Sharon Rosenberg
Cadence Design Systems

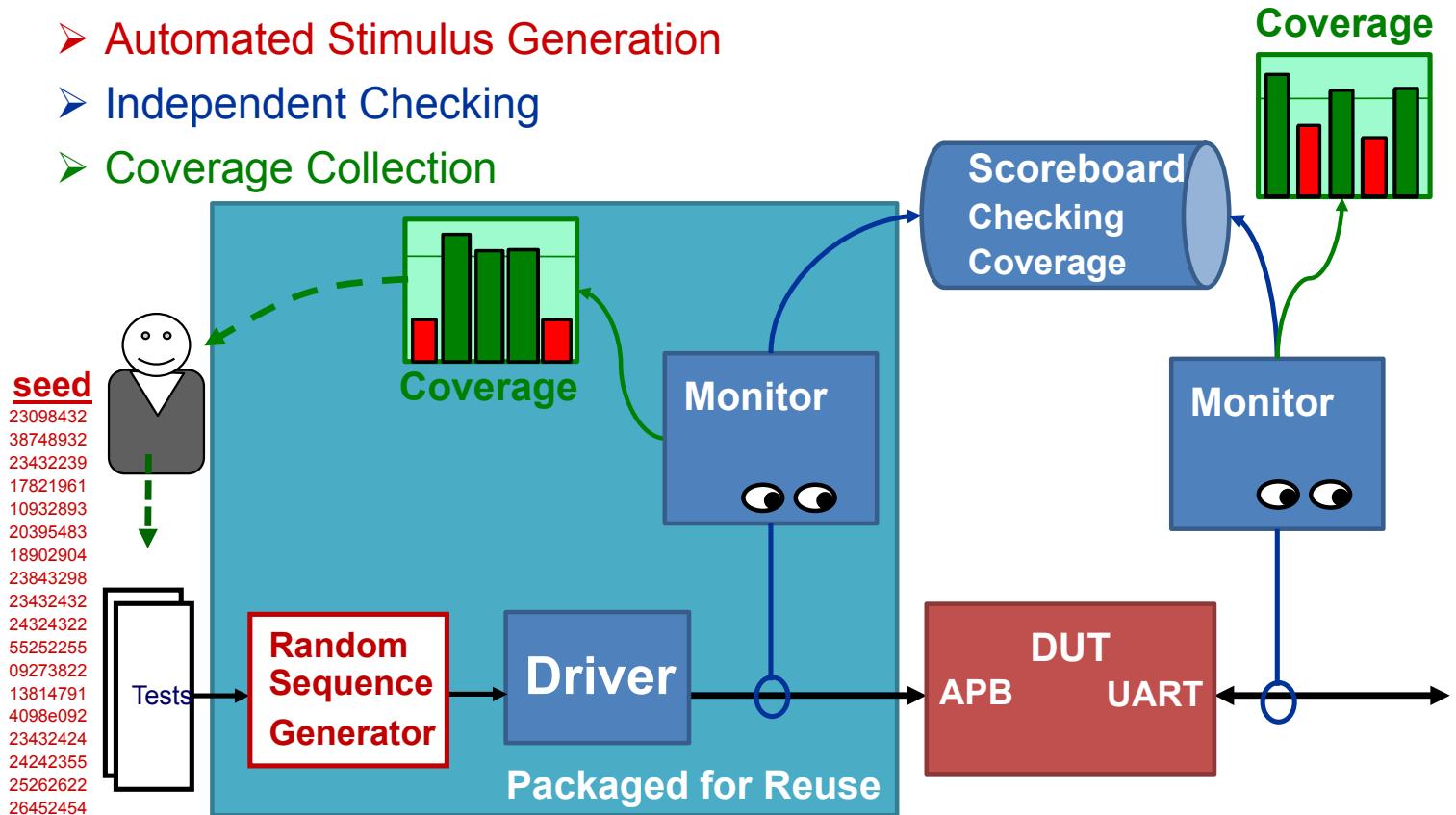


UVM Core Capabilities

- Universal Verification Methodology
 - A methodology and a class library for building advanced reusable verification components
 - Methodology first!
- Relies on strong, proven industry foundations
 - The core of the success is adherence to a standard (architecture, stimulus creation, automation, factory usage, etc')
- We added useful enablers and tuned a few to make UVM1.0 more capable
- This section covers the high-level concepts of UVM
 - Critical to successful deployment of UVM
 - Mature and proven

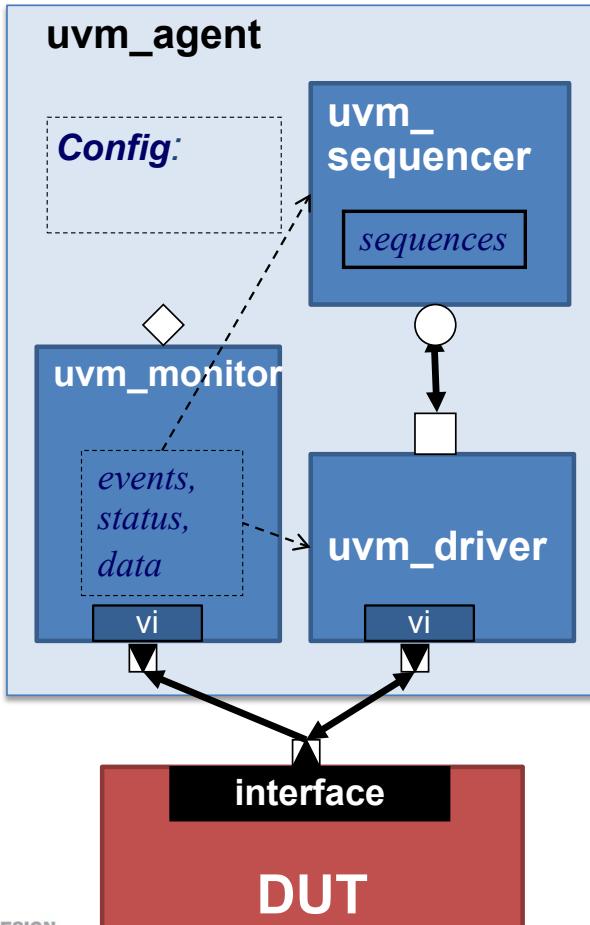
The Goal: Automation

- Coverage Driven Verification (CDV) environments
 - Automated Stimulus Generation
 - Independent Checking
 - Coverage Collection



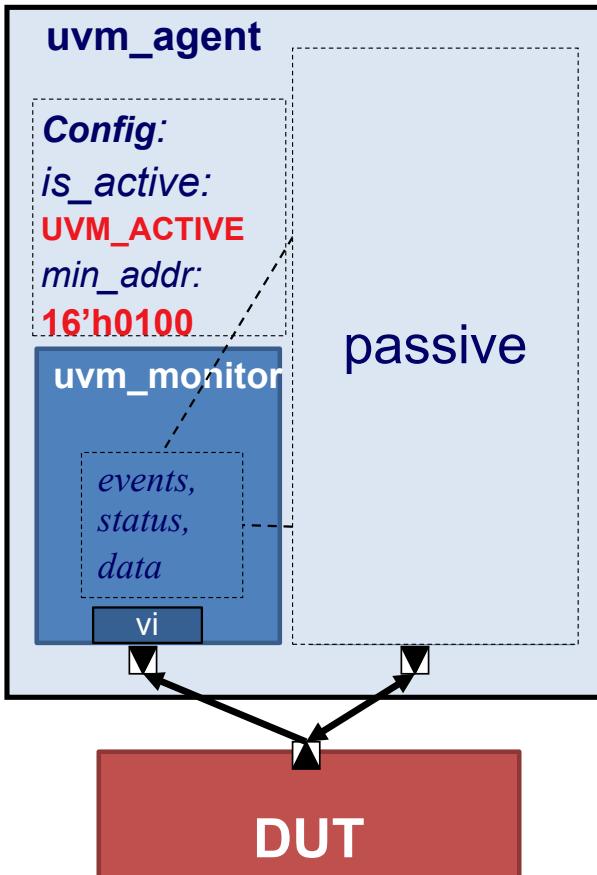
UVM Architecture:

Interface Level Encapsulation



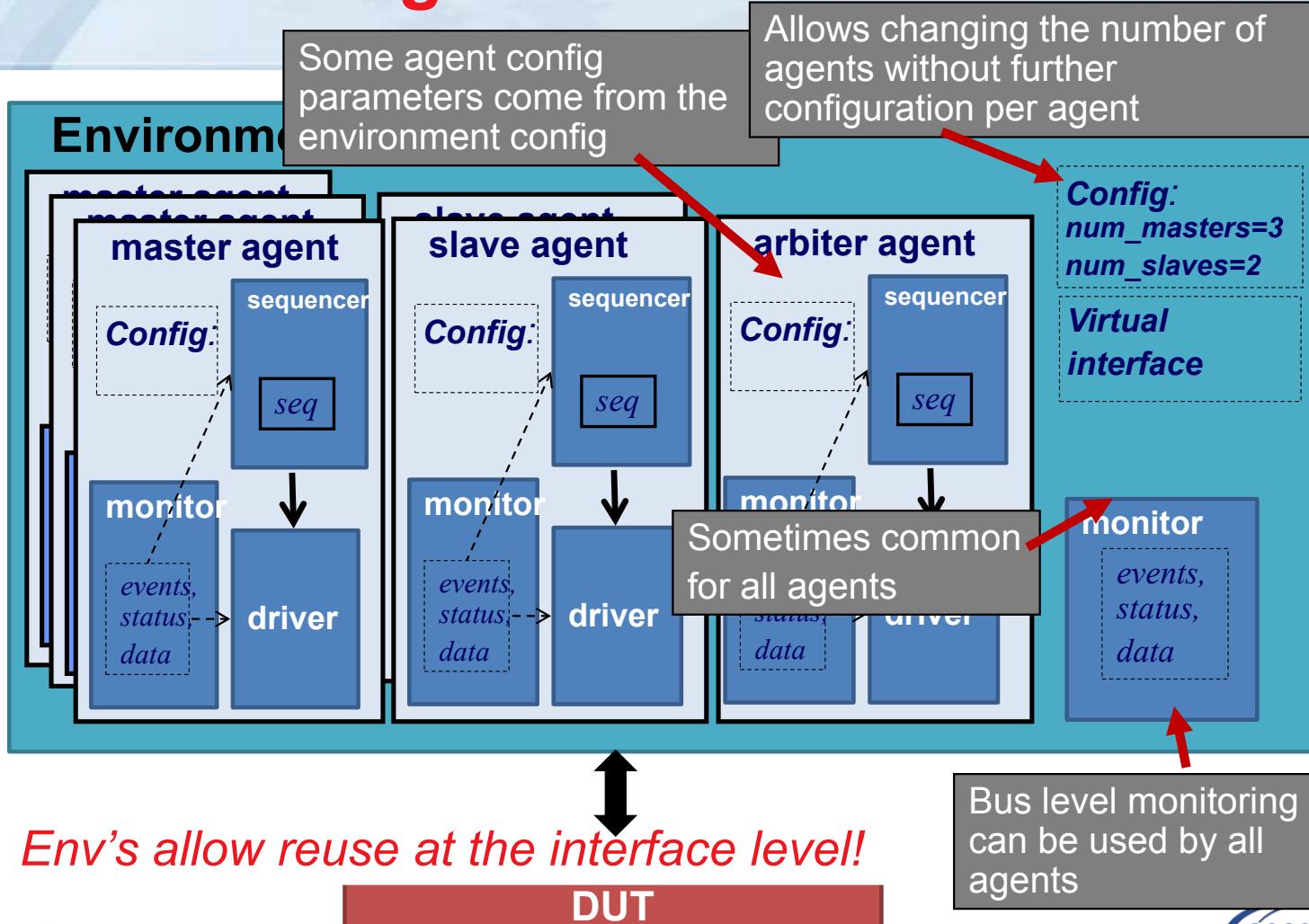
- Agents provide all the verification logic for a device in the system
- Instantiation and connection logic is done by the developer in a standard manner
- A Standard agent has:
 - Sequencer for generating traffic
 - Driver to drive the DUT
 - Monitor
- The monitor is independent of the driving logic
- Agent has standard configuration parameters for the integrator to use

Agent Standard Configuration



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

uvm: Configurable Bus Environment



UVM Configuration Mechanism

- The configuration mechanism allows a powerful way for attribute configuration
- Configuration mechanism advantages:
 - Mechanism semantic allows an upper component to override contained components values
 - No file changes are required
 - Can configure attributes at various hierarchy locations
 - Wild cards and regular expressions allow configuration of multiple attributes with a single command
 - Debug capabilities
 - Support for user defined types (e.g. SV virtual interfaces)
 - Run-time configuration support
 - Type safe solution

UVM1.0 Configuration Enhancements (Cont')

```
class uvm_config_db#(type T=int) extends uvm_resource_db#(T);  
    static function bit set ( uvm_component cntxt,  
        string inst_name,string field_name, T value);  
    static function bit get ( uvm_component cntxt,  
        string inst_name,string field_name, ref T value);  
    static function bit exists(...);  
    static function void dump();  
    static task wait_modified(...);  
endclass
```

Check if configuration exists

Dump the data base

Wait for value to be set in
the uvm_config_db

```
// run-time configuration example:
```

```
task mycomp::run_phase (uvm_phase phase);  
    uvm_config_db#(int)::set(this, "*", "field", 10);  
    #10; uvm_config_db#(int)::set(this, "a.b.c", "field", 20);  
endtask
```

Note – all uvm_config_db
functions are static so they
must be called using the ::
operator

Virtual Interface Configuration Example

```
function void ubus_bus_monitor:: connect_phase( uvm_phase phase);
    if (!uvm_config_db#(virtual ubus_if)::  
        get(this, "", "vif", vif),  
    else  
        `uvm_error("NOVIF", {"virtual interface must be set for: ",  
                           get_full_name(), ".vif"})  
endfunction: connect_phase
```

Built-in checking

```
// setting the virtual interface from the top module
module ubus_top;  
    ...  
    ubus_if ubus_if0(); // instance of the interface  
  
    initial begin  
        uvm_config_db#(virtual ubus_if)::  
            set(null, "* .ubus_demo_tb0.ubus0", "vif", ubus_if0);  
        run_test();  
        ...  
    end  
endmodule
```

Setting in the top removes hierarchy dependencies in the testbench, allows consistency and other configuration capabilities

Where SV Language Stops and UVM Begins

Example: Data Items

```
class data_packet_c ;
    string          pkt_name;
    rand pkt_header _c header;
    rand byte       payload [ ];
    byte           parity;
    rand parity_e   parity_type;
    rand int        ipg_delay;
endclass
```

Does language alone support all the necessary customization operations?

- Randomization
- Printing
- Cloning
- Comparing
- Copying
- Packing
- Transaction Recording

No! Only randomization is defined
in the SystemVerilog LRM

UVM provides the rest!

Enabling Data Item Automation

```
class data_packet_c ; extends uvm_sequence_item;
  string           rev_no = "v1.1p";
  rand pkt_hdr _c header; //class:{dest_addr, pkt_length}
  rand byte        payload [ ];
  byte             parity;
  rand parity_e    parity_type;
  rand int          ipg_delay;

// field declarations and automation flags
`uvm_object_utils_begin(data_packet_c)
  `uvm_field_string( rev_no,   UVM_DEFAULT+ UVM_NOPACK)
  `uvm_field_object( header,   UVM_DEFAULT)
  `uvm_field_array_int( payload, UVM_DEFAULT)
  `uvm_field_int( parity,    UVM_DEFAULT)
  `uvm_field_enum( parity_e,  parity_type, UVM_DEFAULT)
  `uvm_field_int( ipg_delay, UVM_DEFAULT + UVM_NOCOMPARE)
`uvm_object_utils_end
// Additional: constraints
endclass : data_packet_c
```

derived from **uvm_sequence_item**

Enables all automation for **data_packet_c** fields

Specify field level flags:
UVM_NOCOMPARE, **UVM_NOPRINT**, etc.

Data Type Automation

```
// two instances of data_packet_c
data_packet_c packet1, packet2;
initial begin
    // create and randomize new pack
    packet1 = new("my_packet");
    assert(packet1.randomize());
    // print using UVM automation
    packet1.print();
    // copy using UVM automation
    packet2 = new("copy_packet");
    packet2.copy(packet1);
    packet2.rev_no = "v1.1s";
    // print using UVM tree printer
    packet2.print(
        uvm_default_tree_printer);
end
```

UVM also allows manual implementation for performance or other reasons

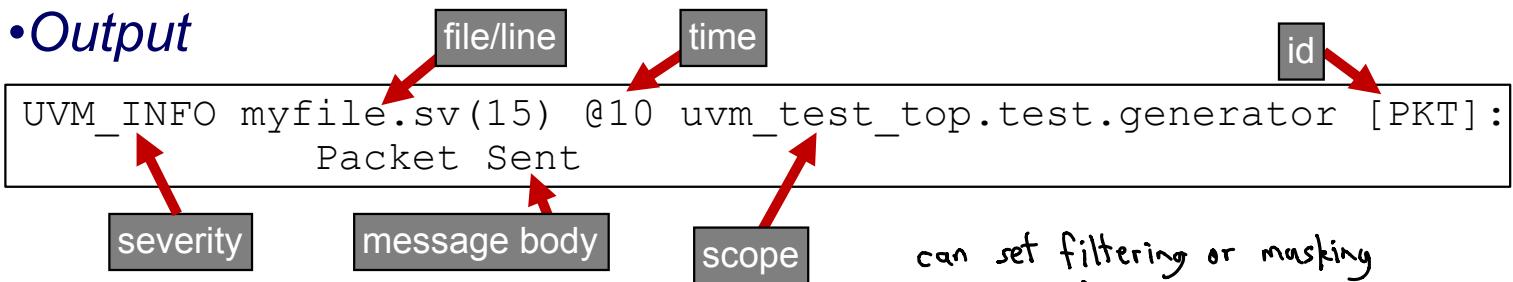
Name	Type	Value
my_packet	data_packet_c	@479
rev_no	string	v1.1p
header	pkt_header_c	@520
dest_addr	integral	'h25
pkt_length	integral	'd29
payload	da(integral)	-
copy_packet: (data_packet_c@489) {		
rev_no: v1.1s		
header: (pkt_header_c@576) {		
dest_addr: 'h25		
pkt_length: 'd29		
}		
payload: {		
[0]: 'h2a		
[1]: 'hdb		
... ...		
[28]: 'h21		
}		
parity: 'hd9		
parity_type: GOOD_PARITY		
ipg_delay: 'd20		
}		

UVM Messaging Facility

- Messages print trace information with advantages over \$display:
- Aware of its hierarchy/scope in testbench
- Allows filtering based on **hierarchy**, **verbosity**, and **time**

```
`uvm_info("PKT", "Packet Sent", UVM_LOW);
```

Output



- Simple Messaging:
 - `'uvm_*(string id, string message, <verbosity>);'`
 - Where * (severity) is one of **fatal**, **error**, **warning**, **info**
 - **<verbosity>** is only valid for `uvm_info`

UVM Sequences

- A sequencer controls the generation of random stimulus by executing **sequences**
- A sequence captures meaningful streams of transactions
 - A simple sequence is a random transaction generator
 - A more complex sequence can contain timing, additional constraints, parameters
- **Sequences:**
 - Allow reactive generation – react to DUT
 - Have many built-in capabilities like interrupt support, arbitration schemes, automatic factory support, etc
 - Can be nested inside other sequences
 - Are reusable at higher levels

UVM Tests and Testbenches

```
module top();
```

```
class test extends ...
```

```
class tb extends uvr
```

```
Module VC  
Scoreboard  
coverage
```



```
Virtual Sequencer
```

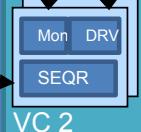
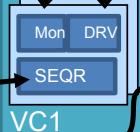
```
DUT
```

```
CPU
```

```
Mem
```

```
Periph
```

```
Periph
```



```
VC1
```

```
VC 2
```

- Placing all components in the test requires lot of duplication
- Separate the env configuration and the test
 - TB class instantiates and configures reusable components
- Tests instantiate a testbench
 - 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

UVM Simulation Phases

- When using 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 are 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
- UVM provides set of standard phases enabling VIP plug&play
 - Allows orchestrating the activity of components that were created by different resources

UVM Simulation Phases

UVM component's built-in phases - run in order

Note: All phases except run() execute in zero time

build	Build Top-Level Testbench Topology
connect	Connect environment topology
end of elaboration	Post-elaboration activity (e.g. print topology)
start_of_simulation	Configure verification components
run reset configure main shutdown	tasks - Run-time execution of the test
extract	Gathers details on the final DUT state
check	Processes and checks the simulation results.
report	Simulation results analysis and reporting

All phase names have postfix “_phase”

UVM Testbench Example

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

apb_env apb;      // APB OVC
ahb_env ahb;      // AHB OVC
bridge_mod_env bridge_mod ; // Module OVC

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(uvm_active_passive_enum)::set(this,
        "apb.slave*", "is_active", UVM_ACTIVE);
    uvm_config_db#(int)::set(this, "ahb", "master_num", 3);
    uvm_config_db#(uvm_active_passive_enum)::set(this,
        "ahb.slave[0]", "is_active", UVM_PASSIVE);
    apb = apb_env::type_id::create("apb", this);
    ahb = ahb_env::type_id::create("ahb", this);
    bridge_mod = bridge_mod_env::type_id::create("bridge mod", this);
endfunction

endclass: bridge_tb
```

Extends from uvm_env

Instances of reusable verification components and module verify component

Configure using wildcards

Create and build using a standard mechanism

The test creates an instance of the testbench, overrides constraints, and sets the default sequences

Overriding SV Components and Data Objects

- UVM Provides a mechanism for overriding the default data items and objects in a testbench
- “Polymorphism made easy” for test writers

Replace ALL instances:

```
object::type_id::set_type_override(  
    derived_obj::get_type() )
```

Example:

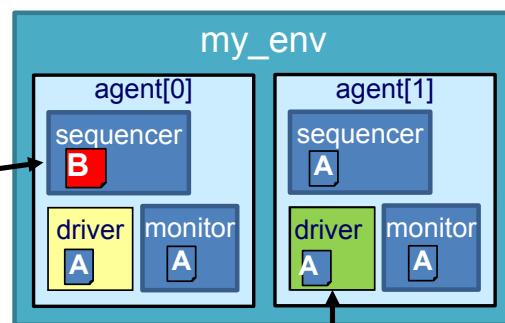
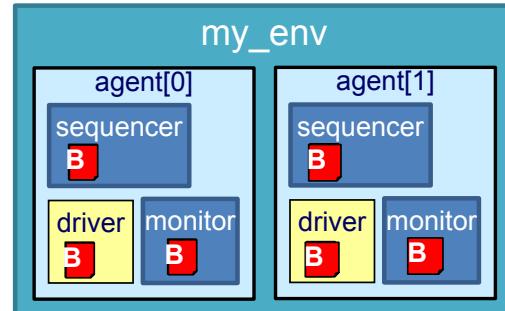
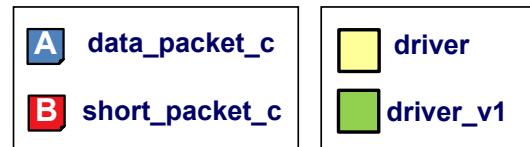
```
data_packet_c::type_id::set_type_override  
    (short_packet_c::get_type() );
```

Replace specific instances:

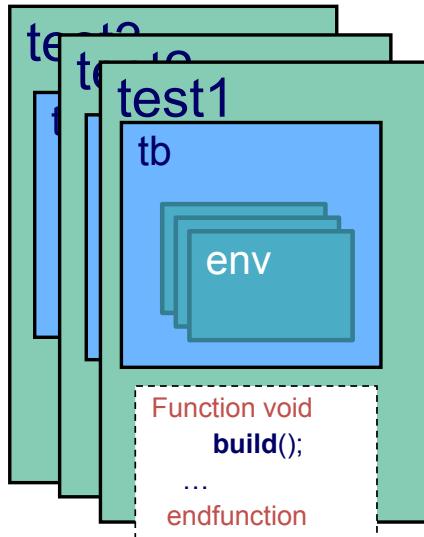
```
object::type_id::set_inst_override  
    (derived_obj::get_type(), "hierarchical_path");
```

Example:

```
data_packet_c::type_id::set_inst_override  
    (short_packet_c::get_type(),  
     "my_env.agent[0].sequencer");  
  
my_driver::type_id::set_inst_override  
    (driver_v1::get_type(), "my_env.agent[1]");
```



The Test Launching Mechanism



```
module top()
  import ... // uvm lib, tests, and packages
  IF1 IF2 IF3 rst clks
  DUT
  initial begin
    run_test();
  end
endmodule : top
```

DUT snapshot

Creates test and starts simulation phases for all components

Allows execution

Compile the entire test suite together and use command-line option to select a test:

```
% <sim> -f run.f +UVM_TESTNAME=test3
```

Extensions Using Callbacks

- Like the factory, callbacks are a way to affect an existing component from outside
- The SystemVerilog language includes built-in callbacks
 - e.g. post_randomize(), pre_body()
- Callbacks requires the developer to predict the extension location and create a proper hook
- Callbacks advantages:
 - They do not require inheritance
 - Multiple callbacks can be combined

UVM Report Catcher Callback

Goal: Message Manipulation

```
class my_catcher extends uvm_report_catcher
virtual function action_e catch();
  if(get_severity() == UVM_ERROR && get_id() == "MYID")
    begin
      set_severity(UVM_INFO);
      set_action(get_action() - UVM_COUNT);
    end
  return THROW; // can throw the message for more
  manipulation or catch it to avoid further processing
endfunction
endclass
```

This example demotes MYID to be an Info

```
// In testbench run phase
my_catcher catcher = new;
uvm_report_cb::add(null,catcher);
`uvm_error("MYID", "This one should be demoted")
#100;
catcher.callback_mode(0); // disable the catcher
`uvm_error("MYID", "This one should not be demoted")
```

- can disable a callback using the built-in callback_mode() method

Command-line Processor Class

- Provide a vendor independent general interface to the command line arguments
- Supported categories:
 - Basic Arguments and values
 - `get_args`, `get_args_matches`
 - Tool information
 - `get_tool_name()`, `get_tool_version()`
 - Built-in UVM aware Command Line arguments
 - Supports setting various UVM variables from the command line such as verbosity and configuration settings for integral types and strings
 - `+uvm_set_config_int`, `+uvm_set_config_string`

Command-line Processor Example

```
class test extends uvm_test;  
...  
function void start_of_simulation();  
    uvm cmdline_processor clp;  
    string arg_values[$];  
    string tool, version;  
    clp = uvm cmdline_processor::get_inst();  
    tool = clp.get_tool_name();  
    version = clp.get_tool_version();  
    `uvm_info("MYINFO1", ("Tool: %s, Version : %s", tool, version,  
        UVM_LOW)  
    void'(clp.get_arg_values("+foo=", arg_values));  
    `uvm_info("MYINFO1", "arg_values size : %0d", arg_values.size(),  
        UVM_LOW));  
    for(int i = 0; i < arg_values.size(); i++) begin  
        `uvm_info("MYINFO1", "arg_values[%0d]: %0s", i, arg_values[i],  
            UVM_LOW));  
    end  
endfunction  
endclass
```

Fetching the command line processor singleton class

Use the class methods

Get argument values

UVM Concepts Summary

- UVM Basics are proven and widely in use with all simulators
- Provides both reuse and productivity
- UVM1.0 adds additional features to complement and tune the existing capabilities
 - UVM 1.1 includes bug fixes after initial users' feedback
- If you have a new project, ***UVM is the way to go!***

Workshop Outline

✓ 10:00am – 10:05am	Dennis Brophy	Welcome
✓ 10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
11:25am – 11:40am	Break	
11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A



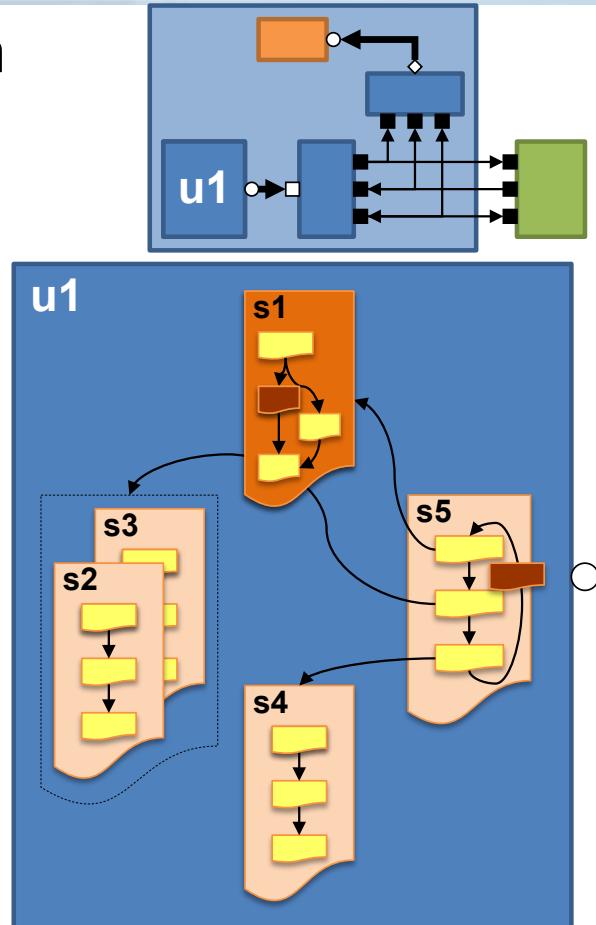
UVM Sequences & Phasing

Tom Fitzpatrick
Mentor Graphics



Sequences

- Decouple stimulus specification from structural hierarchy
 - Simple test writer API
- Sequences define transaction streams
 - May start on any matching sequencer
- Sequences can call children
- Built-in `get_response()` task
- Sequences & transactions customizable via the factory
- Driver converts transaction to pin wiggles



Using Sequences And Sequence Items

- A sequence is a UVM object – to start it:
 - Construction using the factory:
 - `spi_tfer_seq spi_seq =
spi_tfr_seq::type_id::create("spi_seq");`
 - Configure - explicitly or via constrained randomization
 - Start execution on the target sequencer:
 - `spi_seq.start(spi_sequencer);`
- Within a sequence a `sequence_item` is:
 - Constructed
 - Scheduled on a sequencer with `start_item()`
 - Blocking call that returns when the driver is ready
 - Configured – explicitly or via constrained randomization
 - Consumed with `finish_item()`

Sequence_Item Example

```
class spi_seq_item extends uvm_sequence_item;  
  
// UVM Factory Registration Macro  
`uvm_object_utils(spi_seq_item)  
  
// Data Members (Outputs rand, inputs non-rand)  
rand logic[127:0] spi_data;  
rand bit[6:0] no_bits;  
rand bit RX_NEG;  
  
// Analysis members:  
logic[127:0] mosi;  
logic[7:0] cs;  
  
// Methods  
extern function new(string name = "spi_seq_item");  
extern function void do_copy(uvm_object rhs);  
extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);  
extern function string convert2string();  
extern function void do_print(uvm_printer printer);  
extern function void do_record(uvm_recorder recorder);  
  
endclass:spi_seq_item
```

- Data fields:
- Request direction rand (stimulus generation)
- Response direction non-rand

- UVM Object Methods
- These methods get generated by the automation macros
- Write them yourself to improve performance if desired

Basic Sequence-Driver API

```
class my_seq extends uvm_sequence #(my_req, my_rsp);  
task body() ;  
    for(int unsigned i = 0; i < 20000; i++) begin  
        req = my_req::type_id::create("req");  
        start_item(req);  
        assert(req.randomize());  
        finish_item(req);  
        get_response(rsp);  
    end  
endtask  
endclass
```

Sequence parameterized by request/response types

body() method defines what happens

```
class my_driver extends uvm_driver;  
task run_phase(uvm_phase phase);  
    ... begin  
        seq_item_port.get_next_item(req);  
        drive_transfer(req);  
        rsp.set_id_info(rsp);  
        seq_item_port.item_done();  
        seq_item_port.put_response(rsp);  
    end  
endtask  
endclass
```

Sequence API Options

Explicit

```
req = my_req::type_id::create("req");
start_item(req);
assert(req.randomize());
finish_item(req);
get_response(rsp);
```

Using Macros

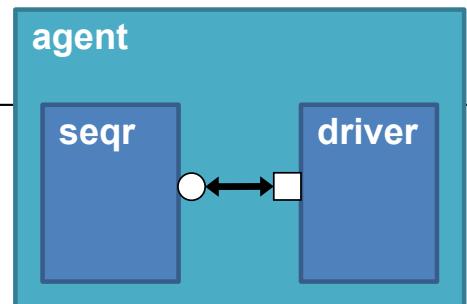
```
`uvm_do(req)
get_response(rsp);
```

- Macros allow constraints to be passed in
- Macros require pre-defined callbacks to modify behavior
- Multiple macro variations
- Explicit API provides greater control and easier debug

Sequencer and Driver

```
typedef uvm_sequencer#(my_req,my_rsp) my_sequencer;
class my_master_agent extends uvm_agent;
    function void build_phase(uvm_phase phase);
        void'(uvm_config_db #(bitstream_t)::get(this,"", "is_active", is_a));
        if(is_a) begin
            seqr = my_sequencer::type_id::create("seqr", this);
            driver = my_driver::type_id::create("driver", this);
        end
    endfunction
    function void connect_phase(uvm_phase phase);
        if(is_a)
            driver.seq_item_port.connect(seqr.seq_item_export);
    endfunction
endclass
```

By default, you don't need
to extend uvm_sequencer



Starting a Sequence

- Explicit

```
task xxx_phase(uvm_phase phase);  
    my_seq.start(seqr);
```

“xxx_phase” can be
any run-time phase

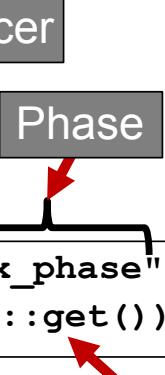
- Implicit (Default Sequence)
 - Using wrapper

```
uvm_config_db#(uvm_object_wrapper)::set(this,"agent.sqr.xxx_phase",  
    "default_sequence", my_seq::type_id::get());
```

- Using instance

```
myseq = my_seq::type_id::create("myseq");  
uvm_config_db#(uvm_sequence_base)::set(this,"agent.sqr.xxx_phase",  
    "default_sequence", myseq);  
my_seq.set_priority(200);
```

Sequence



Calling start()

```
task xxx_phase(uvm_phase phase);
    my_seq.start(seqr);
    uvm_sequencer_base parent = null,
    integer priority = 100,
    bit call_pre_post = 1;
```

on sequencer

Stimulus
code

```
my_seq.pre_start()
my_seq.pre_body();
parent.pre_do(0);
parent.mid_do(this);
my_seq.body();
parent.post_do(this);
my_seq.post_body();
my_seq.post_start();
```

If call_pre_post ==1

If parent!=null

Sequence Library (Beta)

```
class uvm_sequence_library #(type REQ=int,RSP=REQ)
    extends uvm_sequence #(REQ,RSP);
```

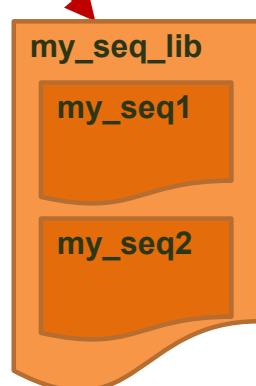
Sequence Library
IS-A Sequence

```
class my_seq_lib extends uvm_sequence_library #(my_item);
    `uvm_object_utils(my_seq_lib)
    `uvm_sequence_library_utils(my_seq_lib)
    function new(string name="";
        super.new(name);
        init_sequence_library();
    endfunction
    ...
endclass
```

```
class my_seq1 extends my_seq;
    `uvm_object_utils(my_seq1)
    `uvm_add_to_seq_lib(my_seq1,my_seq_lib)
    ...
endclass
```

```
class my_seq2 extends my_seq;
    `uvm_object_utils(my_seq2)
    `uvm_add_to_seq_lib(my_seq2,my_seq_lib)
    ...
endclass
```

`uvm_sequence_utils
`uvm_sequencer_utils
et. al. deprecated



Sequence Library

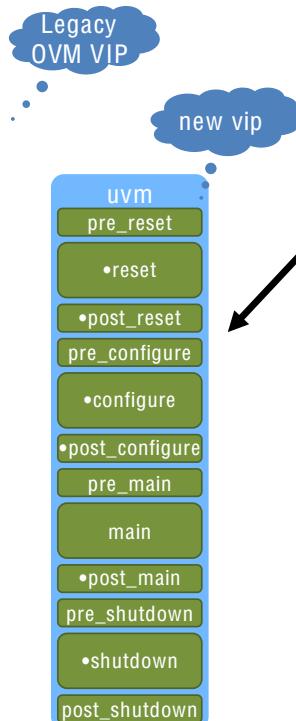
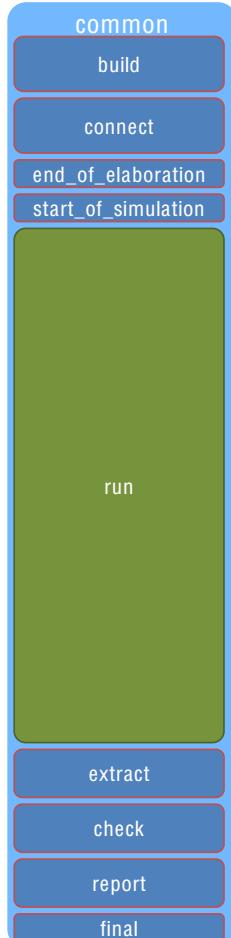
```
uvm_config_db #(uvm_sequence_lib_mode)::set(this,  
    "sequencer.xxx_phase", "default_sequence.selection_mode", MODE);
```

```
typedef enum  
{ UVM_SEQ_LIB RAND,  
  UVM_SEQ_LIB RANDC,  
  UVM_SEQ_LIB ITEM,  
  UVM_SEQ_LIB USER  
} uvm_sequence_lib_mode,
```

- Random sequence selection
- Random cyclic selection
- Emit only items,no sequence execution
- User-defined random selection

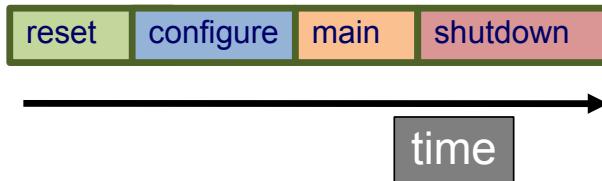
```
function int unsigned select_sequence(int unsigned max);  
  // pick from 0 <= select_sequence < max;  
endfunction
```

UVM Phasing



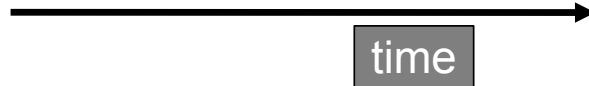
Several new runtime phases
in parallel with run_phase()

To simplify examples, these
slides will show a reduced set
of phases



Phase Synchronization

- By default, all components must allow all other components to complete a phase before all components move to next phase



Phase Semantics

- In UVM Reference Guide, the semantics for each phase are defined, e.g.

reset

Upon Entry

- Indicates that the hardware reset signal is ready to be asserted.

Typical Uses

- Assert reset signals.
- Components connected to virtual interfaces should drive their output to their specified reset or idle value.
- Components and environments should initialize their state variables.
- Clock generators start generating active edges.
- De-assert the reset signal(s) just before exit.
- Wait for the reset signal(s) to be de-asserted.

Exit Criteria

- Reset signal has just been de-asserted.
- Main or base clock is working and stable.
- At least one active clock edge has occurred.
- Output signals and state variables have been initialized.

uvm_component Basic API

Implement these to specify behavior for a specific phase; threads started here are auto-killed

```
task/function <name>_phase(uvm_phase phase)
    phase.raise_objection(this);
    phase.drop_objection(this);
```

Call these to prevent and re-allow the current phase ending

```
function void phase_started(uvm_phase phase)
function void phase_ended(uvm_phase phase)
```

Implement these to specify behavior for the start/end of each phase; threads started here are not auto-killed

uvm_component Example

```
task main_phase(uvm_phase phase);  
    phase.raise_objection(this);  
    ... main test behavior, e.g. send 100  
    phase.drop_objection(this);  
endtask
```

Called automatically
when main phase
starts (after all
phase_started() calls)

```
function void phase_started(uvm_phase phase);  
    if (phase.get_name() == "post_reset")  
        fork background_thread(); join_none  
endfunction
```

Called automatically
when phase first starts.
Thread forks when
phase is post_reset.

User-defined Phases

- Integrator can create one or more phases
- Integrator can create schedules with any mix of standard and user-defined phases then assign components to use one of those schedules

“want new phase named cfg2 after configure and before post_configure”

New sched:

```
uvm_domain common =  
    uvm_domain::get_common_domain();  
common.add(cfg2_phase::get(),  
    .after_phase(configure_phase::get()),  
    .before_phase(post_configure_phase.get()))  
);
```

configure

cfg2

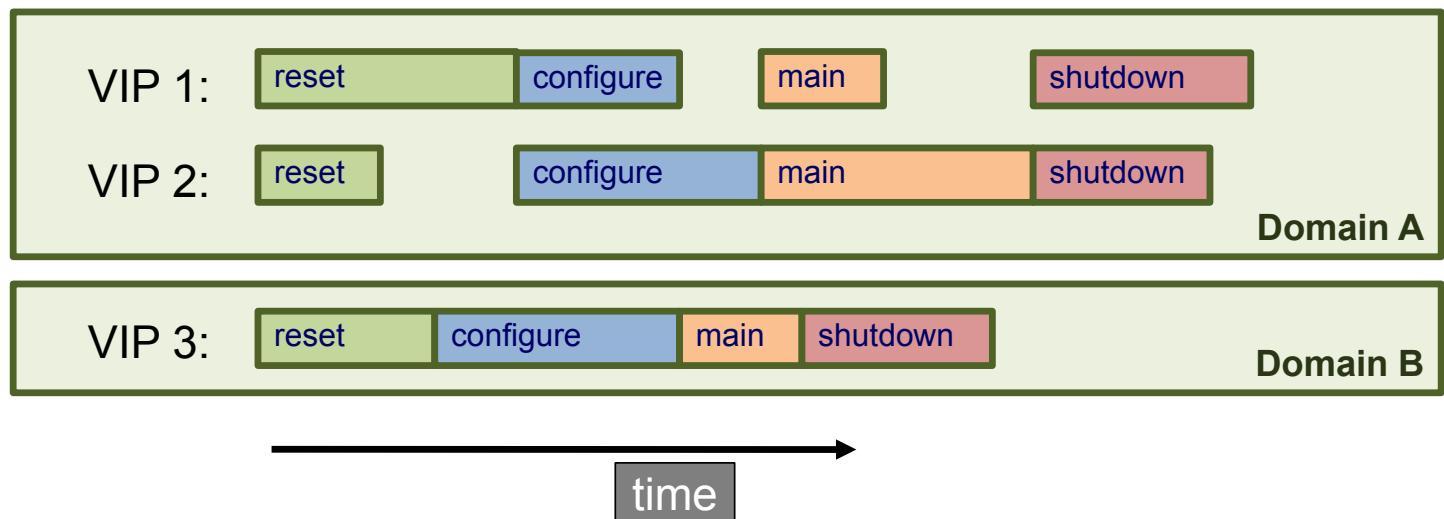
post_configure

Separate Domains

- Sometimes it is OK for a part of the design/environment to do behavior that is out of alignment with the remainder
 - e.g. mid-sim reset of a portion of the chip
 - e.g. one side starts main functionality while other side is finishing configuration

Domains

- Domains are collections of components that must advance phases in unison
 - By default, no inter-domain synchronization



Domain Synchronization

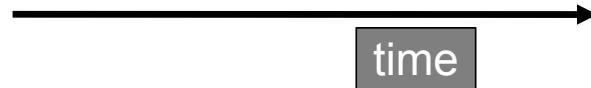
- Domains can be synchronized at one, many, or all phase transitions.

```
domainA.sync(.target(domainB), .phase(uvm_main_phase::get()));
```

Two domains sync'd at main

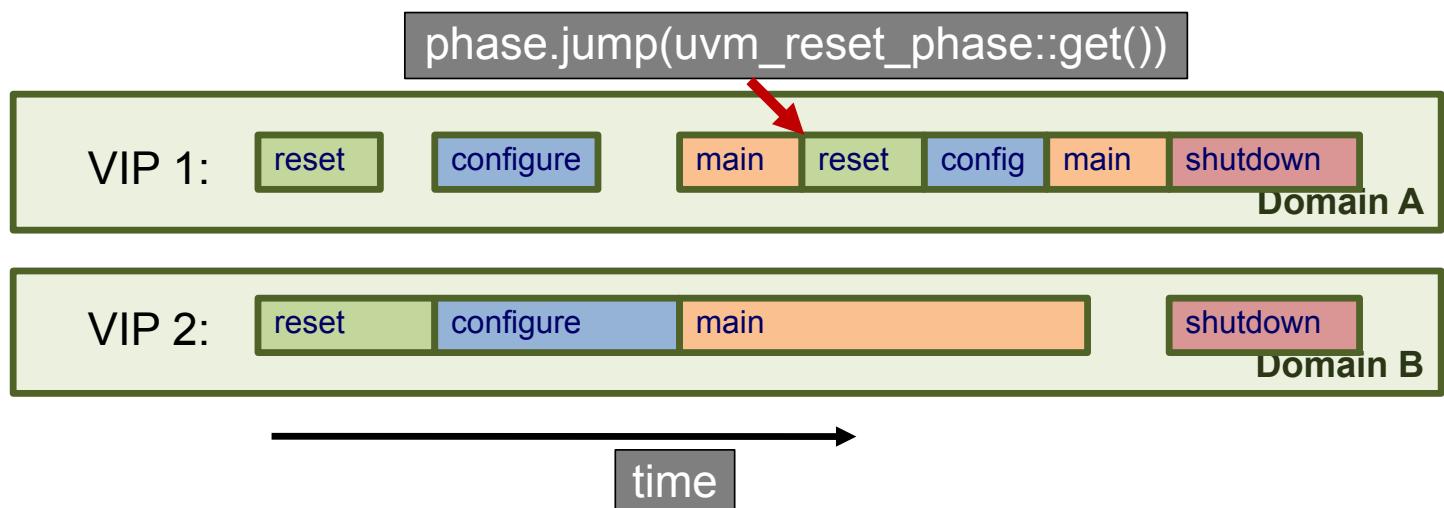
VIP 1: reset configure main shutdown Domain A

VIP 2: reset configure main shutdown Domain B



Fully Synchronized Domains

- If two domains are fully synchronized and one domain jumps back, the second domain will continue in its current phase and wait for the first to catch up



Sequences and Phases

```
class my_seq extends uvm_sequence#(my_req,my_rsp);  
  virtual task body();  
    if (starting_phase != null)  
      starting_phase.raise_objection(this,  
                                     "Starting my_seq");  
    ...//body of sequence  
    if (starting_phase != null)  
      starting_phase.drop_objection(this,  
                                     "Ending my_seq");  
  endtask  
endclass
```

Raise objection first

Drop objection last

Phase won't end until my_seq completes

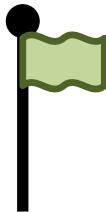
Unless you do a jump()

Ending Phases

```
task run_phase(uvm_phase phase);  
    phase.raise_objection();  
    seq.start();  
    phase.drop_objection();  
endtask
```

Must raise_objection()
before first NBA

Phase ends when all
objections are dropped



VIP:
my_seq

run extract check report

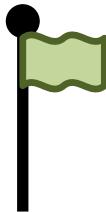


Ending Phases

```
task main_phase(uvm_phase phase);  
    phase.raise_objection();  
    seq.start();  
    phase.drop_objection();  
endtask
```

Must raise_objection()
before first NBA

Phase ends when all
objections are dropped



VIP:

main_seq

main

post_main



Delaying Phase End

```
task main_phase(uvm_phase phase);  
  while(!ending) begin  
    ...  
    if(ending)  
      phase.drop_objection();  
    ...  
  endtask
```

Drop when
really done

```
task main_phase(uvm_phase phase);  
  phase.raise_objection();  
  seq.start();  
  phase.drop_objection();  
endtask
```

all_dropped resets
phase objection

```
virtual function void  
  phase_ready_to_end(uvm_phase phase);  
  if(phase.get_name=="main") begin  
    ending = 1;  
    if(busy)  
      phase.raise_objection();  
  end  
endfunction
```

calls
phase_ready_to_end()

last chance to raise
an objection

VIP: main_seq

post_main



Ending Phases

- Phase objection must be raised before first NBA
- Phase forks off processes
 - wait for `phase.all_dropped` ←
 - call `phase_ready_to_end()`
 - component can raise objection
 - Check objection count
 - call `phase_ended()` —————
 - `kill_processes`
 - `execute successors`

Test Phase Example

```
task reset_phase (uvm_phase phase);
    reset_seq rst = reset_seq::type_id::create("rst");
    phase.raise_objection(this, "resetting");

    rst.start(protocol_sqr); ←
    phase.drop_objection(this, "ending reset");
endtask: reset_phase

task configure_phase (uvm_phase phase);
    configure_seq cfg = configure_seq::type_id::create("cfg");
    phase.raise_objection(this, "configuring dut");
    cfg.start(protocol_sqr);
    phase.drop_objection(this, "dut configured");
endtask: configure_phase

task main_phase (uvm_phase phase);
    test_function1_seq tst = test_function1_seq ::type_id::create("tst");
    phase.raise_objection(this, "functionality test");
    tst.start(protocol_sqr);
    phase.drop_objection(this, "functionality tested");
endtask: main_phase
```

(Test) Component runs different sequences in each phase.
Phase level sequences may run on different sequences in parallel

Workshop Outline

✓10:00am – 10:05am	Dennis Brophy	Welcome
✓10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
✓10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
11:25am – 11:40am	Break	
11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A

Accellera at DAC

- **Accellera Breakfast at DAC: UVM User Experiences**
 - An Accellera event sponsored by Cadence, Mentor, and Synopsys
 - Tuesday, June 7th, 7:00am-8:30am, Room 25AB
- **Accellera IP-XACT Seminar**
 - An introduction to IP-XACT, IEEE 1685, Ecosystem and Examples
 - Tuesday, June 7th, 2:00pm-4:00pm, Room 26AB
- **Birds-Of-A-Feather Meeting**
 - Soft IP Tagging Standardization Kickoff
 - Tuesday, June 7, 7:00 PM-8:30 PM, Room 31AB

Workshop Outline

✓10:00am – 10:05am	Dennis Brophy	Welcome
✓10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
✓10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
✓11:25am – 11:40am	Break	
11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A



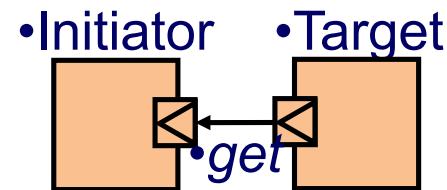
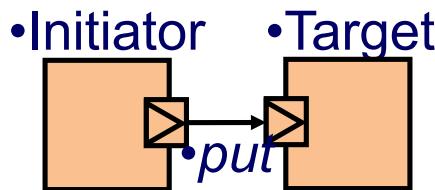
UVM Transaction Level Modeling (TLM2)

Janick Bergeron
Synopsys



TLM-1.0

- Unidirectional put/get interfaces



- Simple message-passing semantics
- No response model
 - E.g. *What did I read back??*
- Never really caught on in SystemC

Why TLM-2.0?

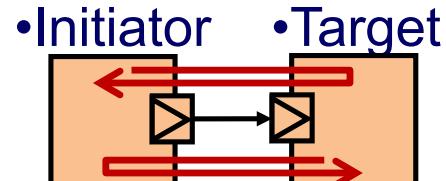
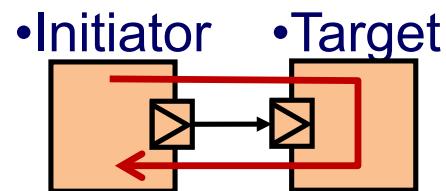
- Better interoperability over TLM-1.0
 - Semantics
 - Pre-defined transaction for buses
- Performance
 - Pass by reference (in SystemC)
 - Temporal decoupling

TLM-2.0 in SV vs SC

- Blocking transport interface ✓
- Nonblocking transport interfaces ✓
- Direct memory interface ✗
- Debug transport interface ✗
- Initiator sockets ✓
- Target sockets ✓
- Generic payload ✓
- Phases ✓
- Convenience sockets ✗
- Payload event queue ✗
- Quantum keeper ✗
- Instance-specific extensions ✗
- Non-ignorable and mandatory extensions ✓
- Temporal decoupling ✓

TLM-2.0 Semantics

- Blocking
 - When the call returns, the transaction is done
 - Response is annotated
- Nonblocking
 - Call back and forth
 - Protocol state changes
 - Transaction is updated
 - Until one says “done”



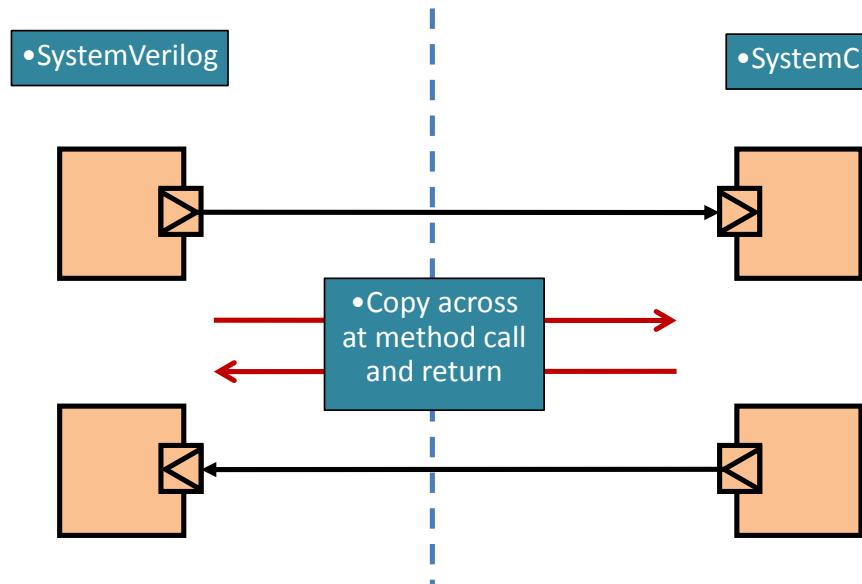
Generic Payload Attributes

- Pre-defined bus transaction

Attribute	Type	Modifiable?
Command	uvm_tlm_command_e	No
Address	bit [63:0]	Interconnect only
Data	byte unsigned []	Yes (read command)
Data length	int unsigned	No
Byte enable pointer	byte unsigned []	No
Byte enable length	int unsigned	No
Streaming width	int unsigned	No
Response status	uvm_tlm_response_status_e	Target only
Extensions	uvm_tlm_extension_base []	Yes

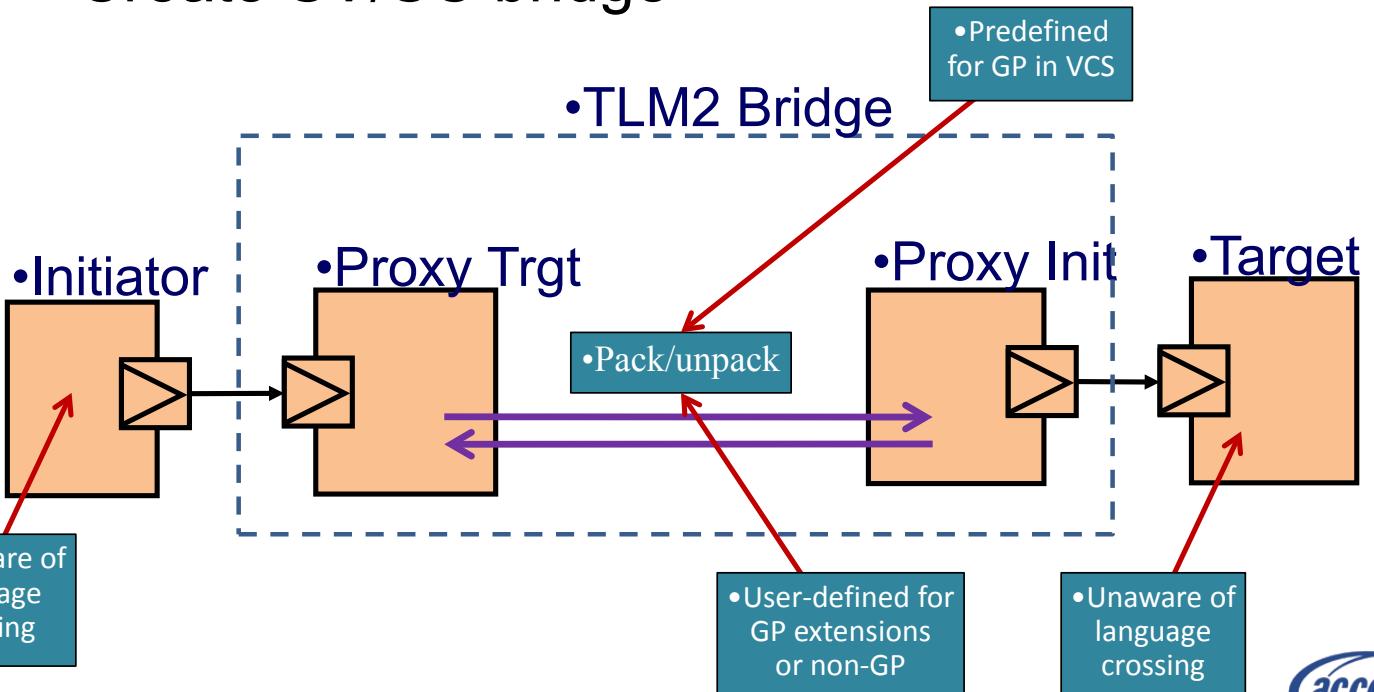
Connecting to SystemC

- Tool-specific mechanism
 - Not part of UVM



TLM-2.0 Bridge Between SV & SC

- Proxy sockets terminate the TLM2 connections in each language
 - Create SV/SC bridge





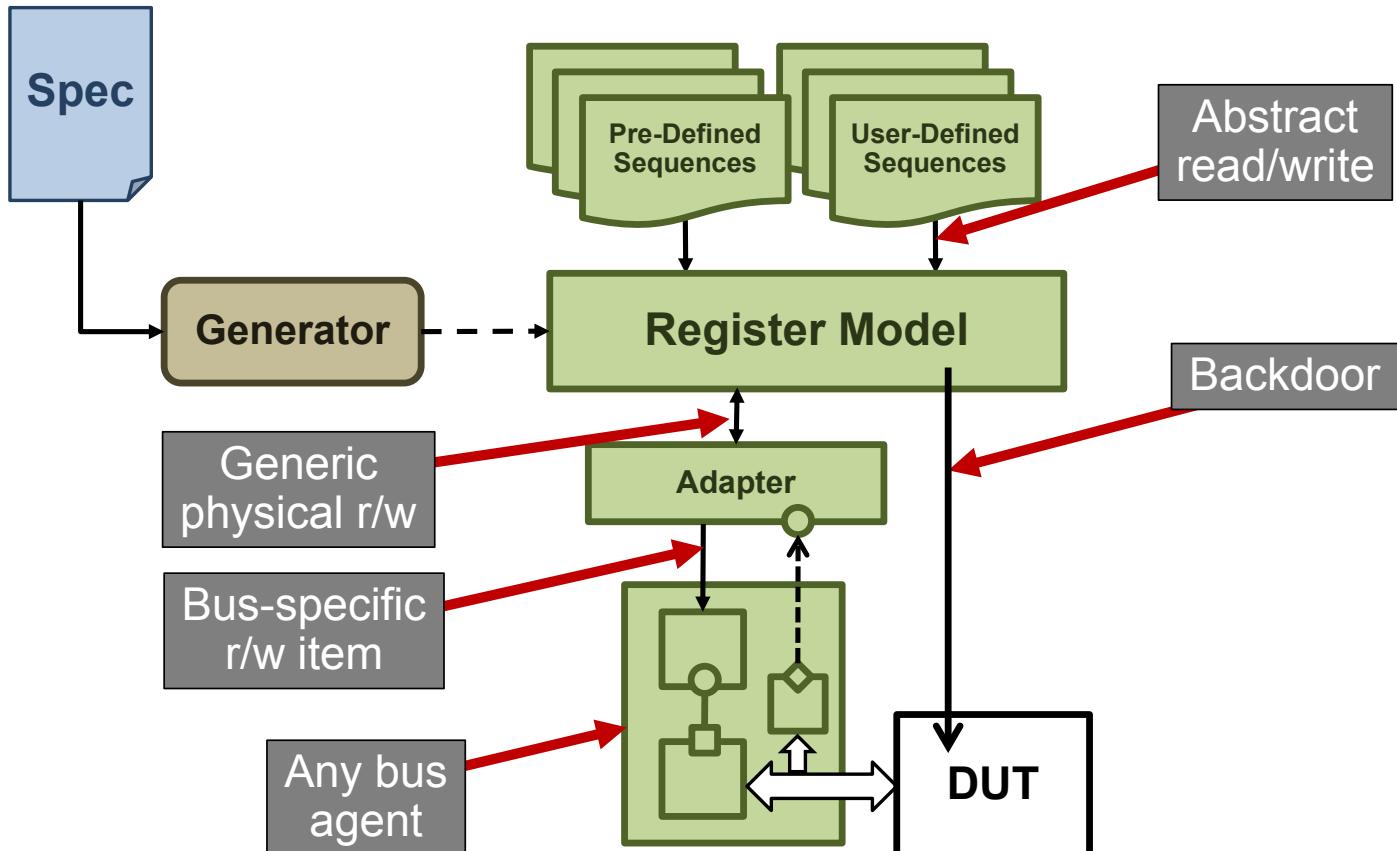
UVM Register Model

Janick Bergeron

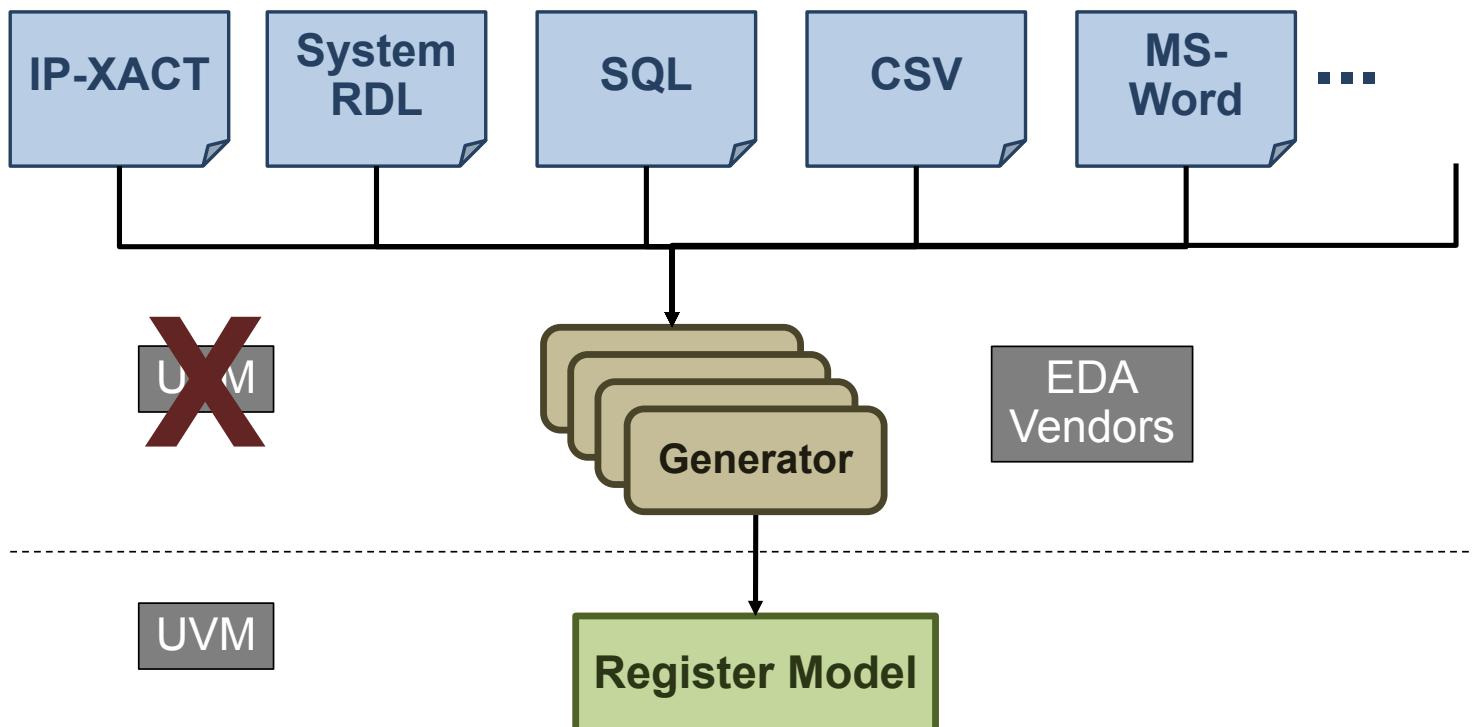
Synopsys



Overall Architecture

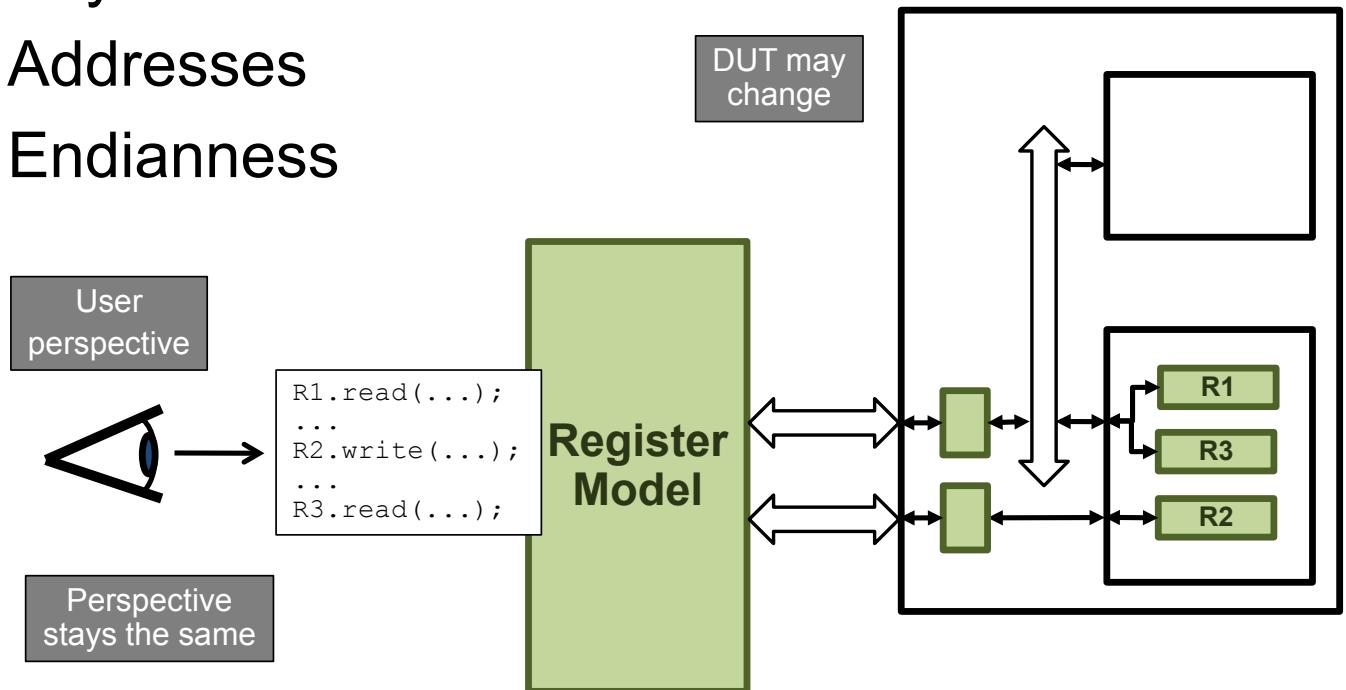


Specification & Generation



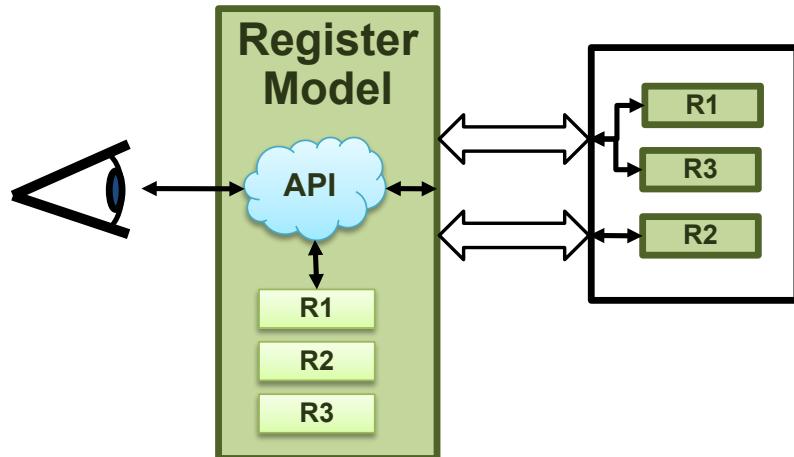
Physical Interfaces

- Register model abstracts
 - Physical interfaces
 - Addresses
 - Endianness



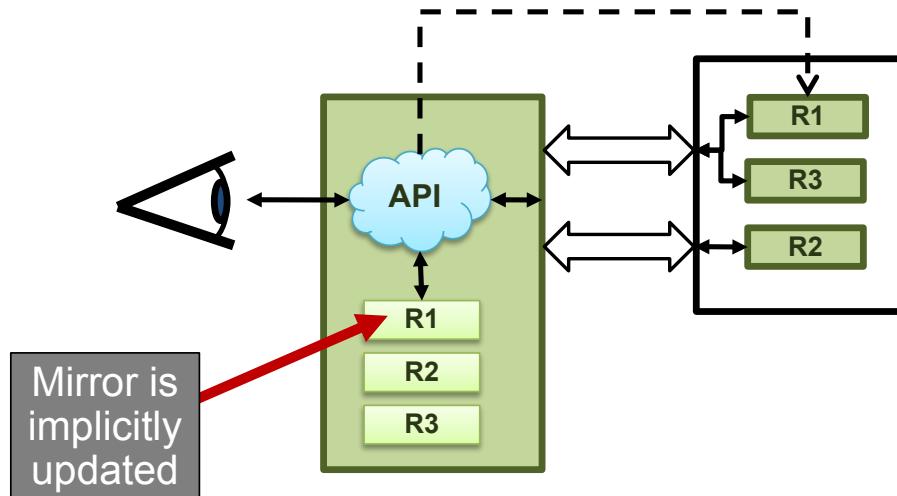
Mirror

- Register model mirrors content of registers in DUT
 - “Scoreboard” for registers
 - Updated on *read* and *write()*
 - Optionally checked on *read()*

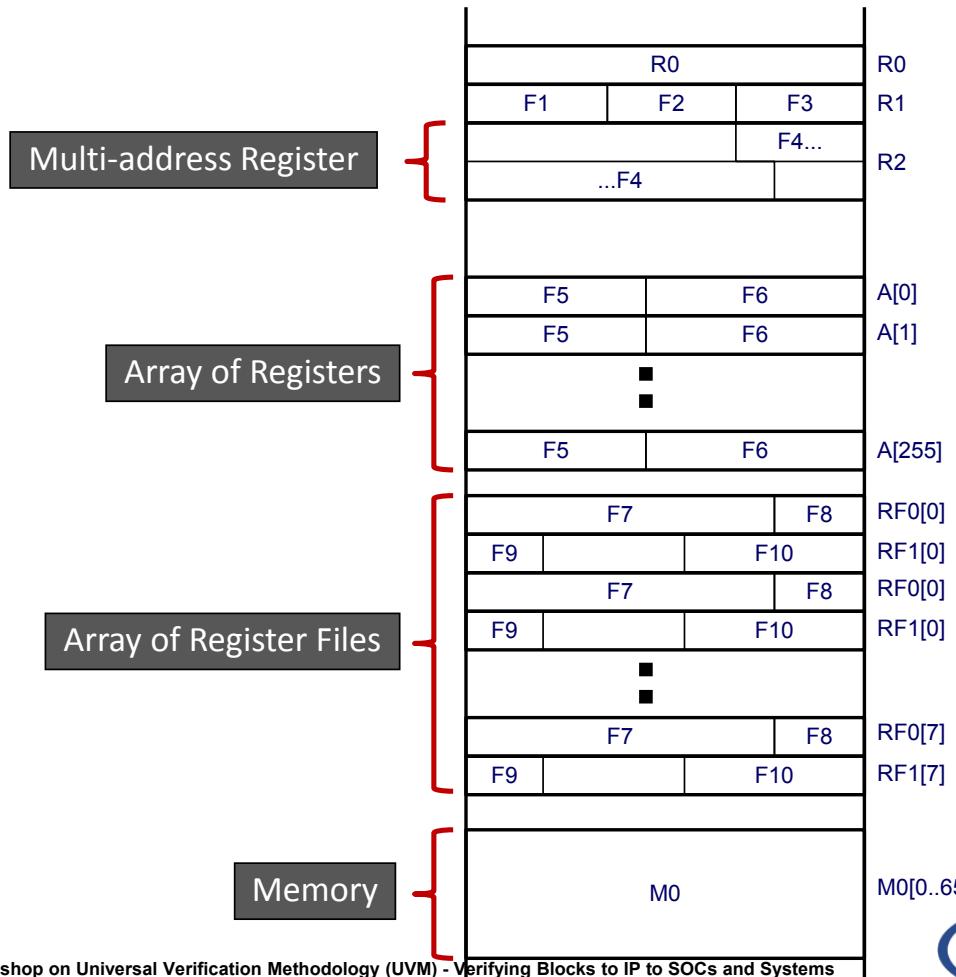


Back-door Access

- Must define HDL path to register in DUT
 - Generator-specific
- Access RTL directly in zero-time via VPI
 - May affect simulator performance



Programmer's View



Class View

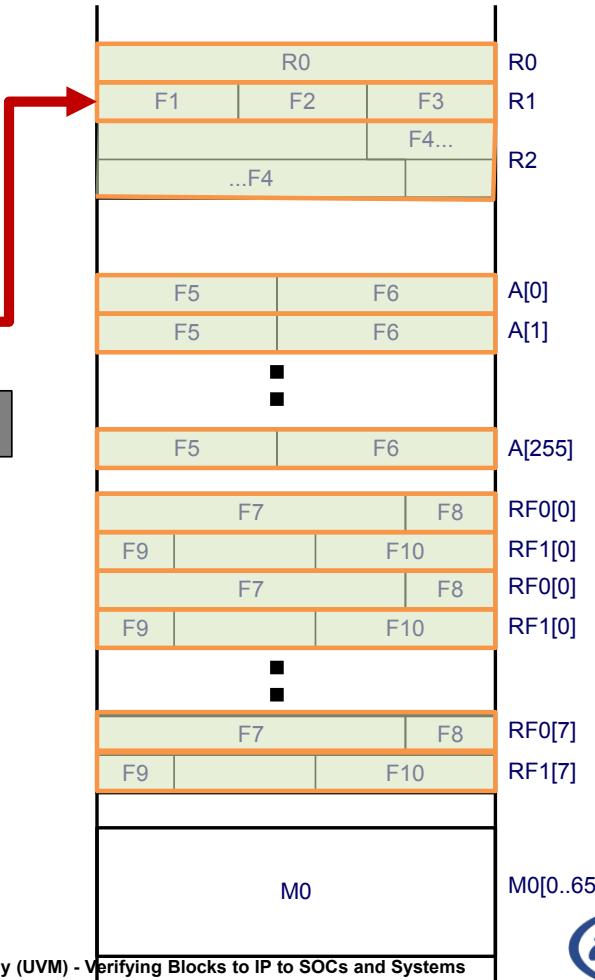
- One class per field
- One class per register

```
class R1_reg extends uvm_reg;  
  uvm_reg_field F1;  
  uvm_reg_field F2;  
  uvm_reg_field F3;  
endclass
```

Generated

Make sure names
are different from
base class methods

- Name
- Address
- Contained fields
- Read/Write methods



Class View

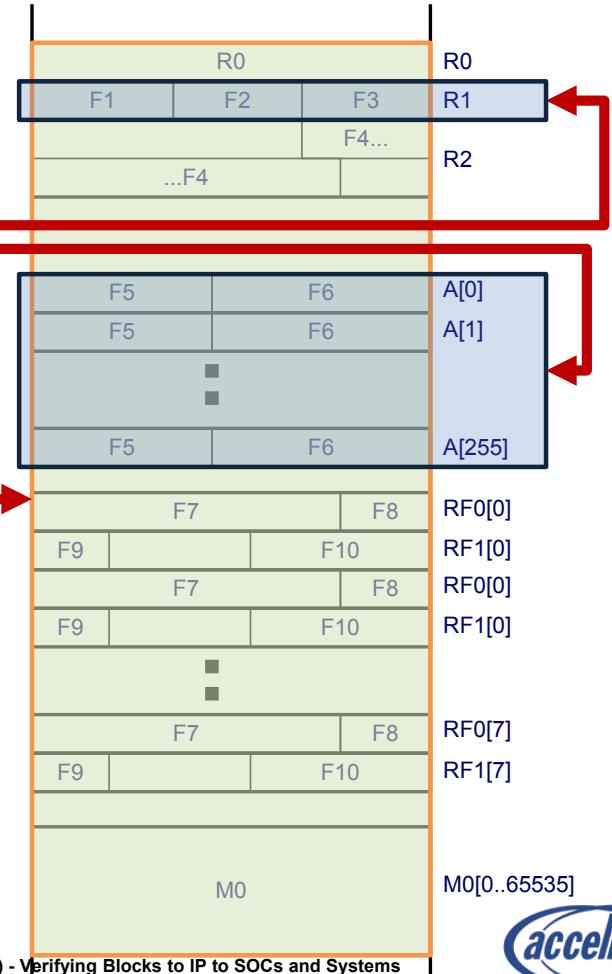
Make sure names
are different from
base class methods

- One class per block

```
class B_blk extends uvm_reg_block;  
  R0_reg   R0;  
  R1_reg   R1;  
  R2_reg   R2;  
  A_reg    A[256];  
  RF_rfile RF[8];  
  ovm_ral_mem M0;  
endclass
```

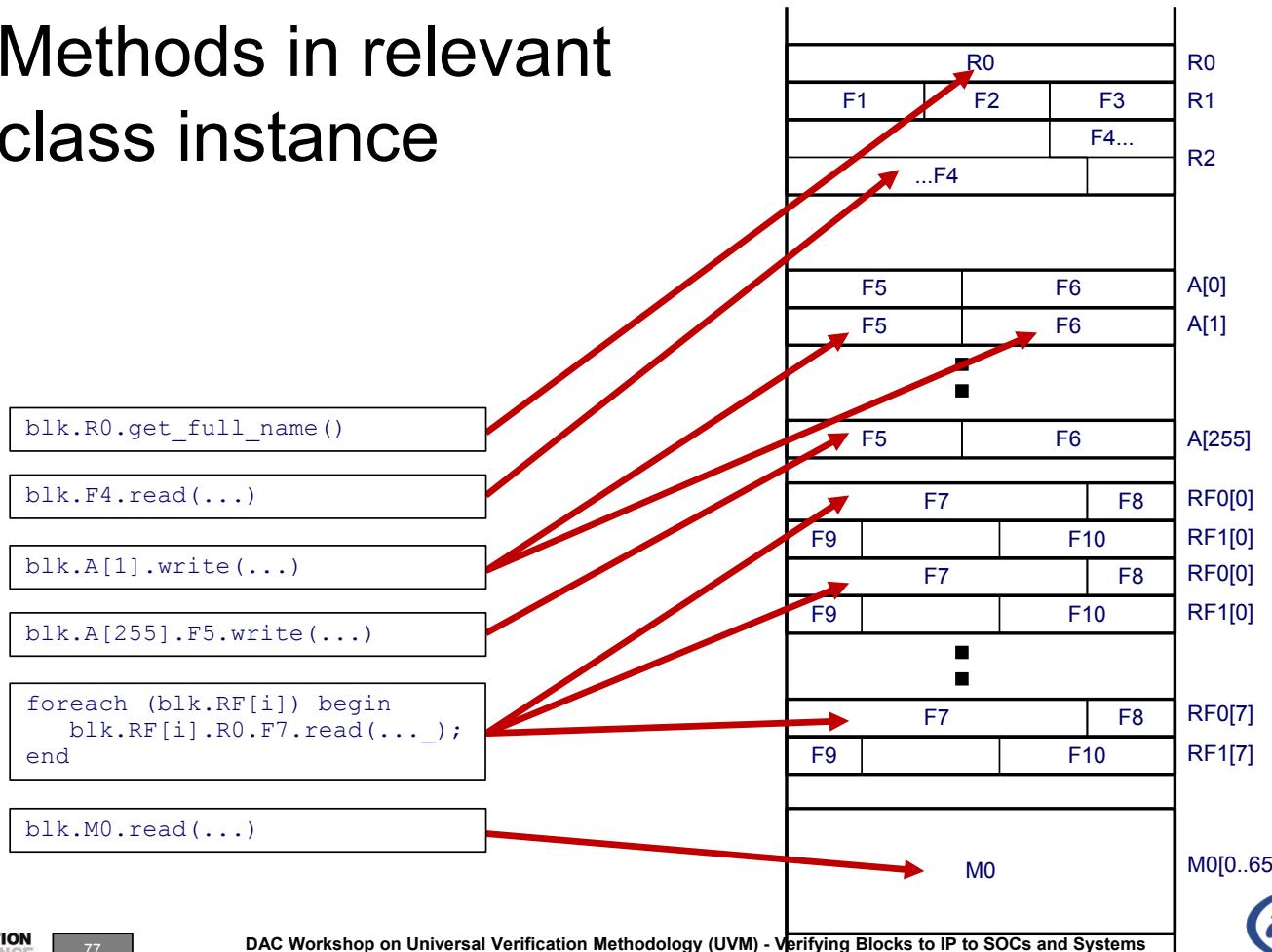
Generated

- Name, Base address
- Contained registers,
register files, memories
 - May be arrays
 - Optional: contained fields



API View

- Methods in relevant class instance



Reading and Writing

- Specify target register by hierarchical reference in register model
 - Compile-time checking

```
blk.blk[2].regfile[4].reg fld
```

By-name API
also available

- Use *read()* and *write()* method on
 - Register
 - Field
 - Memory

```
blk.blk[2].regfile[4].reg fld.read(...);  
blk.mem.write(...);
```

Register Sequences

- Sequences accessing registers should be *virtual sequences*
 - Not associated with a particular sequencer type
- Contain a reference to register model
- Access registers in *body()* task

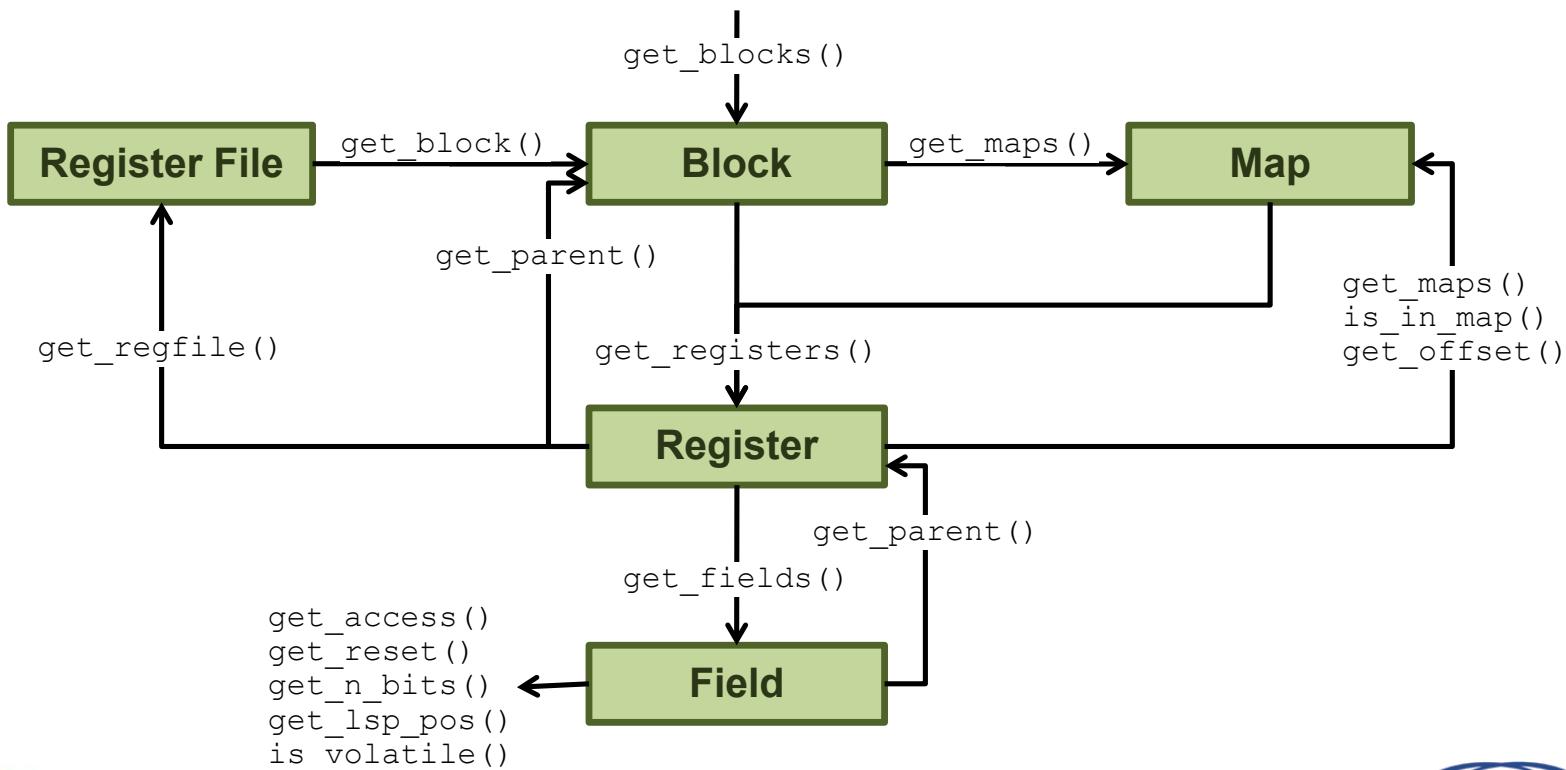
```
class my_test_seq  
extends uvm_sequence;  
  
my_dut_model regmodel;  
  
virtual task body();  
    regmodel.R1.write(...);  
    regmodel.R2.read(...);  
    ...  
endtask  
endclass
```

Register Sequences

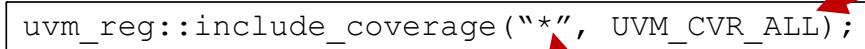
- Includes pre-defined sequences
 - Check reset values
 - Toggle & check every bit
 - Front-door/back-door accesses
 - Memory walking

Introspection

- Rich introspection API



Coverage Models

- Register models *may* contain coverage models
 - Up to the generator
- Not instantiated by default
 - Can be large. Instantiate only when needed.
 - To enable:


uvm_reg::include_coverage ("*", UVM_CVR_ALL);

Annotations:

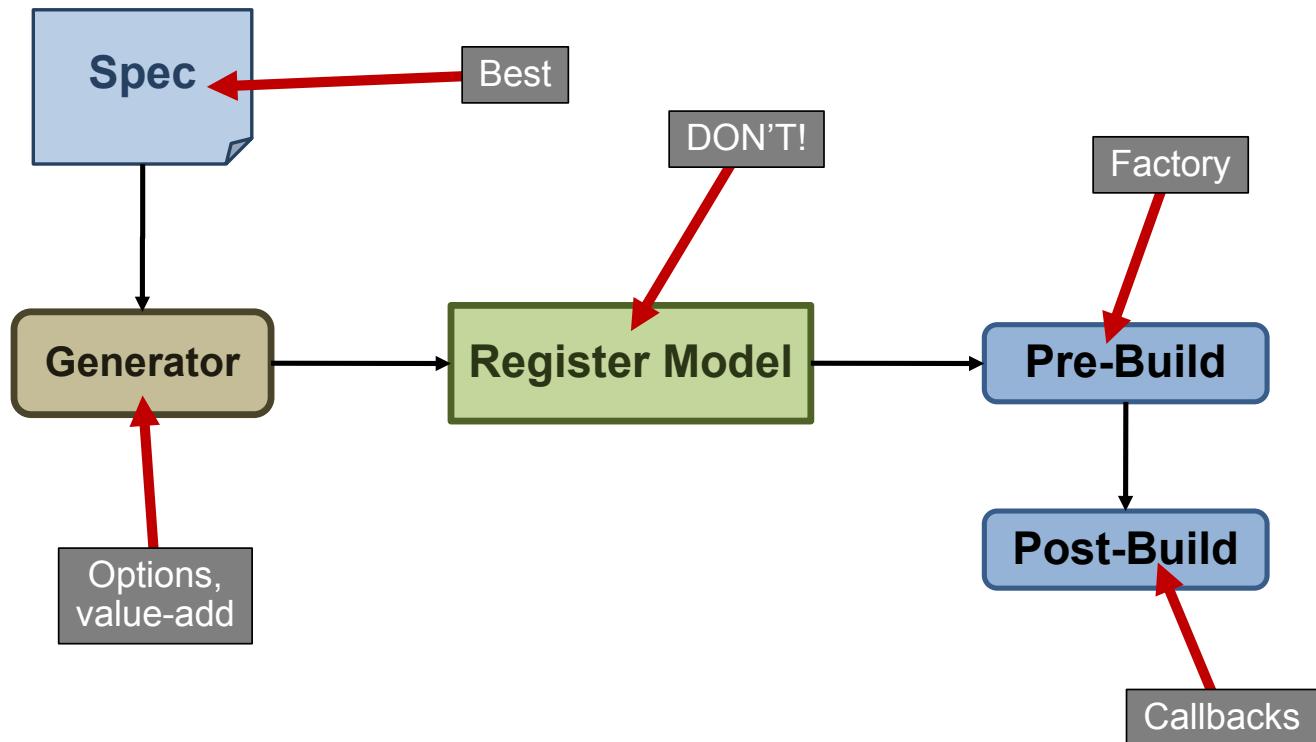
 - All coverage models
 - In all blocks and registersA red arrow points from the code snippet to the "In all blocks and registers" annotation.
- Not collected by default
 - To recursively enable


blk.set_coverage(UVM_CVR_ALL);

Annotation:

 - All coverage models in blockA red arrow points from the code snippet to the "All coverage models in block" annotation.

Customization Opportunities



There's a LOT more!

- DUT integration
- Multiple interfaces
- Randomization
- Vertical Reuse
- “Offline” access
- User-defined front-door accesses
- Access serialization
- Pipelined accesses

Workshop Outline

✓10:00am – 10:05am	Dennis Brophy	Welcome
✓10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
✓10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
✓11:25am – 11:40am	Break	
✓11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches
12:50pm – 1:00pm	All	Q & A



Putting Together UVM Testbenches

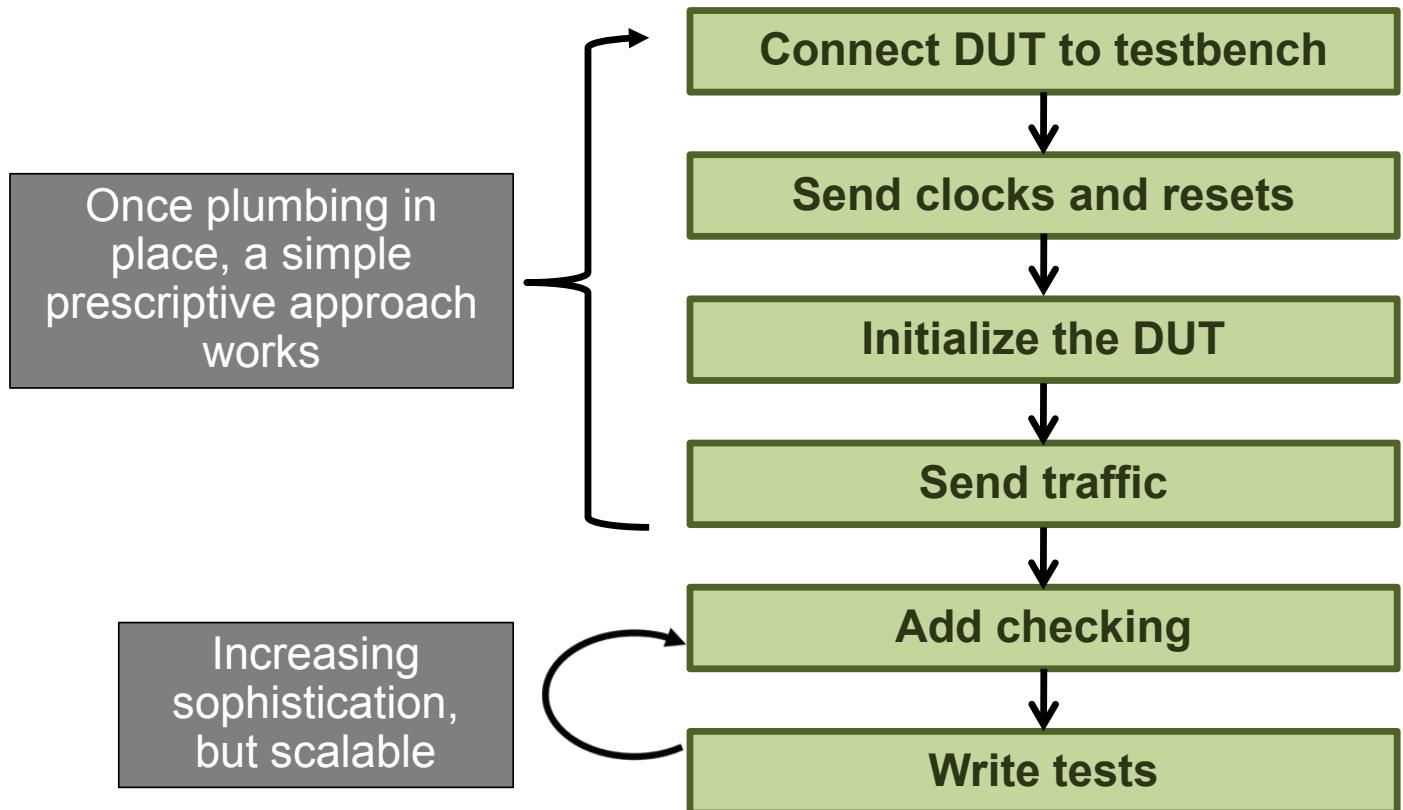
Ambar Sarkar
Paradigm Works, Inc.



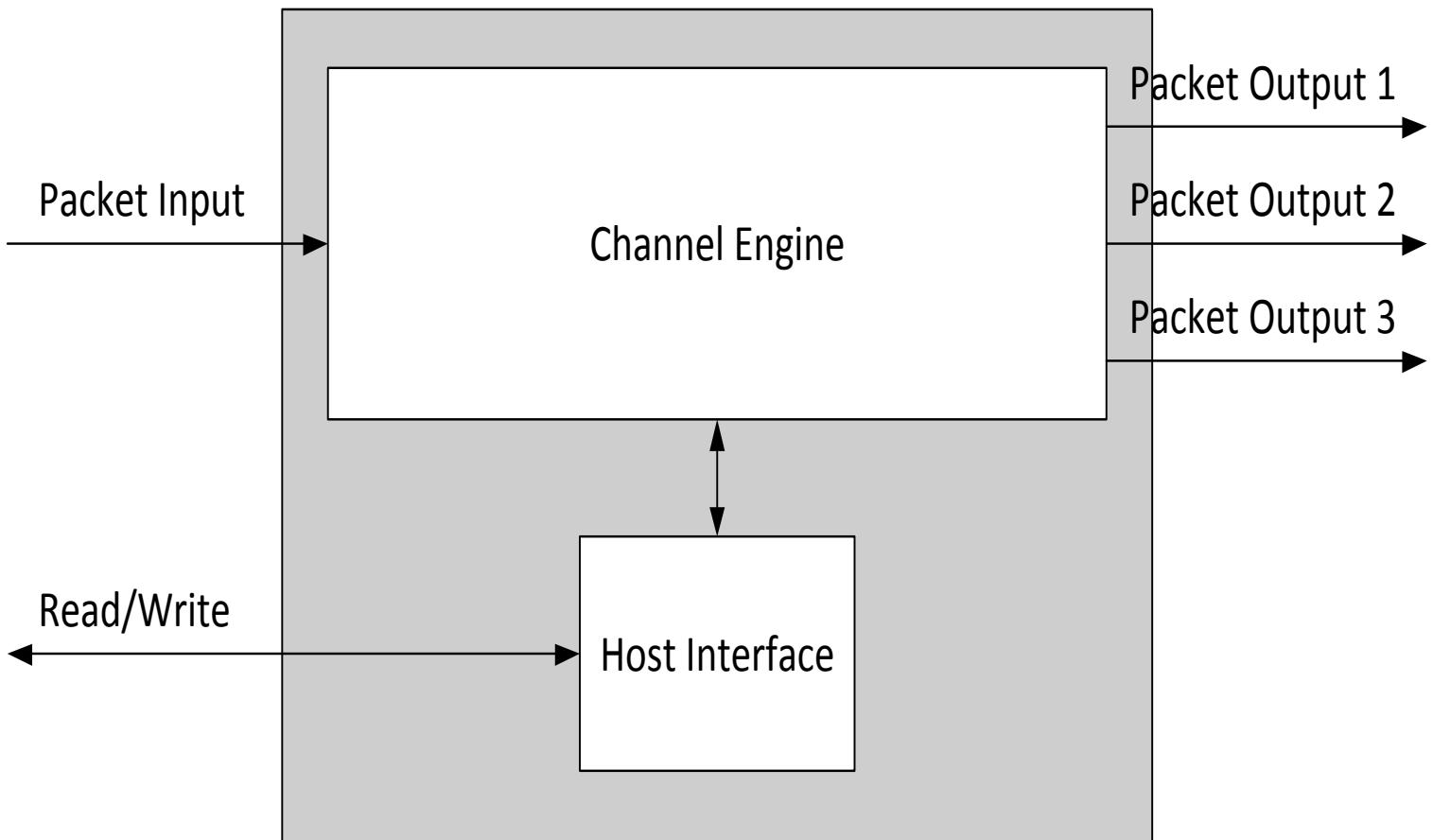
Agenda

- Case studies
 - Several UVM1.0 environments deployed
 - Currently in production
 - Novice to sophisticated teams
- Getting started with UVM is relatively easy
 - Basic tasks remain simple
 - Were able to use a “prescriptive” approach
 - Iteratively developed and scales to any complexity
- Advanced uses
 - Unit to system-level
 - VIP Stacking/Layering

Implementing Basic Steps



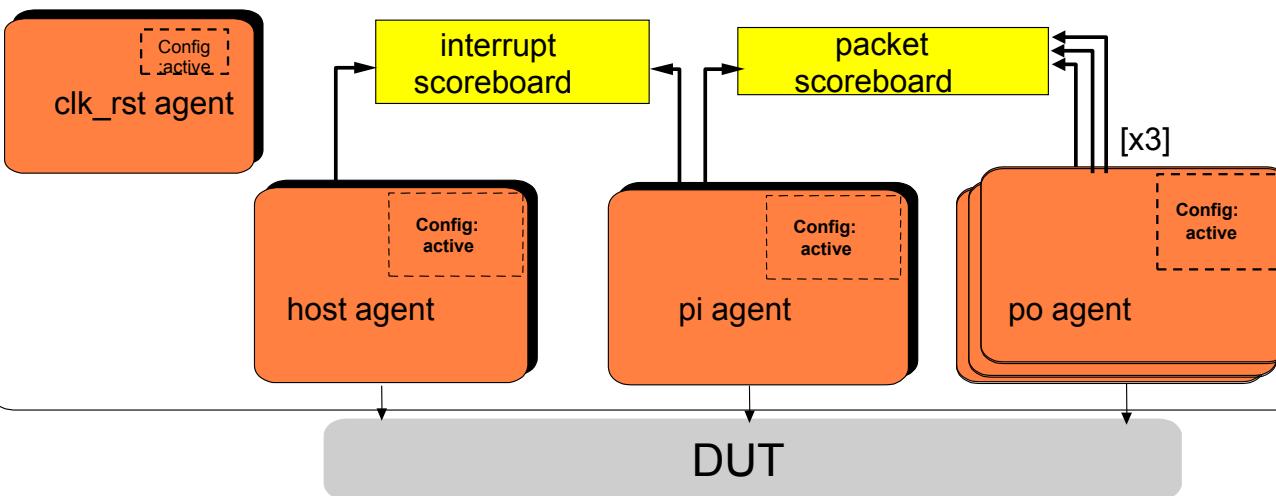
Example



Example Environment in UVM

test cases ...

unit_tb



Connect DUT to Testbench

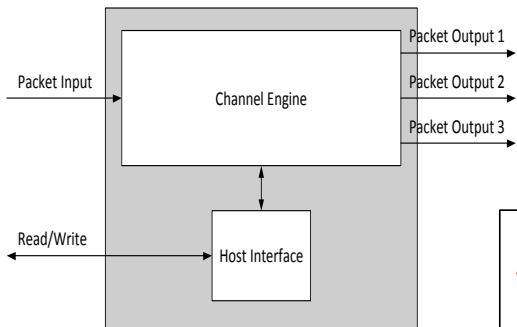
Always use SystemVerilog interface

Use clocking blocks for testbench per interface

Use modports for DUT

Pass interface to the environment

Use uvm_config_db



```
Set (Top level initial block)
uvm_config_db#(virtual host_if)::set(
    null, "my_tb.*", "vif", vif);
```



```
Get (In build phase of the agent)
if(!uvm_config_db#(virtual host_if)::get(
    this,,,"vif",vif))
`uvm_fatal("NOVIF", . . . );
```

Connect Clock and Resets

Combine clock and reset as agents in one env

```
class clk_rst_env extends uvm_env;
    clk_rst_cfg cfg;      //! VIP configuration object
    //! bring ports out for convenience
    uvm_analysis_port #(uvm_transaction) rst_mon_ap_out;

    //! Agents in env
    clk_agent clk_agt;
    rst_agent rst_agt;
    . . .
```

Define reset as sequences

```
task clk_rst_reset_pulse_sequence::body();
    `uvm_do_with(change_sequence, { level == 0;
                                    hold_cycles == init_cycles;
    `uvm_do_with(change_sequence, { level == 1;
                                    hold_cycles == assert_cycles;
    `uvm_do_with(change_sequence, { level == 0;
                                    hold_cycles == negate_cycles;
```

Initializing the DUT

Add transaction definitions for configuration interface (host)

Add driver code for host

Setup host initialization seq

```
class host_transaction extends uvm_sequence_item;  
  ...  
  rand bit[31:0] hi_addr;  
  rand bit[7:0]  hi_wr_data;  
  ...  
  
task pwr_hi_master_driver::drive_transaction(host_transaction trans);  
  if (trans.trans_kind == PWR_HI_WR) begin  
    intf.hif_address = trans.hi_addr;  
    intf.hif_data = trans.hi_wr_data;  
  ...  
  
task my_env_init_sequence::body();  
  regmodel.R1.write(...);  
  regmodel.R2.read(...);  
  ...
```

Sending traffic

Similar to initialization

Sequences differ

```
//! sequence body.  
task pi_sequence::body();  
    pi_transaction#(. . .) req_xn;  
    cfg_inst = p_sequencer.cfg;  
    forever begin  
        p_sequencer.peek_port.peek(req_xn);  
        if (!this.randomize()) begin  
            `uvm_fatal({get_name()}, "RNDFLT"), "Can't randomize");  
        end  
        `uvm_do_with(this_transaction,  
                    { trans_kind == req_xn.trans_kind;  
                      read_vld_delay inside { [0:15] };  
                      . . .  
                    }) ;  
        . . .
```

Checking and Writing Tests

Add checking in the environment

Add monitor code for all components

Connect the scoreboards

```
class my_env extends uvm_env;
    unit_scoreboard#(uvm_transaction, uvm_transaction) sb;

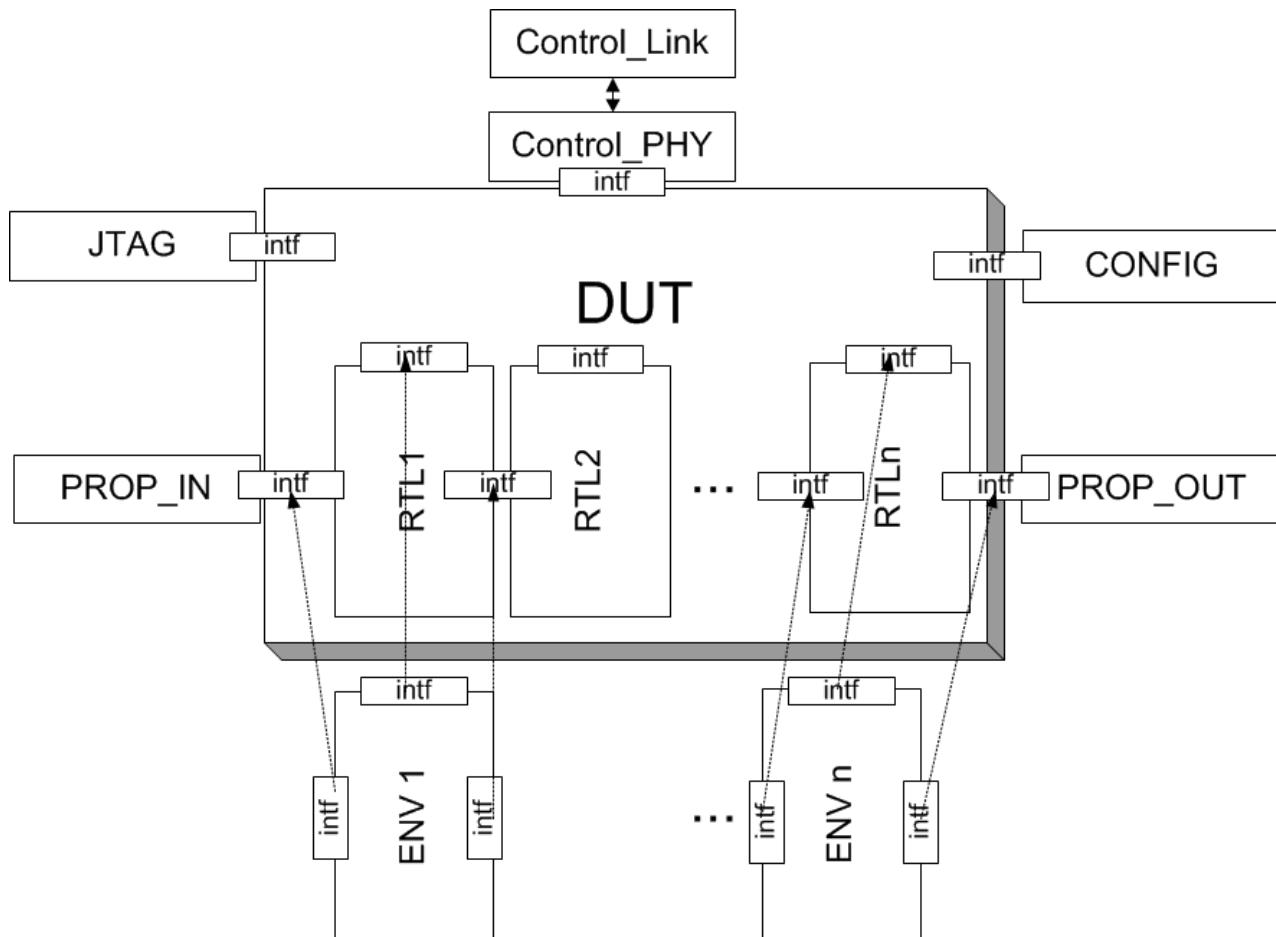
function void my_env::build_phase(uvm_phase phase);
    ...
    sb = unit_scoreboard#(uvm_transaction, uvm_transaction)
        ::type_id::create({get_name(), "_sb"}, this);

function void pwc_env::connect_phase(uvm_phase phase);
    ...
    pi_mon.mon_ap_out.connect(sb.post_export);
    po_mon.mon_ap_out.connect(sb.check_export);

```

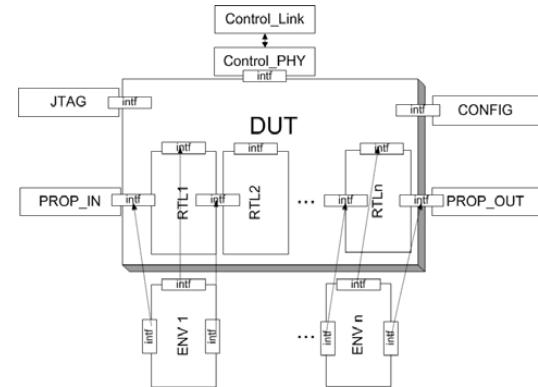
Add test-specific checking in the test as needed

Connecting Unit to System Level



Connecting Unit to System Level: Reuse

- Scoreboarding
 - Reuse SB encapsulated within sub-envs
 - Chain the scoreboards
- Functional Coverage
 - Needed to filter coverage points
- Reuse monitors
 - Avoid duplicated instances
- Gate Simulations
 - Disable monitors
 - Disable internal scoreboards
- Registers
 - Register abstraction reused, but different interfaces used
- Configurations
 - Defined separate system configuration
 - Top level instantiated sub-env configurations
- Sequences
 - Virtual sequencer only at the system level
 - Initialization sequences reused from unit-level
 - Traffic sequences created from scratch at the system level



Connecting Unit to System Level: Prescription

For each sub-env class

Extend sub-env base class

Make all internal components passive

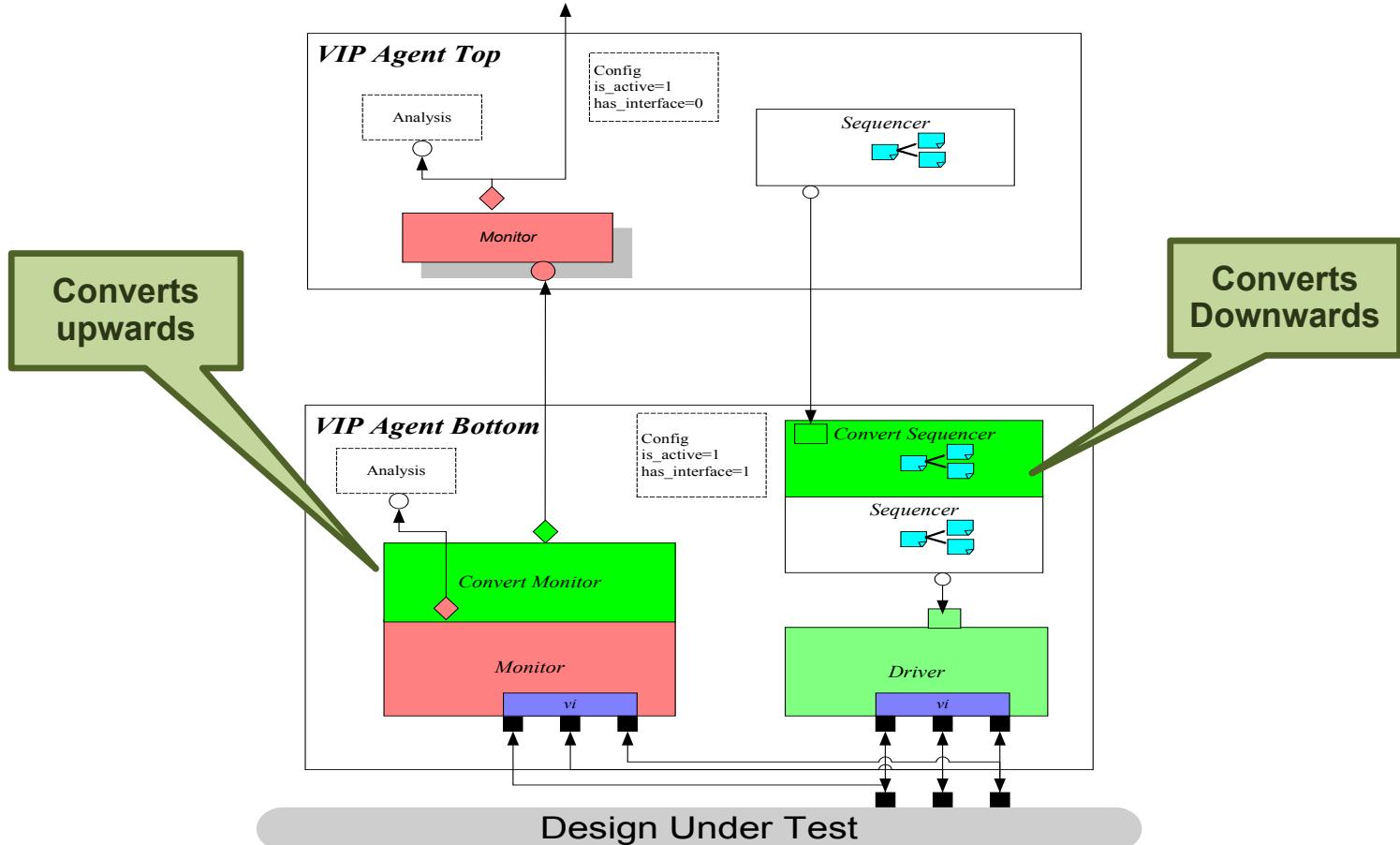
Added sub-env config object to your system config

Declare at system-level:

```
unit_env_cfg          unit_env_cfg_inst;  
`uvm_field_object(unit_env_cfg_inst, UVM_REFERENCE)
```

Turn off virtual sequencer at the unit level

VIP Stacking/Layering



Summary

- Getting started with UVM was relatively easy
 - Once initial plumbing in place
 - Basic tasks remain simple
 - Were able to use a “prescriptive” approach
- Able to iteratively develop testbench
 - Scales to any complexity
 - Unit to system
 - Stacking of VIPs
- Deployed across projects and simulation vendors
 - Worked with minor gotchas
 - No UVM issues found
 - Some SystemVerilog support issues among vendors
 - e.g. Inout ports and modports and clocking blocks

Workshop Outline

✓ 10:00am – 10:05am	Dennis Brophy	Welcome
✓ 10:05am – 10:45am	Sharon Rosenberg	UVM Concepts and Architecture
✓ 10:45am – 11:25am	Tom Fitzpatrick	UVM Sequences and Phasing
✓ 11:25am – 11:40am	Break	
✓ 11:40am – 12:20pm	Janick Bergeron	UVM TLM2 and Register Package
✓ 12:20pm – 12:50pm	Ambar Sarkar	Putting Together UVM Testbenches

12:50pm – 1:00pm

All

Q & A

Questions?



- Download UVM from www.accellera.org
 - Reference Guide
 - User Guide
 - Reference Implementation
 - Discussion Forum

Accellera at DAC

- **Accellera Breakfast at DAC: *UVM User Experiences***
 - An Accellera event sponsored by Cadence, Mentor, and Synopsys
 - Tuesday, June 7th, 7:00am-8:30am, Room 25AB
- **Accellera IP-XACT Seminar**
 - An introduction to IP-XACT, IEEE 1685, Ecosystem and Examples
 - Tuesday, June 7th, 2:00pm-4:00pm, Room 26AB
- **Birds-Of-A-Feather Meeting**
 - Soft IP Tagging Standardization Kickoff
 - Tuesday, June 7, 7:00 PM-8:30 PM, Room 31AB

Lunch in Room 20D

Show your Workshop Badge

for entry

Thank You