

Agenda: Day 2

DAY **2**

5 UVM Configuration & Factory

6 UVM Component Communication

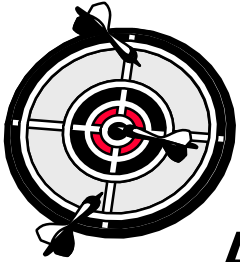


7 UVM Scoreboard & Coverage

8 UVM Callback



Unit Objectives

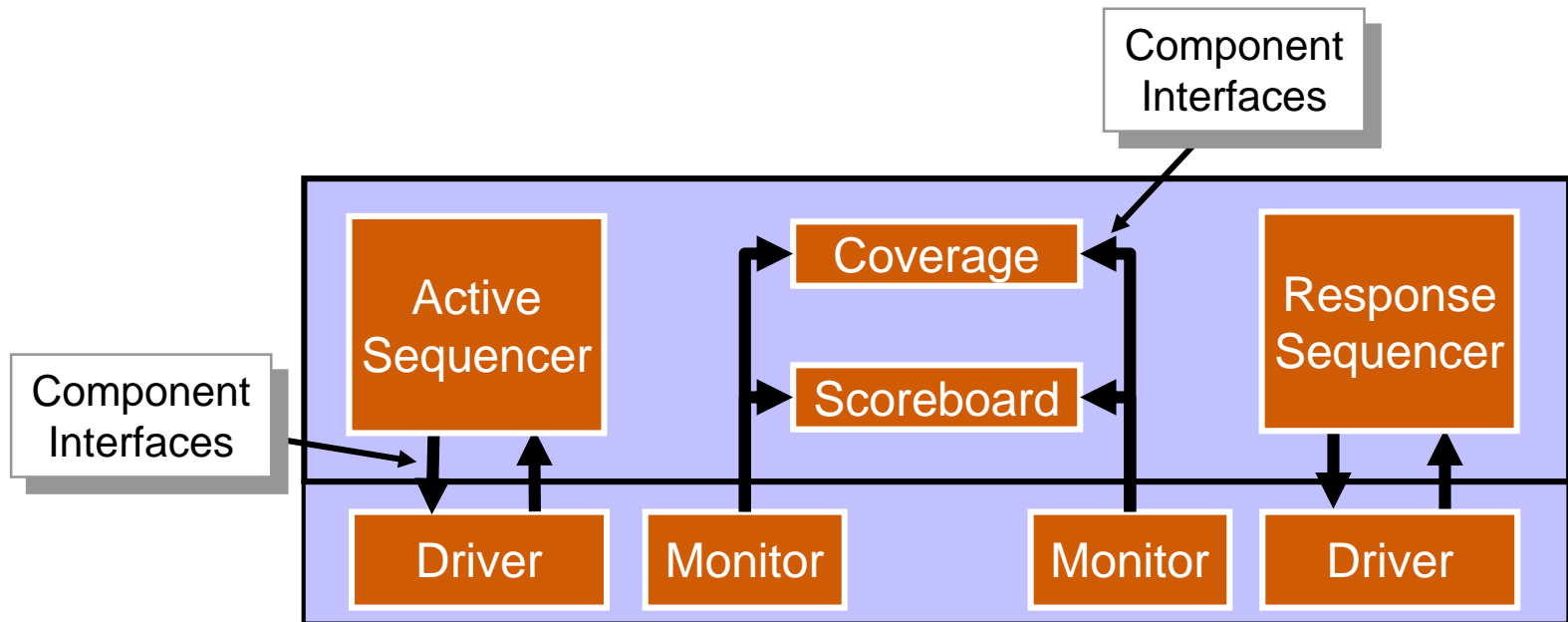


After completing this unit, you should be able to:

- **Describe and implement TLM port/socket for communication between components**

Component Communication: Overview

- Need to exchange transactions between components of verification environment
 - Sequencer → Driver
 - Monitor → Collectors (Scoreboard, Coverage)



Component Communication: Method Based

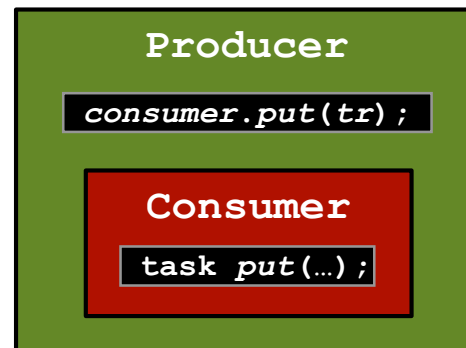
■ Component can embed method for communication

```
class Consumer extends uvm_component;  
  ...  
  virtual task put(transaction tr);  
endclass
```

■ But, the communication method should not be called through the component object's handle

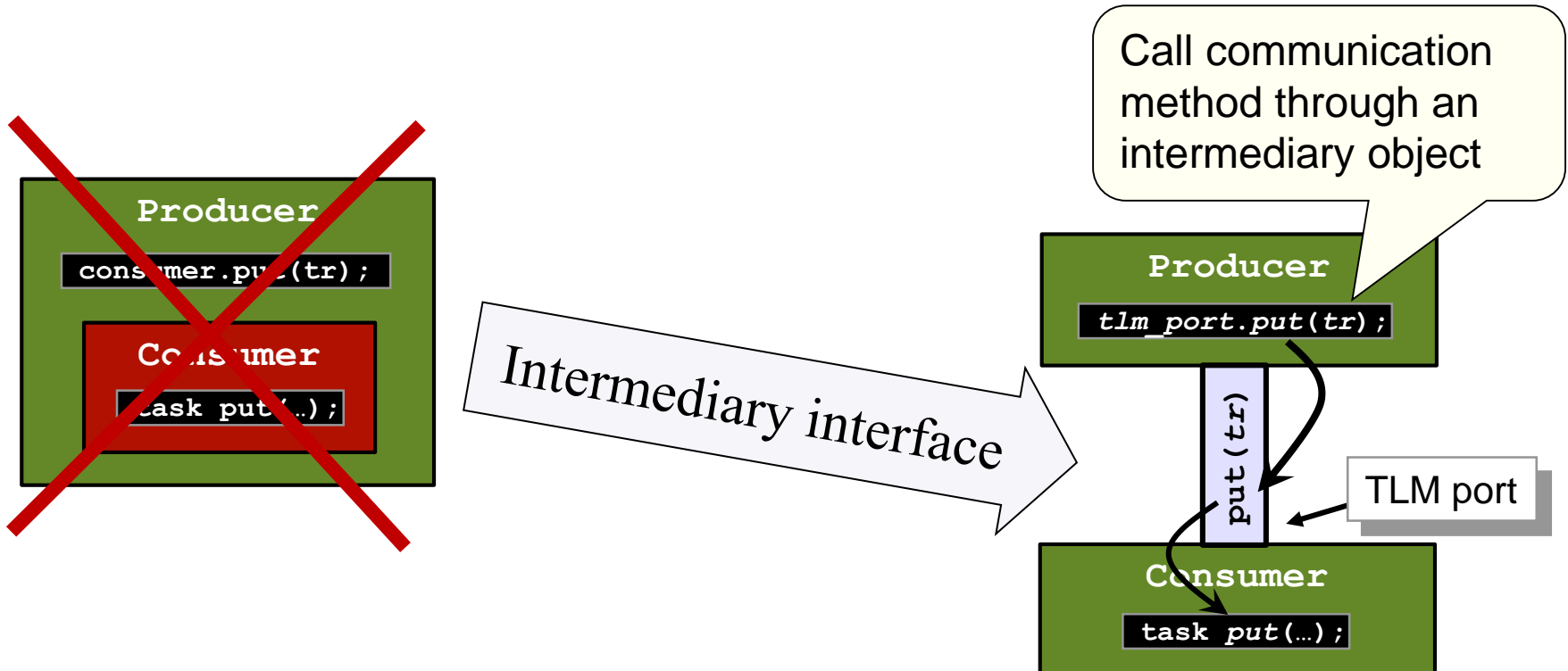
- Code becomes too inflexible for testbench structure
 - ◆ In example below, Producer is stuck with communicating with only a specific Consumer type

Don't Do...



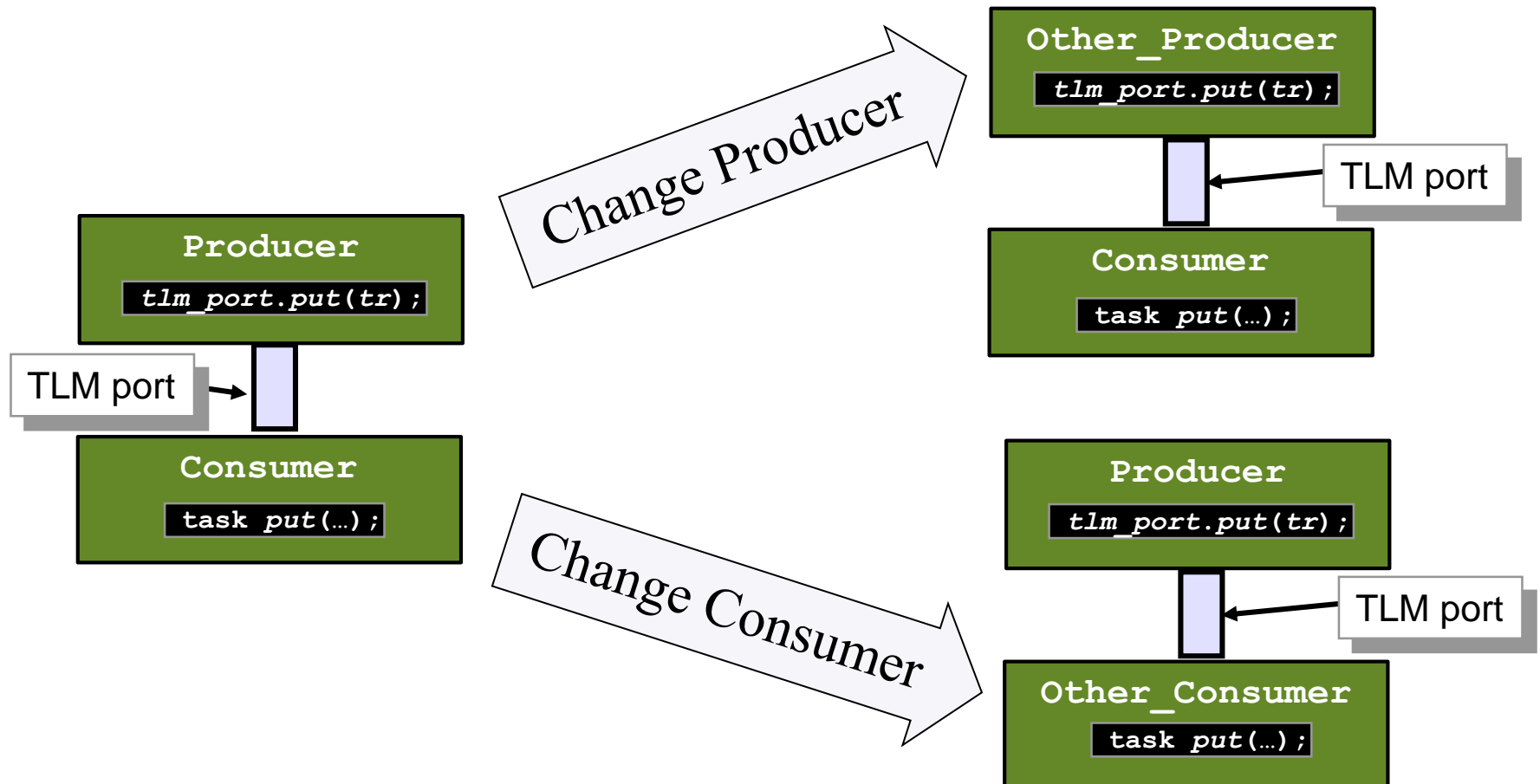
Component Communication: TLM

- Use an intermediary object (TLM) to handle the execution of the procedural communication



Component Communication: TLM

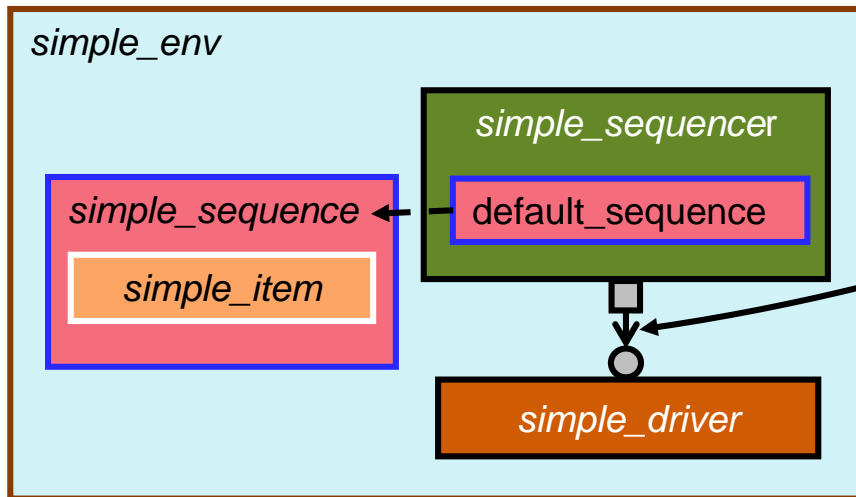
- Through the intermediary object (TLM) components can be re-connected to any other component on a testcase by testcase basis



Communication in UVM: TLM 1.0, 2.0

■ TLM Classes

- `uvm_*_export`
- `uvm_*_imp`
- `uvm_*_port`
- `uvm_*_fifo`
- `uvm_*_socket`



UVM TLM 1.0

square and diamond
ports calls the method

circle ports
implements the method

■ Push

put_producer
calls *port.put()*

port

put_consumer
implements *put()*

■ Pull

get_producer
implements *get()*

port

get_consumer
calls *port.get()*

■ Fifo

producer calls
port.put()

port

fifo

consumer calls
port.get()

One-to-One

■ Analysis (broadcast)

write_producer calls
port.write()

port

write_subscriber
implements *write()*

write_subscriber
implements *write()*

write_subscriber
implements *write()*

One-to-Many

Many-to-One

Push Mode

■ Push mode

put_producer
calls **port.put()**

port

put_consumer
implements **put()**

```
class producer extends uvm_component; ...  
  uvm_blocking_put_port #(packet) put_port;  
  function void build_phase(uvm_phase phase); ...  
    put_port = new("put_port", this);  
  endfunction  
  virtual task initiate_tr(); ...  
    put_port.put(tr);  
endclass
```

```
class consumer extends uvm_component; ...  
  uvm_blocking_put_imp #(packet, consumer) put_export;  
  function void build_phase(uvm_phase phase); ...  
    put_export = new("put_export", this);  
  endfunction  
  virtual task put(packet tr);  
    process_tr(tr);  
endclass
```

```
class environment extends uvm_env; producer p; consumer c; ...  
  virtual function void connect_phase(uvm_phase phase); ...  
    p.put_port.connect(c.put_export); // connection required!  
  endfunction  
endclass
```

Pull Mode

■ Pull mode

get_producer
implements **get()**

port

get_consumer
calls **port.get()**

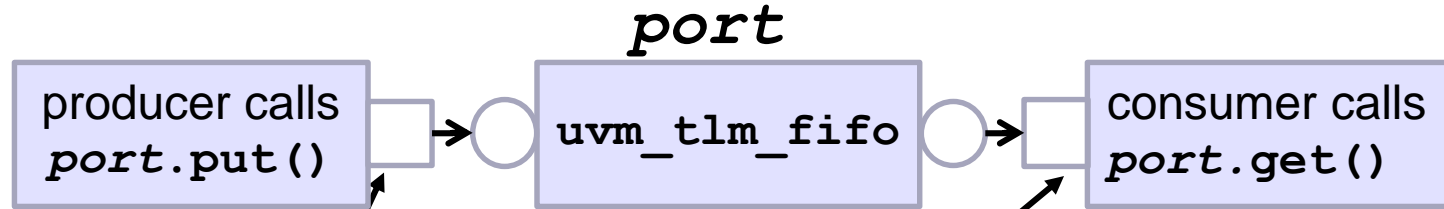
```
class producer extends uvm_component; ...  
  uvm_blocking_get_imp #(packet, producer) get_export;  
  virtual task get(output packet tr);  
    tr = packet::type_id::create("tr", this); ...  
  endtask  
endclass
```

```
class consumer extends uvm_component; ...  
  uvm_blocking_get_port #(packet) get_port;  
  
  virtual task retrieve_tr(); ...  
    get_port.get(tr);  
  endtask  
endclass
```

```
class environment extends uvm_env; ...  
  producer p; consumer c;  
  
  virtual function void connect_phase(uvm_phase phase); ...  
    c.get_port.connect(p.get_export); // connection required!  
  endfunction  
endclass
```

FIFO Mode

■ FIFO Mode



■ Connect producer to consumer via `uvm_tlm_fifo`

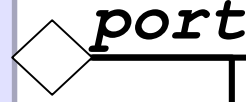
```
class environment extends uvm_env; ...
  producer p;
  consumer c;
  uvm_tlm_fifo #(packet) tr_fifo;
  virtual function void build_phase(uvm_phase phase); ...
    p = producer::type_id::create("p", this);
    c = consumer::type_id::create("c", this);
    tr_fifo = new("tr_fifo", this); // No proxy (type_id) for TLM ports
  endfunction
  virtual function void connect_phase(uvm_phase phase); ...
    p.put_port.connect(tr_fifo.put_export); // connection required!
    c.get_port.connect(tr_fifo.get_export); // connection required!
  endfunction
endclass
```

Analysis Port

■ Analysis (broadcast)

- Analysis port can be left unconnected

write_producer calls
port.write()



write_subscriber
implements **write()**

write_subscriber
implements **write()**

```
class producer extends uvm_component; ...  
  uvm_analysis_port #(packet) analysis_port;  
  virtual task assemble_tr(); ...  
    analysis_port.write(tr);  
endtask  
endclass
```

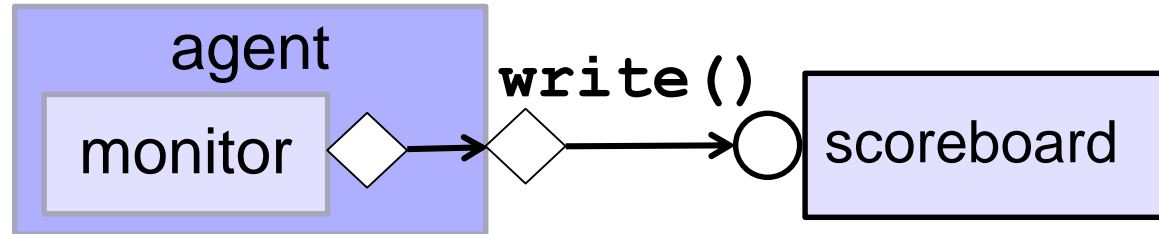
```
class subscriber extends uvm_component; ...  
  uvm_analysis_imp #(packet, subscriber) analysis_export;  
  virtual function void write(packet tr); // cannot block  
    process_transaction(tr);  
endfunction
```

```
class environment extends uvm_env; ...  
  producer p; subscriber s0, s1; // other subscribers  
  virtual function void connect_phase(uvm_phase phase); ...  
    p.analysis_port.connect(s0.analysis_export);  
    p.analysis_port.connect(s1.analysis_export);  
  endfunction  
endclass
```

Port Pass-Through

■ Connecting sub-component TLM ports

- Use same port type

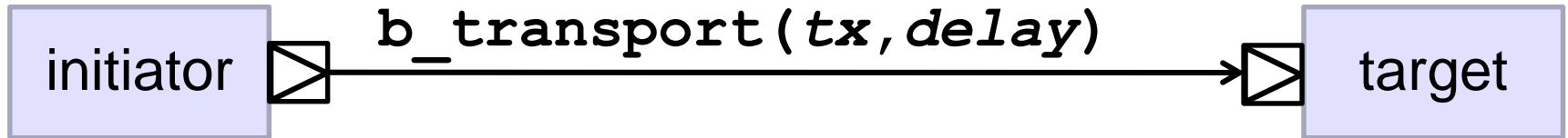


```
class monitor extends uvm_monitor; // other code not shown ...
    uvm_analysis_port #(packet) analysis_port;
    virtual function void build_phase(uvm_phase phase); ...
        this.analysis_port = new("analysis_port", this);
    endfunction
endclass
```

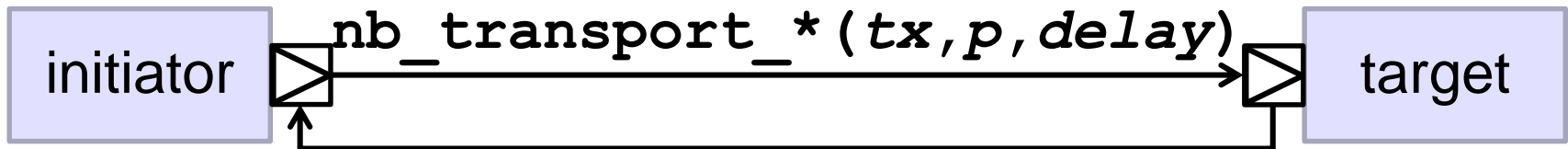
```
class agent extends uvm_agent; // other code not shown ...
    monitor mon;
    uvm_analysis_port #(packet) analysis_port;
    virtual function void build_phase(uvm_phase phase); ...
        this.analysis_port = new("analysis_port", this);
    endfunction
    virtual function void connect_phase(uvm_phase phase); ...
        mon.analysis_port.connect(this.analysis_port);
    endfunction
endclass
```

port
must be
same
type

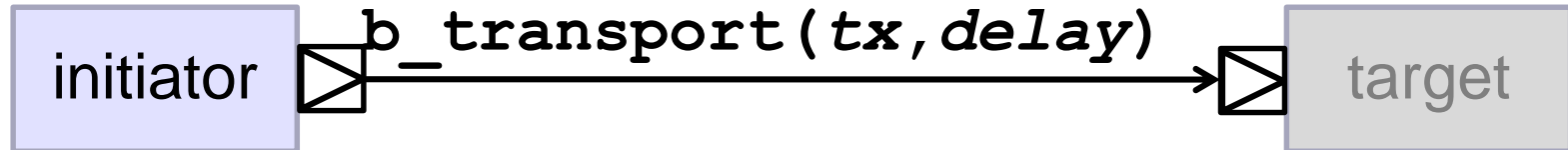
■ Blocking



■ Non-Blocking



Blocking Transport Initiator



```
class initiator extends uvm_component;
  uvm_tlm_b_initiator_socket #(packet) i_socket;
  // constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    i_socket=new("i_socket", this);
  endfunction
  virtual task initiate_tr();
    packet tx = packet::type_id::create("tx", this);
    uvm_tlm_time delay = new();
    delay.set_abstime(1.5, 1e-9); // set delay to 1.5ns
    i_socket.b_transport(tx, delay);
  endtask
endclass
```

Blocking Transport Target

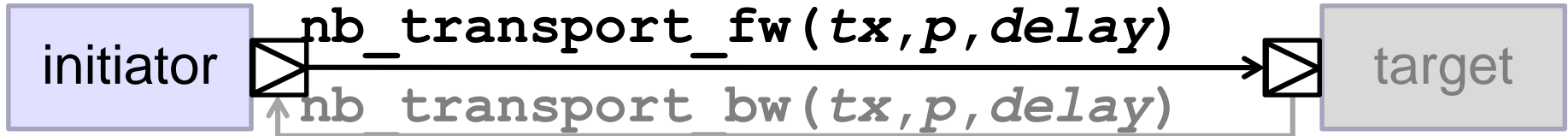


```
class target extends uvm_component; ...
  uvm_tlm_b_target_socket #(target, packet) t_socket;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t_socket=new("t_socket", this);
  endfunction

  virtual task b_transport(packet tx, uvm_tlm_time delay);
    $display("realtime = %t", delay.get_realtime(1ns));
    ...
  endtask
endclass
```

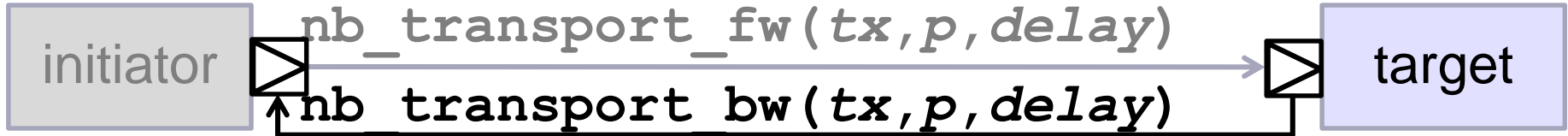
```
class environment extends uvm_env;
  initiator intr;
  target trgt;
  // component_utils, constructor and build_phase not shown
  virtual function void connect_phase(uvm_phase phase);
    intr.i_socket.connect(trgt.t_socket);
  endfunction
endclass
```


Non-Blocking Transport Initiator



```
class initiator extends uvm_component;
  uvm_tlm_nb_initiator_socket #(initiator, packet) i_socket;
  // component_utils, constructor and build_phase not shown
  virtual task initiate_tr();
    uvm_tlm_sync_e sync; uvm_tlm_phase_e p; uvm_tlm_time delay = new;
    packet tx = packet::type_id::create("tx", this);
    if (!tx.randomize()) begin
      `uvm_fatal("RAND_ERR", "tx randomize failed");
    end
    sync = i_socket.nb_transport_fw(tx, p, delay);
  endtask
  virtual function uvm_tlm_sync_e nb_transport_bw(packet tx,
    ref uvm_tlm_phase_e p, input uvm_tlm_time delay);
    // ... Process acknowledgement from target
    return (UVM_TLM_COMPLETED);
  endfunction
endclass
```

Non-Blocking Transport Target



```
class target extends uvm_component;
    uvm_tlm_nb_target_socket #(target, packet) t_socket;
    // component_utils, constructor and build_phase not shown
    virtual function uvm_tlm_sync_e nb_transport_fw(packet tx,
        ref uvm_tlm_phase_e p, input uvm_tlm_time delay);
        tx.print();
        fork process_tr(tx); join_none // for delayed acknowledgement
        return (UVM_TLM_ACCEPTED);
    endfunction
    virtual task process_tr(packet tx);
        uvm_tlm_sync_e sync; uvm_tlm_phase_e p; uvm_tlm_time delay = new;
        // ... After completion of tx processing
        sync = t_socket.nb_transport_bw(tx, p, delay);
    endtask
endclass
```

Unit Objectives Review

Having completed this unit, you should be able to:

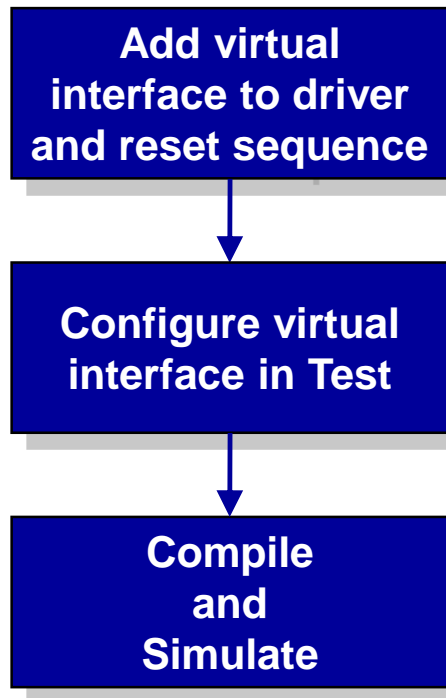
- **Describe and implement TLM port/socket for communication between components**

Lab 3 Introduction



60 min

Implement & configure physical device drivers



Appendix

TLM 2.0 Generic Payload

DVE UVM-Aware Debugging Features

Verdi UVM-Aware Debugging Features

Verdi UVM-Aware Debugging Features

Verdi UVM Debug Switches

For VCS 2016.06 onwards

■ Compile-time switches:

- Requires `-debug_access+all`

■ Post Simulation run-time switches:

```
simv +UVM_VERDI_TRACE +UVM_TR_RECORD +UVM_LOG_RECORD \  
    +UVM_TESTNAME=test_base
```

```
verdi -ssf novas.fsdb -nologo &
```

■ Interactive Simulation run-time switches

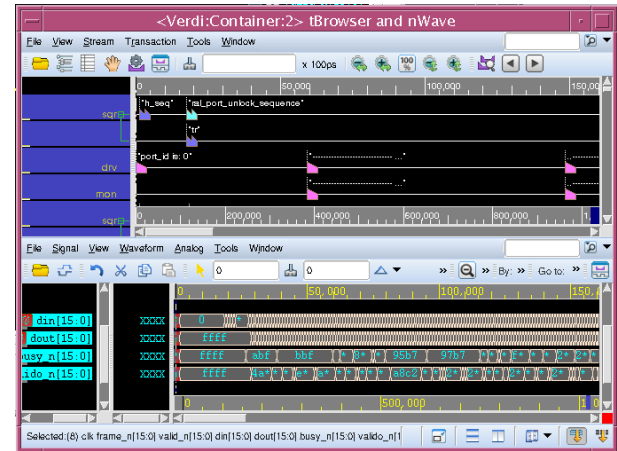
```
simv +UVM_VERDI_TRACE +UVM_TR_RECORD +UVM_LOG_RECORD \  
    +UVM_TESTNAME=test_base -gui=verdi
```

UVM-Aware Features in Verdi

■ Post simulation debug

■ Interactive debug

Transaction Recording



Object Hierarchy Browser

The screenshot shows the Object Hierarchy Browser window in Verdi. It displays a tree view of the simulation objects, including top_levels, env, h_agt, drv, mon, sq, hreg_pre..., i_agt, o_agt, r_agt, sb, and v_reset_sq. The table below shows the details of the objects.

Hierarchy	Class Type	Object Id	Created
top_levels	uvm_compo...		
[0]	test_base	@1	0
env	router_env	@1	0
h_agt	host_agent	@1	0
drv	host_driver	@1	0
mon	host_monitor	@1	0
sq	uvm_seque...	@1	0
hreg_pre...	uvm_reg_pr...	@1	0
i_agt	input_agent[]		
o_agt	output_agent[]		
r_agt	reset_agent	@1	0
sb	ms_scorebo...	@1	0
v_reset_sq	virtual_reset...	@1	0

UVM Window

The screenshot shows the UVM Window in Verdi. It displays a list of UVM phases and their associated actions. The table below shows the details of the phases.

Phase	Begin Time	End Time	Action	Total	Count	Object
build	18500	18500	RAISE	1	1	uvm_test...
connect	18500	18500	RAISE	2	1	uvm_test...
end_of_elaboration	18500	18500	RAISE	3	1	uvm_test...
start_of_simulation	18500	18500	RAISE	4	1	uvm_test...
run	18500	18500	RAISE	5	1	uvm_test...
extract	18500	18500	RAISE	6	1	uvm_test...
check	18500	18500	RAISE	7	1	uvm_test...

UVM Transaction and Log Debug

- Available for both post and interactive simulation

The screenshot displays the Verdi simulation tool interface with several windows open. The top window is the 'Transaction view' showing a hierarchy of components and a timeline of transactions. The bottom window is the 'Waveform window' showing a detailed view of the transaction data, including signal values and timing. A call stack window is also visible on the right side of the transaction view.

Transaction view

Drag and drop component into transaction window to see transaction recorded in that component

Highlight sequence item to see content

Add signal to waveform window

Attribute	Value
\$label	"req"
\$beginTime	18500
\$endTime	35500
\$type	"Transaction"
label	"Transactions"
sa	4'sh1

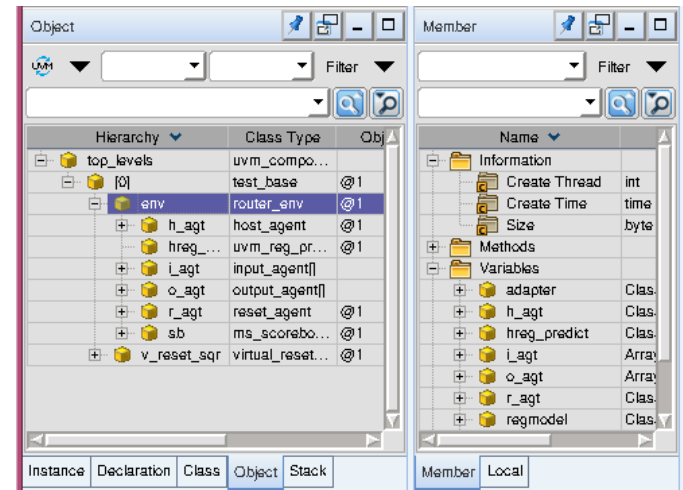
Object Hierarchy Browser

■ General mode and methodology-aware mode

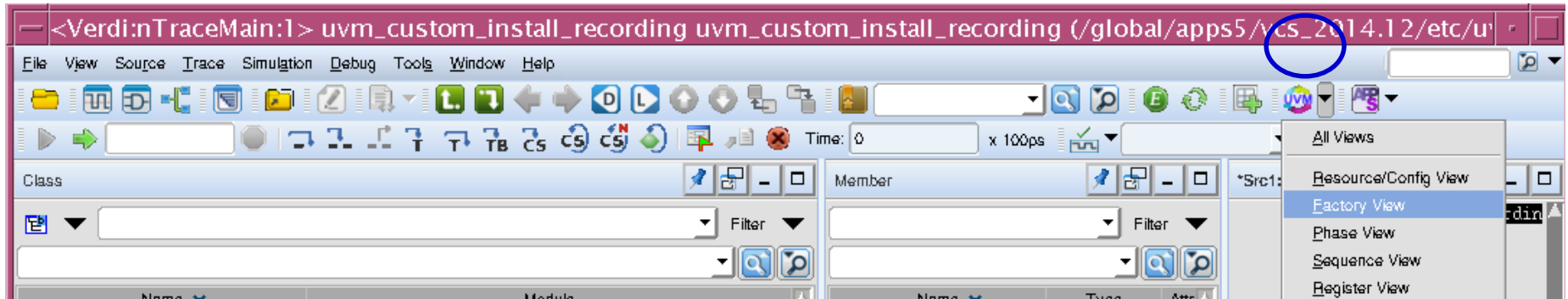
- All Objects
- UVM (Objects/Components)
- OVM (Objects/Components)
- VMM (Objects/Components)

■ Display and navigate all dynamic variables

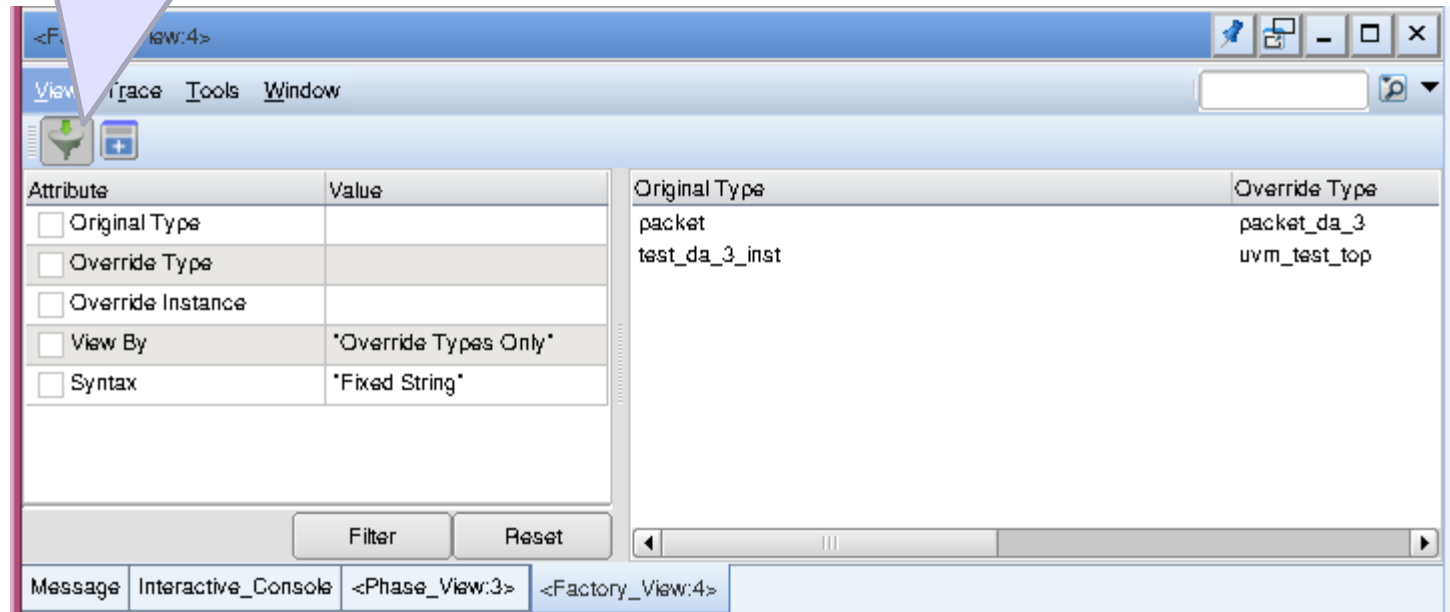
- Testbench hierarchy view
- Object creation time
- Object thread ids
- Object references
- Dynamic memory profiling
- Linked with Member Pane value annotation



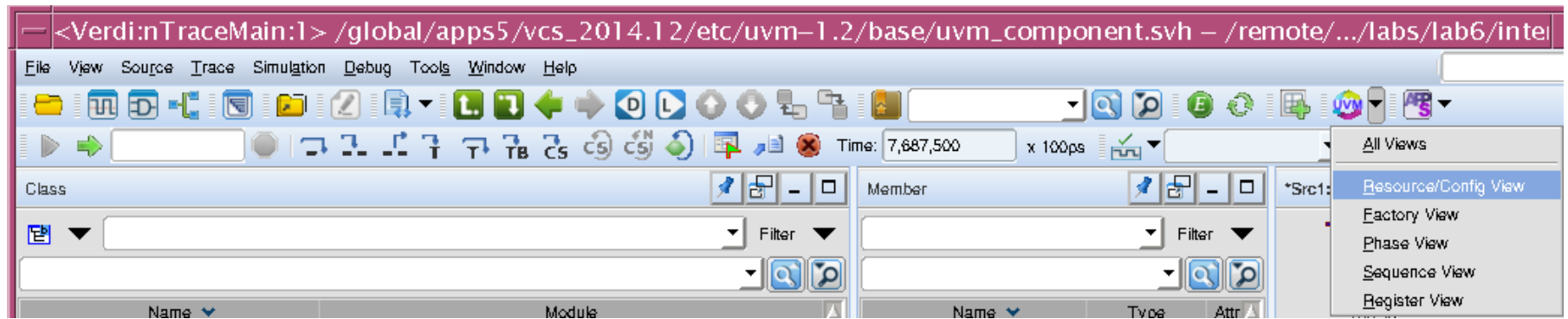
UVM Factory Debug



Filter control by name



UVM Resource Debug



The screenshot shows the <Resource_View:5> window. The left panel has a table with columns 'Attribute' and 'Value'. The 'Name' attribute is checked, and the 'Value' is 'default_sequence'. A callout bubble points to this section with the text 'Filter control by name'. The right panel has a table with columns 'Name', 'Scope', 'Value', and 'Type'. It lists several 'default_sequence' resources. Below this, there is another table with columns 'Attribute' and 'Value'. The 'View By' attribute is checked, and the 'Value' is '*All Calls*'. A callout bubble points to this section with the text 'Filter control by set/get'. A large arrow points from the 'View By' dropdown to a list of actions: '*All Calls*', '*Set Calls*', '*Set Calls Without Get*', '*Get Calls Without Set*', and '*Multiple Set Calls*'. The bottom of the window shows a message bar with 'Interactive_Console' and '<Resource_View:5>'.

Filter control by name

Filter control by set/get

6-44

UVM Phase-Based Breakpoint

The image shows the Verdi IDE interface with the 'Manage Breakpoints' dialog box open. The background window displays the UVM phase sequence, with 'post_main' highlighted. The 'Manage Breakpoints' dialog is configured for a 'Function/Task' breakpoint at 'stop -in uvm_component::post_main_phase -allthreads'. The 'Stop at' section has 'Begin' selected, and the 'Stop Frequency' section has 'Once' selected. The 'Create' button is circled in red.

Manage Breakpoints

VCS

Enable	ID	Type	Breakpoint
<input checked="" type="checkbox"/>	1	Function/Task	stop -in uvm_component::post_main_phase -allthreads

Jump to Source
Delete
Delete All
Enable All
Disable All

File/Line | Time | Value Change | Assertion | **Function/Task** | Constraint | Thread

Stop at: ☒ Begin ☐ End

Break in Task/Function:

Class Object:

Condition:

Stop Frequency: ☒ Once ☐ Repeat Skip Times before Stopping

☐ Create Automatic Checkpoint

Advanced >>

Create Update Clear

Save... Load... Close

UVM Phase Objection Debug

Phase History

Objection History

The screenshot displays the UVM Phase Objection Debug interface. The left pane shows a tree view of UVM phases, with 'main' selected under the 'uvm' domain. The right pane is divided into two sections: 'Phase History' and 'Objections in the selected phase'.

Phase History Table:

Domain	State	Time
uvm	DONE	18500
uvm	SCHEDULED	18500
uvm	STRT	18500
uvm	SKIP	18500
uvm	DONE	18500
uvm	SCHEDULED	18500
uvm	STRT	18500
uvm	DONE	7687500
uvm	SCHEDULED	7687500
uvm	STRT	7687500

Objections in the selected phase Table:

Begin Time	End Time	Action	Total	Count	Object
18500	18500	RAISE	1	1	uvm_test_top.env.i_a...
18500	18500	RAISE	2	1	uvm_test_top.env.i_a...
18500	18500	RAISE	3	1	uvm_test_top.env.i_a...
18500	18500	RAISE	4	1	uvm_test_top.env.i_a...
18500	18500	RAISE	5	1	uvm_test_top.env.i_a...
18500	18500	RAISE	6	1	uvm_test_top.env.i_a...
18500	18500	RAISE	7	1	uvm_test_top.env.i_a...
18500	18500	RAISE	8	1	uvm_test_top.env.i_a...
18500	18500	RAISE	9	1	uvm_test_top.env.i_a...
18500	18500	RAISE	10	1	uvm_test_top.env.i_a...
18500	18500	RAISE	11	1	uvm_test_top.env.i_a...

The interface also includes a 'Show Only Active Phase' checkbox and a bottom status bar with tabs for 'Message', 'Interactive_Console', '<Phase_View:3>', '<Factory_View:4>', and '<Resource_View:5>'.

UVM Sequence Debug

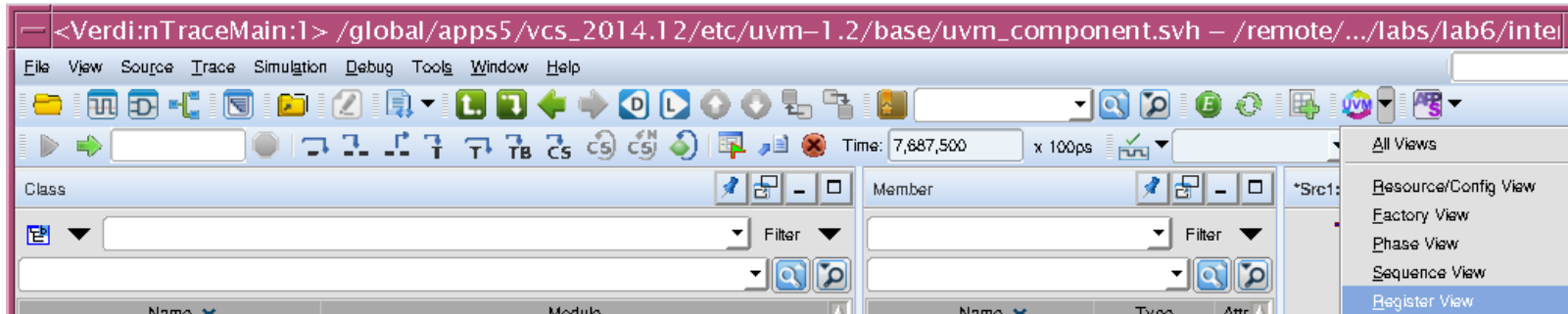
Filter Control

Attribute	Value
<input type="checkbox"/> Sequence	
<input type="checkbox"/> Sequence ID	
<input type="checkbox"/> Start Time	
<input type="checkbox"/> Finish Time	
<input type="checkbox"/> Phase	
<input type="checkbox"/> Thread	
<input type="checkbox"/> Sequencer	
<input type="checkbox"/> Sequencer ID	
<input type="checkbox"/> Active Sequence	"FALSE"
<input type="checkbox"/> Sequence Items	"FALSE"
<input type="checkbox"/> Syntax	"Fixed String"

Sequence	Sequence ID	Start Time	Finish Time	Phase	Thread
+ ral_port_unlock_s...	router_stimulus_...	17500	18500		3669
- packet_sequence	router_stimulus_...	18500	7035500		5810
req	router_stimulus_...	18500	43500		
req	router_stimulus_...	43500	747500		
req	router_stimulus_...	747500	1619500		
req	router_stimulus_...	1619500	2323500		
req	router_stimulus_...	2323500	2987500		
req	router_stimulus_...	2987500	3739500		
req	router_stimulus_...	3739500	4443500		
req	router_stimulus_...	4443500	5459500		5827
req	router_stimulus_...	5459500	6211500		5827
req	router_stimulus_...	6211500	7035500		5827
+ packet_sequence	router_stimulus_...	18500	7499500		5810
+ packet_sequence	router_stimulus_...	18500	7511500		5810
+ packet_sequence	router_stimulus_...	18500	7571500		5810
+ packet_sequence	router_stimulus_...	18500	7607500		5810
+ packet_sequence	router_stimulus_...	18500	7635500		5810
+ packet_sequence	router_stimulus_...	18500	7687500		5810
+ packet_sequence	router_stimulus_...	18500	7047500		5810
+ packet_sequence	router_stimulus_...	18500	7107500		5810
+ packet_sequence	router_stimulus_...	18500	7159500		5810
+ packet_sequence	router_stimulus_...	18500	7171500		5810
+ packet_sequence	router_stimulus_...	18500	7223500		5810
+ packet_sequence	router_stimulus_...	18500	7291500		5810
+ packet_sequence	router_stimulus_...	18500	7359500		5810

Message Interactive_Console <Phase_View:3> <Factory_View:4> <Resource_View:5> <Sequence_View:6>

UVM Register Debug



The screenshot shows the Register View window. The title bar is `<Register_View:7>`. The menu bar includes View, Trace, Tools, and Window. The Register Hierarchy on the left shows the selected register `regmodel` and its sub-entries `R_ARRAY` and `uvm_reg_map`. The main area displays the attributes of the selected register/memory, with a callout 'Filter control' pointing to the Filter button. The table below shows the register details:

Name	Offset	Width (bits)	Value	Desired
HOST_ID	h0	16	h0	h0
CHIP_ID	[15:8]	8	h0	h0
REV_ID	[7:0]	8	h0	h0
LOCK	h100	16	h0	h0
LOCK	[15:0]	16	h0	h0
RAM	h0	16		
R_ARRAY				
R_ARRAY[0]	h1000	16	h0	h0
R_ARRAY[1]	h1001	16	h0	h0
R_ARRAY[2]	h1002	16	h0	h0

Below the table, the 'Access history of the selected register' section is visible, with a callout 'Register access history' pointing to it. The table below shows the access history:

Register	Action	Time	Value
----------	--------	------	-------

The bottom status bar shows the current view is `<Register_View:7>`.