Improving Data Monitoring In UVM
Tips and Recommendations

Yogish Sekhar

Dialog Semiconductor Limited
Swindon, United Kingdom

http://www.dialog-semiconductor.com/

ABSTRACT

UVM Testbenches have evolved traditionally from Verilog and is structural and hierarchical. The ports/exports have to be connected in the connect_phase () at each level before data can be moved from the generators to the subscribers. If connections between the two have to be changed for some reason during the course of verifying a design, separate configuration is needed for the structural elements of the testbench; also the connections need to be traced through the hierarchy and made safe. This costs time and considerable effort to make simple changes. If we can break the structural hierarchical dependence on connecting ports/exports in the connect_phase () and move over to a more dynamic model where connections between generators and subscribers are created on an "As Need Basis" by improving the connection model and the associated helper code, productivity can be improved significantly. This paper discusses a potential "Dynamic Model" that can be used to improve data monitoring and additional recommendations.

Table of Contents

Table of Figures

Table of Tables

Table of Code Blocks

Table of Simulation Output Blocks

1. Introduction

Traditional testbench environments developed in UVM are inherently structural and limited by the quasi static connections between testbench components at various levels of hierarchy. These testbenches are subject to significant changes through the hierarchy; and are prone to errors when structural connections and/or elements need to change. Additionally when these UVM testbench environments are reused in higher level testbenches they introduce ancillary code needed to manage the data flow between various components of the testbench. This ancillary code impacts performance of the simulator as there are instances of the same structural element in each of the UVM environments.

In this paper we shall review the current connection techniques used for data monitoring limitations that are present in the current UVM methodology, potential solutions using late binding techniques, user experience using these solutions, and finally recommendations to enhance UVM base class library.

2. Review of Current UVM Testbench Structure

Figure 1 below provides an overview of elements in a typical UVM block level testbench. In this figure we observe that every data source and every subscriber is connected up via point to point TLM connection through the use of ports and exports.

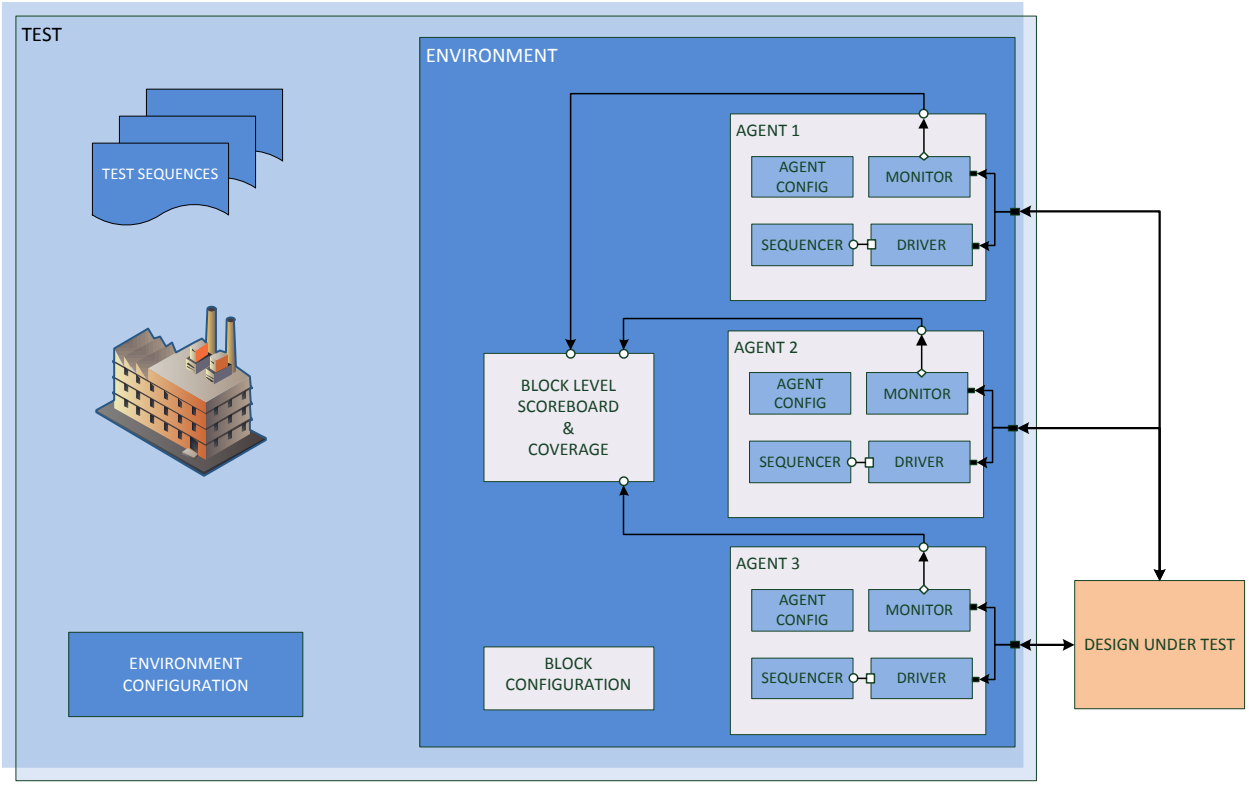

Figure 1 : Block Level Testbench

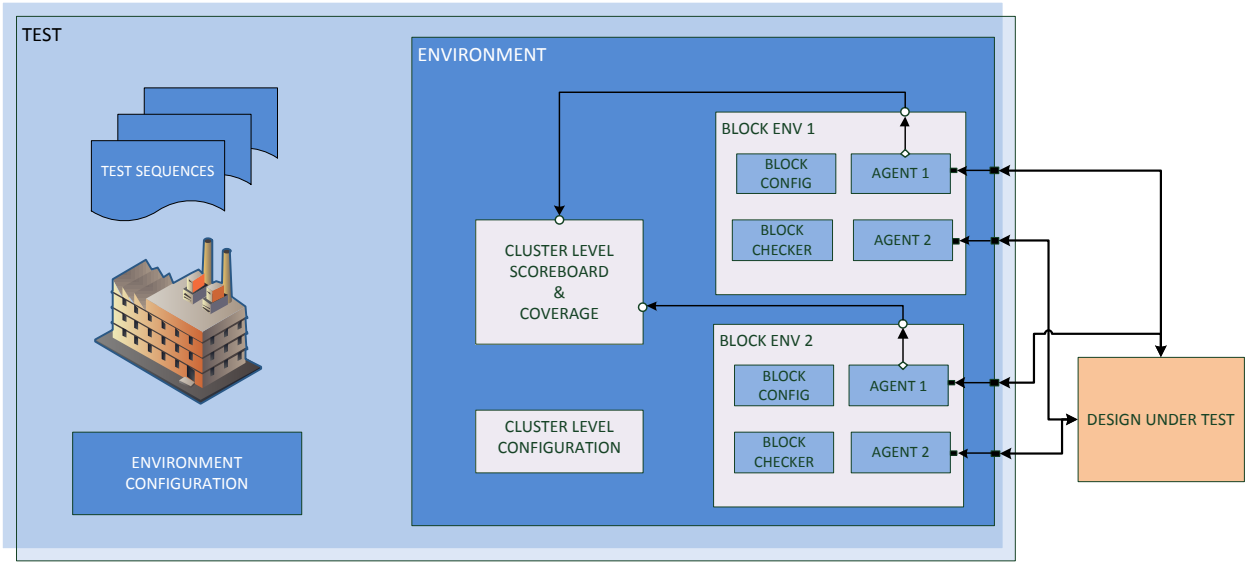

Figure 2 : Cluster/SOC Level Testbench

Figure 2 shows a UVM based cluster or toplevel SOC testbench which comprises of several block level environments. In this figure we observe that some subscribers i.e. checkers/scoreboards are located at a different level of hierarchy (minimum of 2 levels) than the data source ie the monitors. This implies that each block level UVM environment has to export connections through to all the agents instanced in the blocklevel environment which provide information to cluster level/SOC level checkers or scoreboards.

Building up a UVM testbench is achieved in a phased manner. Ideally all testbench components are built in the *build_phase* and each of the testbench components is connected up in the *connect_phase*. The connect_phase uses a bottom up approach in connecting up the testbench and every level in the hierarchy can only connect up elements present at its level. This restriction in the way '`connect_phase`' can work introduces ancillary code at every level just to move data from source to destination.

Code Block 1, Code Block 2, & Code Block 3, below show the connect phase of various UVM testbench components. Here we observe that the analysis port located in the monitor instanced in an agent being connected to an implementation in a checker/scoreboard instanced in a block environment or in the SOC environment through each level of hierarchy. Thus each level of hierarchy i.e. agent and the environment will need to support the analysis port even though they do not process the transaction provided by the analysis port. This introduces ancillary code and increases debug time if connection errors or any issues are observed in the plumbing of the testbench.

```
function void agent::connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(name, "Inside agent::connect()", UVM_LOW);
    if (driver != null && sequencer != null) begin
      // connect the driver with the sequencer
      driver.seq_item_port.connect (sequencer.seq_item_export);
    end

    monitor.analysis_port.connect (monitor_ap);
endfunction : connect_phase
```

Code Block 1 : Agent Connect Phase

```
function void block_env::connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(name, "Inside block_env::connect()", UVM_LOW);

    agent_1.monitor_ap.connect    (block_checker.agent_1_ap);
    agent_2.monitor_ap.connect    (block_checker.agent_2_ap);
    agent_1.monitor_ap.connect    (this.agent_1_ap);
    agent_2.monitor_ap.connect    (this.agent_2_ap);
endfunction : connect_phase
```

Code Block 2 : Block Environment Connect Phase

```
function void cluster_env::connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(name, "Inside block_env::connect()", UVM_LOW);

    block_1.agent_1_ap.connect    (cluster_checker.block1_agent1_ap);
    block_1.agent_2_ap.connect    (cluster_checker.block1_agent2_ap);

    block_2.agent_1_ap.connect    (cluster_checker.block2_agent1_ap);
    block_2.agent_2_ap.connect    (cluster_checker.block2_agent2_ap);
endfunction : connect_phase
```

Code Block 3 : Cluster Environment Connect Phase

UVM is developed using system verilog which is object oriented. Productivity can be improved by moving away from quasi structural testbenches and embracing some of the techniques such as agile development, and late binding. The following chapters introduce the potential improvements needed to *'uvm_monitor'* class and will also discuss improvements to the analysis port implementation macro which is essential for a dynamic testbench to allow late binding.

3. Limitations with Analysis Port Macro's

In order to create an active connection between any data source i.e. a monitor and a subscriber i.e. checker/scoreboard four different elements listed in Table 1 are needed

1.	Data Source	Element that generate data, typically monitors
2.	Transaction Type	Object type that the data source generates
3.	Destination Type	Object type of the receiver/destination
4.	Destination Method	Method in the receiver/destination that is capable of processing the transaction received from the data source

Table 1 : Elements for Active Connection

Currently UVM enforces a four stage process to establish an active connection between a data source and its subscriber.

Firstly, one of `'*_imp_decl (<_suffix>)'` macros needs to be used to define a port object and the expected method `'*<_suffix> ()'` that the subscriber needs to provide an implementation based on the type of macro used. One of the most commonly used uvm macro for providing an implementation for an analysis port is – `'uvm_analysis_imp_decl (<_suffix>)'` which defines a template class `'uvm_analysis_imp_suffix #(type T=<transaction_type>, type IMP=<destination_type>)'` and places a restriction that a method `'write_suffix( T trans)'` is implemented in the subscriber which needs to be connected to the data source.

Secondly, the template class for the port defined in the first stage needs to be customised by providing it with a specific transaction type and the subscriber type where the final implementation for the analysis port will be available

Thirdly, an instance of the customised port needs to be created passing it a reference handle of the subscriber.

Finally, the port needs to be connected to the data source via a sequence of export-port connections through each hierarchical component of the testbench that contains the data source and the subscriber; or use hierarchical reference at the environment to connect up the data source and subscriber. Hence the reason why build_phase() uses a top-down approach to build all the elements used in the test and connect_phase() uses a bottom-up approach.

The above four stage process has three major limitations.

1- Restriction on the name of the method that can be used in the subscriber
2- The need to perform customisation of the template class. This process can be defined inside the macro itself
3- The need for using connect_phase() through the hierarchy.

4. Improving UVM Monitor

The 'uvm_monitor' in the UVM library shown in Code Block 4 below is just a wrapper on 'uvm_component'.

```
virtual class uvm_monitor extends uvm_component;

  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  const static string type_name = "uvm_monitor";

  virtual function string get_type_name ();
    return type_name;
  endfunction

endclass
```

Code Block 4 : Library - uvm_monitor

A specific implementation of the monitor is usually associated with an interface and will always provide a specific type of transaction to its subscribers. Hence just as 'uvm_driver', 'uvm_monitor' can be changed into a template as shown in Code Block 5.

```
virtual class dvm_monitor #(type T = uvm_sequence_item) extends uvm_monitor;
  T                     mon_trans;

  // Declare a  default analysis port in the monitor
  uvm_analysis_port#(T) default_ap;

  // registration bit for callbacks
  static local bit m_register_cb;

  `uvm_component_param_utils(dvm_monitor #(T))

  function new (string name, uvm_component parent);
    super.new(name, parent);
    default_ap = new ({name.toupper(), "_DEFAULT_AP"}, this);
    m_register_cb = uvm_callbacks#(dvm_monitor#(T),
                    dvm_callback#(T))::m_register_pair(this.get_type_name(),
                    {name.toupper(),"_MONITOR_CB"});
  endfunction

  function void notify(T trans);
    // Indicate the current transaction being driven.
    uvm_report_info(name, $psprintf("Notifying transaction: \n%s",
                    trans.sprint()), UVM_HIGH);
    `uvm_do_callbacks(dvm_monitor#(T), dvm_callback#(T), indicated(trans))
    default_ap.write(trans);
  endfunction : notify
endclass : dvm_monitor
```

Code Block 5 : Potential - uvm_monitor

The monitor template described above will provide some common elements that are typically present in any monitor such as default analysis port to export transactions, hooks for callback and a broadcast method called 'notify()' which transfer interesting transactions to various subscribers which are connected either by analysis ports and/or callbacks. Specific monitor

implementations should now derive from the above described monitor template using a specific transaction type which is derived from 'uvm_sequence_item' and provide a 'run_phase()' implementation that monitors the physical interface, creates transactions at interesting events/point, and call the 'notify()' method to pass the relevant transactions that need to be broadcast to connected subscribers.

5. Improving Analysis Port Macro

Table 1 : Elements for Active Connection in Section 3 identified the elements required in order to establish an active connection between a data source and its subscriber. Section 3 also highlighted the limitations currently enforced by the library code on analysis ports. Improving the analysis port macro will aid in overcoming the limitations introduced due to the current macro.

Code Block 6 provides an overview of a potential implementation for the macro that would allow to overcome the restrictions placed by the current implementation. The proposed implementation still uses all the four elements that define an active connection and enforces a simplified three stage process i.e. using the macro, creating an instance of the analysis object, and finally calling the analysis object's 'auto_connect()' method to establish the connection to the data source i.e. a monitor.

```
`define dvm_analysis_imp_decl(ANALYSIS_OBJ_TYPE, MONITOR_NAME, TRANS_TYPE, RCV_TYPE, RCV_FUNC)  \
class ANALYSIS_OBJ_TYPE extends uvm_port_base #(uvm_tlm_if_base #(TRANS_TYPE, TRANS_TYPE));     \
  `UVM_IMP_COMMON(`UVM_TLM_ANALYSIS_MASK,`"ANALYSIS_OBJ_TYPE`",RCV_TYPE)                         \
  function void write( TRANS_TYPE t);                                                           \
    m_imp.``RCV_FUNC(t);                                                                        \
  endfunction                                                                                   \
                                                                                                \
  virtual function void auto_connect();                                                        \
    string                   monitor_name;                                                    \
    dvm_monitor#(TRANS_TYPE) _monitor;                                                        \
    uvm_component            _monitor_list[$];                                                \
                                                                                              \
    monitor_name = MONITOR_NAME;                                                              \
    this.m_imp.uvm_report_info(get_name(), {`"ANALYSIS_OBJ_TYPE`", "::connect()"}, UVM_MEDIUM); \
    this.m_imp.uvm_report_info(get_name(),                                                    \
                               {"Finding Monitor: ", monitor_name.toupper()}, UVM_LOW);       \
    uvm_top.find_all (monitor_name, _monitor_list);                                           \
    if (_monitor_list.size() > 1) begin                                                       \
      this.m_imp.uvm_report_fatal(get_name(), {monitor_name, " - Monitor Not Unique In The    \
                                               Environment. Please Specify Unique Name"});    \
    end                                                                                       \
    else if (_monitor_list.size() == 0) begin                                                 \
      this.m_imp.uvm_report_fatal(get_name(), {monitor_name, " - Monitor Not Found In The     \
                                               Environment. Please Specify Unique Name"});    \
    end                                                                                       \
    $cast (_monitor, _monitor_list[0]);                                                       \
    _monitor.default_ap.connect(this);                                                        \
    this.m_imp.uvm_report_info(get_name(),                                                    \
                               {"Connected       : ", _monitor.get_full_name(),               \
                                "To              : ", this.m_imp.get_full_name(),             \
                                "Transaction     : ", `"TRANS_TYPE`",                         \
                                "Function Call   : ", `"RCV_FUNC`"}, UVM_LOW);                \
  endfunction                                                                                 \
endclass
```

Code Block 6 : Improved Analysis Port Macro

The macro needs five elements to be provided in order to customise and create a singular connection between the data source and the subscriber. These are listed in Table 2 : Parameters for New Analysis Object Macro.

1.	`ANALYSIS_OBJ_TYPE`	Unique name to identify the customised analysis object class type
2.	`MONITOR_NAME`	Name string to uniquely identify the monitor component in the testbench. This string can contain wild characters to identify hierarchical components
3.	`TRANS_TYPE`	Data transaction class type used by the monitor
4.	`RCV_TYPE`	Class type of the receiver
5.	`RCV_FUNC`	User defined method available in the in the subscriber that processes a transaction of type `'TRANS_TYPE'`

Table 2 : Parameters for New Analysis Object Macro

The '`auto_connect()`' method defined in the analysis object makes the hierarchical '`connect_phase()`' redundant. It uses the built in UVM component registry to find the necessary data sources and connects the analysis object instance to the default analysis port present in the monitor. Since the analysis object has all the necessary information needed to pass the transaction to the subscriber when the data source invokes the '`notify()`' method, an active connection between the data source and subscriber is dynamically established.

Using the analysis object, connections are establised by the subscribers that need the information for the respective and not by any hierachical components present between the data source and the subscribers. These dynamic connections can be established anytime after the `'build_phase()'` has completed and is established in one place i.e. the subscriber. This mechanism provides us with a flexibility of establishing data connection in the 'run_phase()' and also aids in managing code and simplifies debug as all connection information is available in a single class or file.

Section 7: Example Testbench, highlights the dynamic connection model and provide simulation extracts to show how connections can be established.

6. TLM and UVM

TLM 2.0 provides a standardized approach for creating models and transaction-level simulations. The standard enables exchange of models and a common ground for interfacing. A simple but solid architecture allows TLM newcomers to quickly get up to speed and produce interoperable models. For seasoned TLMers, the standard provides a solution for the increasingly hard problem of model interoperability[8].

UVM ports, exports, and analysis ports are also based on OSCI standard and SystemC implementation for TLM. Unlike SystemC analysis ports, UVM is restrictive in the way analysis ports can be bound or connected. SystemC analysis ports are not derived from 'sc_port' and have no restrictions on when and how they are bound. Connections in SystemC analysis ports can be established even in the 'run_phase'. Whereas in UVM all ports are derived from the same 'uvm_port_base' class and enforce a restriction that analysis port connections need to be resolved before the 'run_phase'. This restriction has a significant impact on performance particularly when executing SOC level hardware-software co-simulations using all block level checkers/environments.

In the current setup using UVM, all analysis port connections in a co-simulation environment need to be resolved at time '0'. This would mean instancing all block level environments at a SOC level even before the SOC is initialised. The disadvantages' using this method is that the SOC co-simulation environment has:

- Increased memory footprint
- Inability to dynamically enable certain block level environment based on software testcases after SOC initialisation
- Re-Initialisation of SOC in every testcase if enabling of block level checkers is needed at runtime by use of plusargs

All the above disadvantages significantly increase simulation runtime and cost to perform hardware-software co-simulation. If UVM analysis ports can inherit from a new class as described in Code Block 7 : Potential 'uvm_analysis_port_base.sv' which is similar to 'uvm_port_base.sv', but will remove the restrictions on analysis port bindings and enable UVM analysis ports to establish connections in the 'run_phase'.

```
virtual function void connect (this_type provider);
    uvm_root top = uvm_root::get();
    // Remove Connection checks for late connection
    /*
    if (end_of_elaboration_ph.get_state() == UVM_PHASE_EXECUTING || // TBD tidy
        end_of_elaboration_ph.get_state() == UVM_PHASE_DONE ) begin
      m_comp.uvm_report_warning("Late Connection",
        {"Attempt to connect ",this.get_full_name()," (of type,this.get_type_name(),
         ") at or after end_of_elaboration phase.  Ignoring."});
      return;
     end
     */

    if (provider == null) begin
      m_comp.uvm_report_error(s_connection_error_id,
```

```
                        "Cannot connect to null port handle", UVM_NONE);
      return;
    end
    ……
    ……
    ……
    ……

    void'(m_check_relationship(provider));
    m_provided_by[provider.get_full_name()] = provider;
    provider.m_provided_to[get_full_name()] = this;

    // Add binding resolutions if connections established after post_elab_phase
    if (end_of_elaboration_ph.get_state() == UVM_PHASE_DONE ) begin
       resolve_bindings();
    end

  endfunction


virtual function void resolve_bindings();
    // Allow binding to be resolved even if previously called
    /*
    if (m_resolved) return;
    */
    ……
    ……
  endfunction
```

Code Block 7 : Potential 'uvm_analysis_port_base.sv'

The effect of the change described in Code Block 7 will now allow for building a hardware-software co-simulation that can have analysis port connection resolved not just at time '0', but also after an arbitrary simulation time where the SOC software can be initialised and a snapshot can be built. Once the initialised snapshot is built, testcases then can use this snapshot and enable various block level environments on a testcase by testcase basis using plusargs. This reduces co-simulation runtime as SOC software initialisation is done once and reused across multiple testcases. There is also a reduction on the memory foot print as not all block environments are now enabled by default in the co-simulation.

Allowing analysis port connections to be established in the 'run_phase' provides additional advantages when used with all the other improvements suggested earlier on in the paper. The simplification in the overall environment code makes the environments more maintainable and improves the mechanism used to port block level environment code to SOC level. The disadvantage that is foreseeable is that a barrier needs to be established in the SOC environment such that all block level environments created in the 'run_phase' are synchronised and their respective build, connect, and elaboration phases are executed before any further simulation time is expended.

Furthermore the UVM analysis port and connection model needs to be improved to get it in sync with the OSCI standard & SystemC implementation. Potentially using system verilog interfaces as described in the 2012 LRM.

7. Example Testbench

Case 1: Block Level UVM Environment

Figure 3 below provides an overview of a block level UVM environment built for the example testbench.

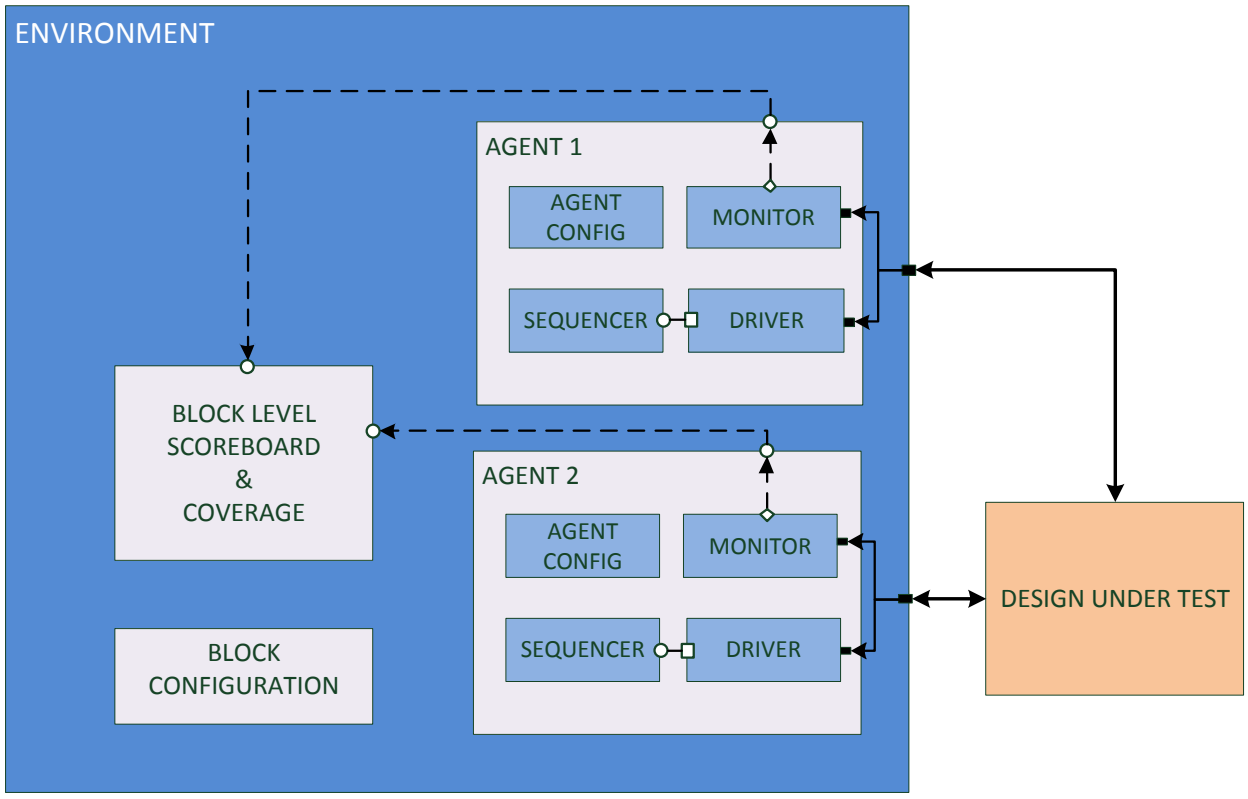

Figure 3 : Example Testbench – Block Level Environment

The environment is constructed in a traditional sense and contains multiple agents and checkers/scoreboards. Traditionally the checkers would have been connected via export-port connections as depicted by the dotted lines. But when using the dynamic connections, the checker will establish a connection with the monitors when required. Code Block 8 : Block Level Checker shown below provides an example of usage for the analysis port macro. The checker will be connected to two different agents as shown by the usage of macro `` `dvm_analysis_imp_decl ``.

```
class snug2014_checker_1 extends uvm_component;

  // Create the classes needed for analysis objects
  `dvm_analysis_imp_decl(
       CK1_AP_MON_1, "*.snug2014_block_env_1.*.agent_1_monitor", snug2014_agent_1_transaction,
       snug2014_checker_1, func_agent_1)
  `dvm_analysis_imp_decl(
       CK1_AP_MON_2, "*.snug2014_block_env_1.*.agent_2_monitor", snug2014_agent_2_transaction,
       snug2014_checker_1, func_agent_2)

  // Declare the Analysis Objects that are to be used
  CK1_AP_MON_1          ap_mon_1;
  CK1_AP_MON_2          ap_mon_2;
```

```
  `uvm_component_utils_begin(snug2014_checker_1)
  `uvm_component_utils_end

  // UVM Functions
  extern function new(string name = "snug2014_checker_1", uvm_component parent=null);
  extern virtual function void build_phase(uvm_phase phase)  ;
  extern virtual function void connect_phase(uvm_phase phase);

  // User Defined Callback functions for Analysis
  extern function void func_agent_1  (snug2014_agent_1_transaction      trans);
  extern function void func_agent_2  (snug2014_agent_2_transaction      trans);

endclass : snug2014_checker_1

function void snug2014_checker_1::build_phase(uvm_phase phase);
  super.build_phase(phase);

  `uvm_info (get_name(), "Inside snug2014_checker_1::build_phase()", UVM_LOW);

  // Create the Analysis Object that links to the Monitors in the TB
  // Analysis Objects do not support type_id::create(), hence the use of new
  ap_mon_1        = new("ANALYSIS_PORT_AGENT_1", this);
  ap_mon_2        = new("ANALYSIS_PORT_AGENT_2", this);
endfunction : build_phase

function void snug2014_checker_1::connect_phase(uvm_phase phase);
  `uvm_info (get_name(), "Inside snug2014_checker_1::connect_phase()", UVM_LOW);

  // Call auto_connect inorder to connect up the monitor and the checker
  ap_mon_1.auto_connect();
  ap_mon_2.auto_connect();
endfunction : connect_phase
```

Code Block 8 : Block Level Checker

Code Block 8 : Block Level Checker also shows usage of the '`auto_connect()`' method in the checker '`connect_phase()`'.

```
function void agent::connect_phase(uvm_phase phase);
  if (driver != null && sequencer != null) begin
    // connect the driver with the sequencer
    driver.seq_item_port.connect (sequencer.seq_item_export);
  end

  // No connections to the monitor
endfunction: connect_phase
```

Code Block 9 : Example Testbench - Agent Connect Phase

Code Block 9 : Example Testbench - Agent Connect Phase shown above provides an overview of the '`connect_phase()`' implemented when using the dynamic connection flow. Comparing with the code present in Code Block 1 : Agent Connect Phase, we can observe that the connect phase is simplified with no support for monitor connections.

Similarly, comparing code shown in Code Block 10 : Example Testbench – Block Environment Connect Phase and Code Block 2 : Block Environment Connect Phase, we observe the connect phase using the dynamic connection model reduces the amount of support code needed. Hence simplifying the environment construction.

```
function void snug2014_block_env_1::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  `uvm_info(get_name(), "Inside snug2014_block_env_1::connect_phase()
                         - Nothing Done Here", UVM_LOW);
endfunction : connect_phase
```

Code Block 10 : Example Testbench – Block Environment Connect Phase

Output Block 1 provides a snapshot of the connect phase when a simulation is invoked for the example testbench. Here we observe a dynamic connection is established between two agent monitors and the block level checker. The connect phase is only implemented by the structural elements that need to establish connections and not for any other component in the testbench.

```
UVM_INFO @ 0ns : [checker_1]             : Inside snug2014_checker_1::connect_phase()
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_1]: CK1_AP_MON_1::auto_connect()
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_1]:
              Finding Monitor: *.SNUG2014_BLOCK_ENV_1.*.AGENT_1_MONITOR
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_1]:
              Connected    : uvm_test_top.snug2014_block_env_1.agent_1.agent_1_monitor
              To           : uvm_test_top.snug2014_block_env_1.snug2014_checker_1
              Transaction  : snug2014_agent_1_transaction
              Function Call: func_agent_1
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_2]: CK1_AP_MON_2::auto_connect()
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_2]:
              Finding Monitor: *.SNUG2014_BLOCK_ENV_1.*.AGENT_2_MONITOR
UVM_INFO @ 0ns : [ANALYSIS_PORT_AGENT_2]:
              Connected    : uvm_test_top.snug2014_block_env_1.agent_2.agent_2_monitor
              To           : uvm_test_top.snug2014_block_env_1.snug2014_checker_1
              Transaction  : snug2014_agent_2_transaction
              Function Call: func_agent_2
UVM_INFO @ 0ns : [snug2014_env_1]      :
              Inside snug2014_block_env_1::connect_phase() - Nothing Done Here
```

Output Block 1 : Example Testbench – Block Level Simulation

Output Block 2 : Example Testbench – Transaction Transfer, below provides a snapshot of how data transfer is achieved once a dynamic connection is established.

```
UVM_INFO @ 100ns : [agent_1_monitor] : Something Interesting Seen Here - Build Transaction
UVM_INFO @ 100ns : [agent_1_monitor] : dvm_monitor::notify() - Called
UVM_INFO @ 100ns : [checker_1]       : Inside snug2014_checker_1::func_agent_1()
UVM_INFO @ 200ns : [agent_2_monitor] : Something Interesting Seen Here - Build Transaction
UVM_INFO @ 200ns : [agent_2_monitor] : dvm_monitor::notify() - Called
UVM_INFO @ 200ns : [checker_1]       : Inside snug2014_checker_1::func_agent_2()
```

Output Block 2 : Example Testbench – Transaction Transfer

In the snapshot we observe a monitor transferring data to a checker whenever it sees some interesting event. It builds a transaction that the connected subscriber will process and calls the inbuilt '`notify()`' method to transfer the transaction. The monitor is not concerned if the subscribers are connected via analysis ports or by callbacks but only to broadcast a transaction for every interesting event. If there are no connected clients to the monitor the notify method will eventually drop the transaction and does nothing.

Case 2: SOC Level UVM Environment

Figure 4 below provides an overview of a SOC level UVM environment built for the example testbench.

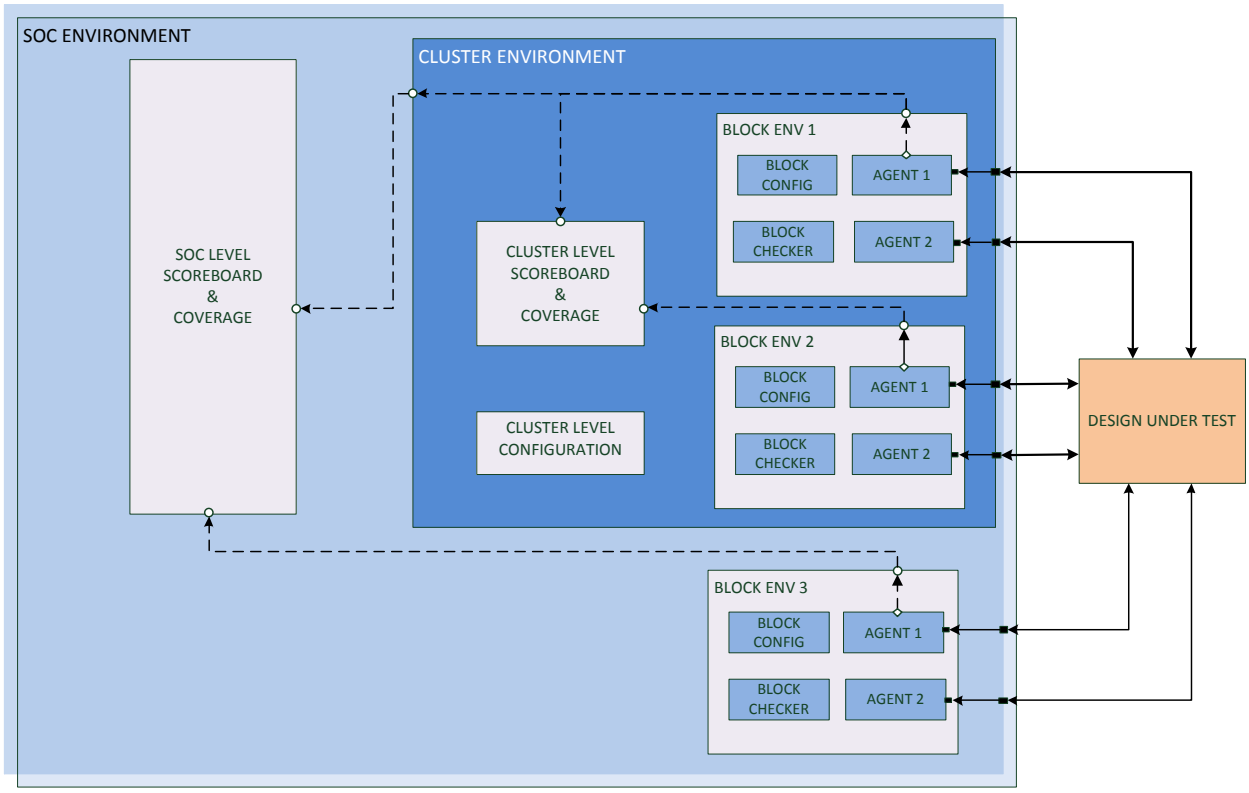

Figure 4 : Example Testbench – SOC Level Environment

The environment is constructed similarly to the block level environment described in the Case 1. The SOC environment has a lot more levels of hierarchy and establishing a connection in the traditional sense would have created a lot ancillary code in the environment which is technically redundant.

Code Block 11 : Example Testbench - SOC Level Checker, provided an overview of the checker that is implemented. Comparing it with Code Block 8 : Block Level Checker we observe that the implementation is similar in that the analysis macro is used to connect up three monitors located in three different environments in the hierarchy. Additionally, there are two instances of the same monitor connected to the checker from two different block environments.

```
class snug2014_soc_checker extends uvm_component;
  `dvm_analysis_imp_decl(CK_SOC_C1_AP_MON_2, "*.snug2014_cluster_env.*.agent_2_monitor",
       snug2014_agent_2_transaction, snug2014_soc_checker, cluster_func_agent_2)
  `dvm_analysis_imp_decl(CK_SOC_C1_AP_MON_4, "*.snug2014_cluster_env.*.agent_4_monitor",
       snug2014_agent_4_transaction, snug2014_soc_checker, cluster_func_agent_4)
  `dvm_analysis_imp_decl(CK_SOC_AP_MON_4    , "*.snug2014_block_env_3.*.agent_4_monitor",
       snug2014_agent_4_transaction, snug2014_soc_checker, block_func_agent_4  )

  // Declare the Analysis Connection Objects
  CK_SOC_C1_AP_MON_2         cl_ap_mon_2 ;
  CK_SOC_C1_AP_MON_4         cl_ap_mon_4 ;
  CK_SOC_AP_MON_4            blk_ap_mon_4;
```

```
  `uvm_component_utils_begin(snug2014_soc_checker)
  `uvm_component_utils_end

  // UVM Functions
  extern function new(string name = "snug2014_soc_checker", uvm_component parent=null);
  extern virtual function void build_phase(uvm_phase phase)  ;
  extern virtual function void connect_phase(uvm_phase phase);

  // User Defined Callback functions for Analysis
  extern function void cluster_func_agent_2  (snug2014_agent_2_transaction      trans);
  extern function void cluster_func_agent_4  (snug2014_agent_4_transaction      trans);
  extern function void block_func_agent_4    (snug2014_agent_4_transaction      trans);

endclass : snug2014_soc_checker

function void snug2014_soc_checker::build_phase(uvm_phase phase);
  super.build_phase(phase);

  `uvm_info (get_name(), "Inside snug2014_soc_checker::build_phase()", UVM_LOW);

  // Create the Function Object that links to the Monitors in the TB
  // Function Objects do not support type_id::create(), hence the use of new
  cl_ap_mon_2        = new("SOC_CHK_CL_ANALYSIS_CONN_OBJ_2" , this);
  cl_ap_mon_4        = new("SOC_CHK_CL_ANALYSIS_CONN_OBJ_4" , this);
  blk_ap_mon_4       = new("SOC_CHK_BLK_ANALYSIS_CONN_OBJ_4", this);
endfunction : build_phase


function void snug2014_soc_checker::connect_phase(uvm_phase phase);
  `uvm_info (get_name(), "Inside snug2014_soc_checker::connect_phase()", UVM_LOW);

  cl_ap_mon_2.auto_connect();
  cl_ap_mon_4.auto_connect();
  blk_ap_mon_4.auto_connect();
endfunction : connect_phase
```

Code Block 11 : Example Testbench - SOC Level Checker

Similar to the connect_phase() for the block level environment described in Code Block 10 : Example Testbench – Block Environment Connect Phase, Code Block 12 and Code Block 13 below shows the connect_phase() for the cluster level environment and SOC level environment are just empty functions. Comparing Code Block 12 and Code Block 13 to Code Block 3 : Cluster Environment Connect Phase, we observe that the `'connect_phase()'` is significantly simplified and the environment is easier to maintain.

```
function void snug2014_cluster_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  `uvm_info(get_name(), "Inside snug2014_cluster_env::connect_phase()
                         - Nothing Done Here", UVM_LOW);
endfunction : connect_phase
```

Code Block 12 : Example Testbench – Cluster Level Environment Connect Phase

```
function void snug2014_soc_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  `uvm_info(get_name(), "Inside snug2014_soc_env::connect_phase()
                         - Nothing Done Here", UVM_LOW);
endfunction : connect_phase
```

Code Block 13 : Example Testbench – SOC Environment Connect Phase

Output Block 3 provides a snapshot of the connect phase when a simulation is invoked for the SOC Level example testbench. Here we observe a dynamic connection is established between SOC Level checker and agent monitors embedded within cluster and block level environments. We also observe it is quite simple to resolve hierarchical path information.

```
UVM_INFO @ 0ns : [checker_soc]                    : Inside snug2014_soc_checker::connect_phase()
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_2] : CK_SOC_C1_AP_MON_2::auto_connect()
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_2] : Finding Monitor:
           *.SNUG2014_CLUSTER_ENV.*.AGENT_2_MONITOR
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_2] :
           Connected   : uvm_test_top.snug2014_soc_env.snug2014_cluster_env
                         .snug2014_block_env_1.agent_2.agent_2_monitor
           To          : uvm_test_top.snug2014_soc_env.checker_soc
           Transaction : snug2014_agent_2_transaction
           Function    : cluster_func_agent_2
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_4] : CK_SOC_C1_AP_MON_4::auto_connect()
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_4] : Finding Monitor:
           *.SNUG2014_CLUSTER_ENV.*.AGENT_4_MONITOR
UVM_INFO @ 0ns : [SOC_CHK_CL_ANALYSIS_CONN_OBJ_4] :
           Connected   : uvm_test_top.snug2014_soc_env.snug2014_cluster_env
                         .snug2014_block_env_2.agent_4.agent_4_monitor
           To          : uvm_test_top.snug2014_soc_env.checker_soc
           Transaction : snug2014_agent_4_transaction
           Function    : cluster_func_agent_4
UVM_INFO @ 0ns : [SOC_CHK_BLK_ANALYSIS_CONN_OBJ_4]: CK_SOC_AP_MON_4::auto_connect()
UVM_INFO @ 0ns : [SOC_CHK_BLK_ANALYSIS_CONN_OBJ_4]: Finding Monitor:
           *.SNUG2014_BLOCK_ENV_3.*.AGENT_4_MONITOR
UVM_INFO @ 0ns : [SOC_CHK_BLK_ANALYSIS_CONN_OBJ_4]:
           Connected   : uvm_test_top.snug2014_soc_env.snug2014_block_env_3
                         .agent_4.agent_4_monitor
           To          : uvm_test_top.snug2014_soc_env.checker_soc
           Transaction : snug2014_agent_4_transaction
           Function    : block_func_agent_4
UVM_INFO @ 0ns : [snug2014_block_env_1]           : Inside snug2014_block_env_1::connect_phase()
                                                     - Nothing Done Here
UVM_INFO @ 0ns : [snug2014_block_env_2]           : Inside snug2014_block_env_2::connect_phase()
                                                     - Nothing Done Here
UVM_INFO @ 0ns : [snug2014_cluster_env]           : Inside snug2014_cluster_env::connect_phase()
                                                     - Nothing Done Here
UVM_INFO @ 0ns : [snug2014_soc_env]               : Inside snug2014_soc_env::connect_phase()
                                                     - Nothing Done Here
```

Output Block 3 : Example Testbench – SOC Level Simulation

8. Conclusions

This paper has presented an alternative approach using dynamic late binding compared to traditional testbench construction techniques that are employed today. Additionally this paper has discussed some of the potential improvements that are possible for 'uvm_monitor' and the supporting analysis macros used.

The advantages of using the 'Dynamic Connection Model' proposed in the paper are

- Reduction is the code needed to support data movement in testbenches
- Automating connection process between data source (monitors) and subscribers (checkers/scoreboards)
- Eliminating the need for connect_phase() in components that do not process any transactions
- 'connect_phase()' limited to subscribers that process transactions and need establishing connections with a data source, Library ensures transaction type checking at compile time
- Simplifies the connection model to allow for callbacks to be used just as analysis ports and allow for feedback to active sequences which need information from monitors. E.g. Interrupt Monitoring
- Allows the use of wild character based search string for establishing connections
- Simplifies debug as all information regarding establishing a data connection is available at a single location

The limitations of using the 'Dynamic Connection Model' proposed in the paper are

- The data source (monitors) uses a broadcast mechanism and effectively is limited to using a push model for data transmission
- Data source search path strings used in the subscribers (checkers/scoreboard) should be uniqifiable (i.e. identify a single data source) when trying to establish a dynamic connection

Overall the dynamic connection technique will reduce environment code maintenance. It also aids in reducing debug time spent on environment when different environment are brought together as all elements needed to create a connection are located in a single file.

9. References

[1] IEEE Std 1800-2012 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language"

[2] http://www.accellera.org/downloads/standards/uvm

[3] http://www.accellera.org/community/uvm/

[4] https://verificationacademy.com/

[5] Universal Verification Methodology (UVM) 1.1d Class Reference

[6] Universal Verification Methodology (UVM) 1.1 User's Guide

[7] Doulos System Verilog & UVM Training Material

[8] Transaction Level Modelling using OSCI TLM 2.0 By Marcelo Montoreano, Synopsys, Inc. 2007