Coverage Driven Verification of an Unmodified DUT within an OVM Testbench

Michael Baird
Willamette HDL, Inc
14314 SW Allen Blvd., Suite 625
Beaverton, OR 97005
+1 503-590-8499

ABSTRACT

A coverage-driven verification plan defines verification goals in terms of functional coverage points. Each area of functionality required to be tested is described in terms of values, events and combinations of these. SystemVerilog provides covergroups as one way of obtaining coverage statistics to drive the testing activities.

In a SystemVerilog Open Verification Methodology (OVM) testbench it is rather straight forward to create covergroups to obtain coverage results for points that are internal to the testbench itself and from objects, such as drivers and monitors that provide access to the pins of the Device Under Test (DUT). Hence this is what is typically done. What is often desirable is creating covergroups that provide coverage of the internals of the DUT, without modifying the DUT itself, and then providing the coverage information to objects, such as coverage collectors inside the OOP testbench. This is more difficult to achieve, as it is not as straightforward.

This paper will first describe how to obtain coverage results from points internal to the testbench and from the pins of the DUT as background and will then focus on how to create covergroups that target points internal to the DUT, without modifying it, and provide the information, without using hierarchical references, to a coverage collector inside of an OVM testbench which allows for coverage driven testing.
No new "magic" trick or tricks are needed to get coverage from inside the DUT. Rather a combination of three things is used; a language construct (SystemVerilog bind), an OOP technique (Abstract/Concrete classes) and an OVM polymorphism capability (factory override) are used. Each of these is not too difficult in and of itself but bringing them together in the right way requires a good understanding all three to address the problem.

1. INTRODUCTION

Verification engineers may use coverage information for defining when to stop testing, for judging the quality of the testing and as feedback for determining where to focus further testing In other words coverage information is used to answer the questions "Are we done testing yet?" and "Have we done adequate testing?". An OVM testbench typically has a Device Under Test (DUT) which is either a Verilog module or a VHDL entity/architecture. The focus of this paper is on how to gather the information for computing coverage information with particular focus on gathering information from inside the DUT. Obtaining coverage information from inside the DUT may be broken down into two parts or problems. 1) How to embed a covergroup inside the DUT without modifying the DUT. 2) How to get access to the covergroup inside the DUT without using hierarchical references.

In this document the example used in diagrams and code is from an OVM testbench which has as its DUT an Ethernet Media Access Controller (MAC) core [1]. The MAC connects to an Ethernet PHY chip through its Media Independent Interface (MII) and to the WISHBONE SoC bus [3]. The MAC registers are memory mapped onto the WISHBONE bus. The WISHBONE bus is a 32 bit address and 32 bit data (non-multiplexed) synchronous bus.

2. Coverage Collectors

A key coverage component is the coverage collector scoreboard. Its role is to collect coverage information and to determine when adequate testing is complete. In an OVM testbench the coverage collector can also fulfill an additional role of halting the simulation when a threshold is reached [2]. The coverage collection is performed by covergroups which may either be instantiated inside the coverage collector or at the source of the coverage information. There are in general three sources of coverage information for a coverage collector:

- Information gathered from inside testbench.
- Information gathered from the DUT pins.
- Information gathered from inside the DUT itself.

Gathering information from inside the testbench will be discussed briefly. Then gathering information from the DUT pins will be discussed in more detail with examples to provide background information for understanding the techniques used for gathering information from inside the DUT itself. Gathering information from inside the DUT will then be discussed.

2.1 Gathering Coverage Information from Inside the Testbench

A covergroup inside of a coverage collector may provide coverage metrics based on information gathered from inside the testbench. To illustrate this type of information consider a WISHBONE bus write to a memory mapped register in the MAC. A WISHBONE write transaction object is created by a stimulus generator and then applied to the WISHBONE bus to write data to the MAC register. A copy of this WISHBONE write transaction object may be presented to the covergroup in the coverage collector. The covergroup may then measure (count) that a certain address was written or that certain data was written or a cross of both the address and data (certain data was written at a certain address). This type of coverage is common in OVM testbenches because the information is easily understood, as it is already "gathered" so to speak in the transaction object and it is easy to present the information to the coverage collector. It should be noted that the data being measured is what may be called "inferred" information because the sampled data is not from the actual address and data lines of the WISHBONE bus. Rather from a

transaction object that if properly applied will drive the specified address and data lines.

2.2 Gathering Coverage Information from the Pins of the DUT

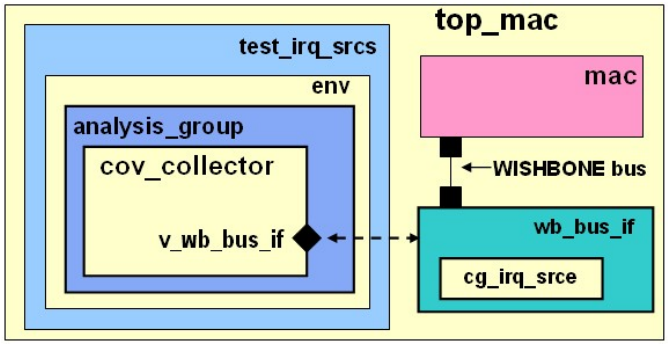
There are two approaches for gathering information at the pins of the DUT for coverage purposes. Using virtual interfaces and using the "Abstract/Concrete classes" approach. Here the information gathered is not "inferred" information such as was described in the previous section. Instead the actual address and data lines of the WISBONE bus are observed as transactions are applied to the bus and then presented to the covergroup.

2.2.1 Using Virtual Interfaces

SystemVerilog provides interfaces for connecting hardware modules (the MAC in our example) without having to do so pin by pin. SystemVerilog also provides virtual interfaces, which are essentially a pointer to an interface instance, for connecting class based testbench components to an interface instance [2]. The diagram in Figure 1 shows the DUT (`mac`) connected to an instance (`wb_bus_if`) of a WISHBONE bus interface and a virtual interface connection (`v_wb_bus_if`) inside of a coverage collector (`cov_collector`) pointing to the `wb_bus_if` instance. The coverage collector resides inside of a container class called the `analysis_group`. `test_irq_srcs` is the top level class object while `top_mac` is the top level module. Note that in this figure the rest of the testbench components and the MII of the MAC are not shown.

In Figure1 the covergroup instance (`cg_irq_srce`) is inside the interface (`wb_bus_if`) and is accessed by the coverage collector through the virtual interface (`v_wb_bus_if`). Alternatively the covergroup could have been placed inside the coverage collector. In this case the information would be provided to the covergroup typically by a monitor or driver component which acts as a gatherer on the coverage collector's behalf.

Figure 1. MAC Testbench - Virtual Interface Connection



2.2.2 Using the Abstract/Concrete Classes Approach

An alternative approach uses what will be referred to as the Abstract/Concrete classes approach. It is used for bridging between the object oriented world of an OVM testbench and the static instance world of the DUT. A use of this approach is described in detail by Rich and Bromley [5].

In the Abstract/Concrete classes approach an abstract class (SystemVerilog virtual class) is defined. The abstract class has pure virtual methods which define a public interface or Application Programming Interface (API) for accessing information that will be needed by the testbench. In this example the class is `irq_srce_base` and the API consists of a method called `get_cg_coverage()` and is for accessing coverage information from the DUT pins. See Figure 2.

Figure 2. Abstract class irq_srce_base and analysis_pkg

```
virtual class irq_srce_base extends ovm_component;
`ovm_component_utils(irq_srce_base)

 function new( string name = "" ,
               ovm_component parent = null);
  super.new( name , parent );
 endfunction
 //event signals change in value may be used as the
 //event sample for a covergroup to check coverage
 event int_result_event;

 // for retrieving coverage of the irq source reg
 pure virtual function int get_cg_coverage();
endclass

package analysis_pkg;
 `include "irq_srce_base.svh"
 // base class handle for making connection to
 // derived class object in interface instance
 irq_srce_base irq_srce_concrete;
 `include "mac_cov_collector.svh"
 // rest of package not shown
endpackage
```

The implementations of the API methods are not in the abstract class but rather are in a derived class which is referred to as the concrete class. The concrete class `irq_source_cov` is defined inside of a SystemVerilog interface, `wishbone_bus_syscon_if` in figure 3. A concrete class handle `irq_srce_concrete` and an allocation method `get_irq_srce()` are also declared inside the interface. The concrete class' scope is inside of the interface and is not visible outside of the interface. In this example an interface is used as the container but a module could be used instead.

Figure 3. Wishbone bus interface containing the irq_srce_cov concrete class

```
interface wishbone_bus_syscon_if
   #(int num_masters = 8, int num_slaves = 8,
     int data_width = 32, int addr_width = 32) ();

import analysis_pkg::irq_srce_base;

 // wishbone slave inputs
 logic [addr_width-1:0]  s_addr;
 bit s_cyc;
 bit s_stb[num_slaves];
 bit s_we;
 // wishbone slave outputs
 logic [data_width-1:0] s_rdata[num_slaves];

// rest of WISHBONE interface logic not shown

//--------------------------------------
// Generate event for coverpoint sample
 wire r_slave_0 = s_cyc & s_stb[0] & !s_we & clk;

 class  irq_srce_concr extends irq_srce_base;
  // Covergroup for the Interrupt Source Register
  covergroup cg_irq_srce @ (posedge r_slave_0);
   int_srce_addr: coverpoint s_addr[11:2]
     // Interrupt Source reg word address is `h1
```

```
    { bins addr_bin = {1};}
  busy: coverpoint s_rdata[0][4] // Busy
    { bins busy_1 = {1};}
  rxe:  coverpoint s_rdata[0][3] // Receive error
    { bins rxe_1 = {1};}
  rxb:  coverpoint s_rdata[0][2] // Receive buffer
    { bins rxb_1 = {1};}
  txe:  coverpoint s_rdata[0][1] // Transmit error
    { bins txe_1 = {1};}
  txb:  coverpoint s_rdata[0][0] // Transmit buffer
    { bins txb_1 = {1};}
  busy_c: cross int_srce_addr, busy;
  rxe_c:  cross int_srce_addr, rxe;
  rxb_c:  cross int_srce_addr, rxb;
  txe_c:  cross int_srce_addr, txe;
  txb_c:  cross int_srce_addr, txb;
 endgroup

 function new(string name = "",
              ovm_component parent = null);
  cg_irq_srce = new();  // create covergroup
  $display("---- My Verilog path is: %m");
 endfunction

 function int get_cg_coverage();
  // get the coverage info from the cover group
  return (cg_irq_srce.get_coverage() );
 endfunction
endclass

// Concrete class handle
irq_srce_concr irq_srce_concrete;

// lazy allocation of concrete class
function irq_srce_base get_irq_srce();
 if(irq_srce_concrete == null)
  irq_srce_concrete = new();
 return (irq_srce_concrete);
endfunction

always @ (negedge r_slave_0)
 // trigger event for sampling
 -> irq_srce_concrete.int_result_event;

endinterface
```

The interface is it self instantiated along side of the DUT and connected to its pins. See the diagram in Figure 4 and the code in Figure 5.

Figure 4. MAC testbench - Abstract/Concrete class connection

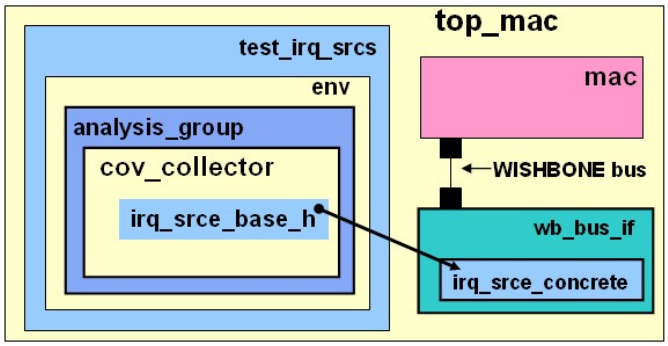


Figure 5. WISHBONE interface instantiation and connection

```
module top_mac;
  import ovm_pkg::*;
  import tests_pkg::*;
  import analysis_pkg::irq_srce_base;

  // Wishbone interface instance
  wishbone_bus_syscon_if wb_bus_if();
  // MAC instance
   eth_top mac
  (
    // WISHBONE common
    .wb_clk_i( wb_bus_if.clk ),
    .wb_rst_i( wb_bus_if.rst ),
    // WISHBONE slave
    .wb_adr_i( wb_bus_if.s_addr[11:2] ),
    .wb_we_i ( wb_bus_if.s_we  ),
    .wb_cyc_i( wb_bus_if.s_cyc ),
    .wb_stb_i( wb_bus_if.s_stb[0] ),
    .wb_dat_o( wb_bus_if.s_rdata[0] ),
  // other MAC port connections not shown
  );

initial begin
  // Assign class handle in analysis_pkg
  analysis_pkg::irq_srce_concrete =
              wb_bus_if.get_irq_srce();
  // rest of module not shown
endmodule
```

This instance of the interface resides in the static instance world and is tightly coupled to the DUT. Since the concrete class definition and instance are inside this interface it can access everything that the interface is connected to.

Inside of a coverage collector (`mac_cov_collector`) a variable (`irq_srce_cov`) of the abstract class type (a base class handle) is created which may then hold a reference to, or point to the instance of the concrete class (derived class object) inside of the interface. See Figure 4. This provides access for the covergroup to DUT pin information by calling the API methods on the abstract class variable. See figure 6.

Figure 6. Coverage collector

```
class mac_cov_collector extends ovm_component;
 `ovm_component_utils(mac_cov_collector)

// Abstract class handle
 irq_srce_base  irq_srce_base_h;

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

 function void build();
  super.build();
  // get handle to object in interface from
  // handle in analysis package
  irq_srce_base_h =
        analysis_pkg::irq_srce_concrete;
 endfunction

 // check coverages and if 100% stop the tests
 task run();
    //look for event to signal change irq srce reg
  forever @(irq_srce_base_h.int_result_event) begin
   ovm_report_info("MAC_COV", $psprintf(
    "MAC interrupt source reg coverage is %.2f%%",
    irq_srce_base_h.get_cg_coverage() ));
```

```
    if(irq_srce_base_h.get_cg_coverage() == 100)
      ovm_top.stop_request();  //100% coverage - done
  end
 endtask
endclass
```

In an OVM testbench there are several ways that the abstract class handle inside the coverage collector may be assigned to the instance of the concrete class. In this example a global variable is created inside of the package `analysis_pkg` (see Figure 2) which is assigned in the top module (see Figure 5) to point to the interface instance. The abstract class handle is assigned from this global variable (see Figure 6). Another way this assignment could be accomplished is with the OVM configuration database. Passing the location of the interface instance through constructor arguments would not be allowed here because of the use of the OVM factory for creation of objects in this example.

2.3 Gathering Coverage Information from the inside the DUT

It is often desirable or necessary to not only gather information at the pins of the DUT but from inside the DUT as well. Verilog permits hierarchical references to the inside of the DUT. In the crudest case a hard coded hierarchical path might be used from within the coverage collector to access the information. Alternatively either of the two methods described previously for obtaining information from the DUT pins might be augmented to include hierarchical paths to access inside the DUT. None of these approaches, however, is very desirable. If the DUT is written in VHDL then hierarchical references for accessing internal information are not allowed. Another solution is to add a covergroup inside the DUT by modifying the DUT. While this may be feasible in some cases often it is not. And even if feasible it is rarely desirable to modify the DUT.

Even if a covergroup may be added inside the DUT there is still the issue of getting the information out without using a hierarchical reference.

3. Adding a Covergroup to the DUT without modifying it – SystemVerilog bind

As mentioned in the introduction the first part or problem to getting coverage information from inside the DUT is embedding a covergroup inside the DUT without modifying the DUT. The use of the SystemVerilog bind statement to embed assertions inside a DUT is common and well understood and is described in detail by Cummings[6]. Similarly the bind statement may be used to embed a covergroup inside the DUT. A summary only of the bind statement is provided here. Consider an example interface that has a covergroup defined and instantiated inside of it. Additionally the interface has ports which connect to the variables that the covergroup will access. The bind statement is then used to "bind" the interface to either a specific sub-module instance or all the instances of a sub-module of the DUT. The result of the bind is as if an instance of the interface was created inside of the sub-module to which it is bound, a "remote instantiation" if you will. One may verify this by displaying the "bound" interface's hierarchical path using this Verilog print statement placed inside the interface:

```
$display("My Verilog path is: %m");
```

In the bind statement the connection of the ports of the interface is specified. Using the SystemVerilog bind statement a covergroup inside of a bound interface can access the required information inside the DUT without modification to the DUT and without using hierarchical references.

4. Accessing the covergroup information inside the DUT

The first part or problem is solved using the bind statement to add the covergroup to the DUT without modifying the DUT. The second part or problem is to access or retrieve the covergroup information from the bind instance inside the DUT *without using hierarchical references*. In the example illustrated in the next sections the target of the coverage is the Interrupt Source Register of the MAC which is implemented in the `eth_registers` sub-module of the MAC design. Internally this register is called `INT_SOURCEOut`.

4.1 Accessing the covergroup Information inside the DUT using a Virtual Interface

As described in section 2.2.1 a virtual interface may be used to point to an instance of an interface for accessing information that the interface is connected to. It is required that the virtual interface variable be assigned the location of the instance of the interface. In the case of a bind of the interface to a DUT sub-module the location of the interface instance is not the location of the bind statement itself, but rather as shown in section 3 it is *inside* of the DUT. Thus a hierarchical path to the interface instance inside the DUT must be assigned to the virtual interface property inside of the testbench. Obtaining and using this hierarchical path is not feasible with a VHDL DUT and may not be feasible with a Verilog DUT. Consequently it is concluded that using a virtual interface *is not* a solution to accessing the covergroup information inside the DUT.

4.2 Accessing the covergroup Information inside the DUT using the Abstract/Concrete classes Approach

As described in section 2.2.2 the Abstract/Concrete classes approach may also be used for accessing information from the DUT to which the interface containing the concrete class is connected to. In the Abstract/Concrete classes approach an assignment of the concrete class object inside the interface instance is made to the abstract class handle inside the coverage collector OVM testbench component. This requires the location of the interface instance be known. Consequently it is concluded that using the Abstract/Concrete classes approach *alone* is not a solution to accessing the covergroup information inside the DUT. This approach alone has exactly the same problem described in the previous section with using virtual interfaces,

5. OVM Technology that is part of the solution

The proposed solution to getting the covergroup information out of the DUT without using hierarchical references requires use of the Abstract/Concrete classes approach together with the use of OVM factory overrides. Additionally understanding the way OVM maintains OVM testbench hierarchies with its own path names while not part of the solution can help clarify how the solution works.

5.1 OVM Factory and Factory Overrides

OVM has a singleton object called the "factory". It implements the object-oriented design pattern known as the *factory method* pattern [4]. In general the term *factory method* means a method whose

purpose is the creation of objects. This creation pattern has to deal with the problem of creating objects without specifying the exact type of object to be created. It does so by providing a method for creating an object (`create`) that may be overridden (factory override) to specify a derived type object.
The OVM factory is used for dynamic or run time object creation. These objects may be transaction objects such as WISHBONE transactions or Ethernet transactions, or may be testbench components such as a coverage collector or driver. Objects that are to be created using the factory must be registered with the factory prior to the start of simulation. Thus the factory can create any of the registered object types by calling it's `create()` method. Furthermore, in keeping with the factory method pattern, the OVM factory has an override mechanism which may be best described with an example. Given that types A and B are both registered with the factory *and* type B is derived from type A, then type B can be set to override type A. Once this override is in place a request to create an object of type A will result in the creation of object type B in its place. The OVM factory provides for overriding by type or by type name. In our example an override by type name is used. It should be noted that although the OVM factory is used in this solution outside of an OVM testbench the same approach could be implemented using a hash table of the abstract class type.

5.2 OVM Hierarchical Path

OVM maintains its own path information for each testbench component that is separate and different from the underlying SystemVerilog hierarchical path. That is each object in an OVM testbench has an "OVM path" which is used by OVM and a "SystemVerilog path" which is used by the SystemVerilog compiler. Each testbench component, when it is created is given an OVM path name which is unique. It is similar to the SystemVerilog path in that the path it is constructed of names separated by the period (.) character. Additionally each testbench component knows who its hierarchical parent is in the OVM structure and who its hierarchical children are. OVM uses the OVM path information for things like traversing the testbench structure to execute phase methods, for locating objects within the testbench and so forth. A testbench component must reside within the OVM hierarchical structure to function properly within the testbench.

6. Solution Using Factory Overrides with Abstract/Concrete Classes Approach and bind

A solution example will be presented in this section which uses the SystemVerilog bind statement to solve the first problem of embedding a covergroup inside the DUT with out modifying the DUT and uses the Abstract/Concrete classes approach together with the OVM factory override to solve the second problem of accessing the covergroup information inside the DUT without using hierarchical references.

6.1 Solving the First Problem using Bind

An interface `mac_regs_cov_container` is created to contain the declaration of the concrete class `irq_srce_concr`. See Figure 7. Note that since a virtual interface connection is not used in this approach a module could be used instead of an interface as the container.
Inside the concrete class `irq_srce_concr` the `cg_interrupt_source` covergroup covers the Interrupt Source Register (`INT_SOURCEOut`) of the MAC. The `INT_SOURCEOut` is inside the eth_registers sub-module of the MAC.

Figure 7. Interface Container for the Concrete Class

```
`include "ovm_macros.svh"
interface mac_regs_cov_container(
 input Clk,                        // clock
 input wire [31:0] INT_SOURCEOut // irq srce reg
 );
 import ovm_pkg::*;
 import analysis_pkg::irq_srce_base;

 class  irq_srce_concr extends irq_srce_base;
  // OVM factory registration
  `ovm_component_utils(irq_srce_concr)

  // Covergroup for the Interrupt Source Register
  covergroup cg_interrupt_source;
   busy: coverpoint INT_SOURCEOut[4]//Busy
     { bins busy_1 = {1};}
   rxe:  coverpoint INT_SOURCEOut[3]//Receive error
     { bins rxe_1 = {1};}
   rxb:  coverpoint INT_SOURCEOut[2]//Receive buffer
     { bins rxb_1 = {1};}
   txe:  coverpoint INT_SOURCEOut[1]//Transmit error
     { bins txe_1 = {1};}
   txb:  coverpoint INT_SOURCEOut[0]//Transmit buff
     { bins txb_1 = {1};}
  endgroup

  function new(string name, ovm_component parent);
   super.new(name,parent);
   cg_interrupt_source = new(); //create covergroup
  endfunction

  function int get_cg_coverage();
   // get the coverage info from  cover group
   return (cg_interrupt_source.get_coverage() );
  endfunction

  task run();
   ovm_report_info("IRQ_SRCE_CONCR", $psprintf(
     "My OVM path name is: %s", get_full_name()));
   ovm_report_info("IRQ_SRCE_CONCR",
      $psprintf("My Verilog path is: %m"));
   forever @ (INT_SOURCEOut[4:0]) begin
    cg_interrupt_source.sample();//sample covergroup
    -> int_result_event;         //trigger event
   end
  endtask
 endclass
endinterface
```

Figure 8 shows the bind instance `mac_regs_bind` of the interface `mac_regs_cov_container` to the `eth_registers` sub-module.

Figure 8. bind instance

```
module top_mac;
  bind eth_registers mac_regs_cov_container
       mac_regs_bind(.*);
 // rest of top_mac not shown
endmodule
```

6.2 Solving the Second Problem using Abstract/Concrete classes and OVM Factory Override

In order to use the Abstract/Concrete classes approach the creation of the concrete class `irq_srce_concr` must be done using a factory override instead of by explicitly calling `new()`. As noted in section 5.1 this requires that both the abstract and the concrete classes must be registered with the factory. This factory registration, done with the macro, can be seen in Figure 7 for the concrete class and in Figure 8 for the abstract class.

Figure 8. Abstract class irq_srce_base

```
virtual class irq_srce_base extends ovm_component;
 // OVM factory registration
`ovm_component_utils(irq_srce_base)

 function new( string name = "" ,
               ovm_component parent = null);
  super.new( name , parent );
 endfunction
 //event signals change in value may be used as the
 //event sample for a covergroup to check coverage
 event int_result_event;

 // for retrieving coverage of the irq source reg
 pure virtual function int get_cg_coverage();
endclass
```

The factory override of the abstract class with the concrete class in this example is done in the top module `top_mac`. See figure 9.

Figure 9. Factory Override

```
module top_mac;
initial
  // Factory overrides
  factory.set_type_override_by_name(
      "irq_srce_base", "irq_srce_concr");
 // rest of module not shown
endmodule
```

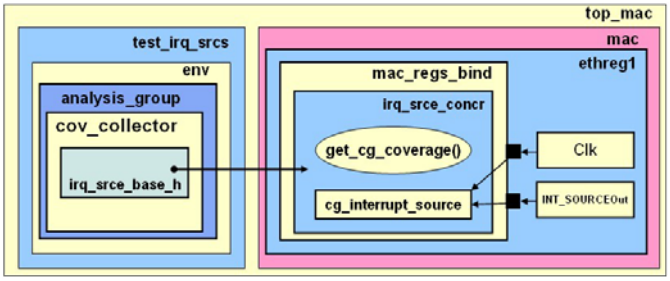
In the coverage collector (See Figure 10), the `create()` method of the abstract class is called and the resultant object is assigned to the abstract class handle `irq_srce_base_h`. Because of the factory override in `top_mac` (Figure 9) the resultant object t is a concrete class object instead of the abstract class object.

Figure 10. Coverage collector

```
class mac_cov_collector extends ovm_component;
 `ovm_component_utils(mac_cov_collector)

 // components
irq_srce_base  irq_srce_base_h;

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

 function void build();
  super.build();
  // create base coverage container
  // This is meant to be overridden!
  irq_srce_base_h =
   irq_srce_base::type_id::create("irq_srce_base_h",
this);
 endfunction

 task run();  // check coverages and if 100% stop
the tests
  //look for event to signal change in irq srce reg
  forever @ (irq_srce_base_h.int_result_event) begin
   ovm_report_info("MAC_COV", $psprintf(
    "MAC interrupt source register coverage is
%.2f%%",
     irq_srce_base_h.get_cg_coverage() ));
   if(irq_srce_base_h.get_cg_coverage() == 100)
     ovm_top.stop_request();  //100% coverage - done
  end
 endtask
endclass // rest of module not shown
```

The diagram in Figure 11 shows the resultant "connection" between the coverage collector and the bind instance inside the DUT. The abstract base handle `irq_srce_base_h` points to the concrete object `irq_srce_concr`, the scope of which is inside the bind instance `mac_regs_bind`.

Figure 11. MAC testbench. Bind, Abstract/Concrete classes, Factory Override



The coverage collector can access information from the covergroup that is inside of the DUT! How? Because, in effect, the concrete class object (`irq_srce_concr`) that is created "exists" in both the static instance world of the DUT and the object oriented world of an OVM testbench at the same time. You can think of it as having "dual citizenship" in both these worlds.

How does it "exist" in both?

The concrete class object "exists" inside the DUT. Because it is defined or declared inside of the interface (`mac_regs_cov_container`), which is bound to a sub-module of the DUT, the scope of the created object is inside of the interface bind instance inside the DUT. In other words the SystemVerilog path of the object shows it inside the interface bind instance inside the DUT. As discussed earlier, this can be demonstrated by executing this statement inside the concrete class:

`$display("My Verilog path is: %m");`

Consequently the concrete class object can access DUT information there.

The concrete object "exists" inside the OVM testbench. The request to create the abstract class is executed in the coverage collector, which is an OVM testbench component. Because of the override, the concrete class object is created instead and because this object is an OVM testbench component and because the call to create it is inside of another OVM testbench component, the concrete class object is created as part of the OVM testbench structure. Its OVM path shows that it is a hierarchical child object to the coverage collector. This can be shown by executing the following statement inside the concrete class object:

`$display("My OVM path name is: %s",`

```
            get_full_name());
```

The coverage collector may call the concrete class object's API method `get_cg_coverage()` (See Figure 10) for accessing the coverage information of the covergroup.

The bind statement does the "remote instantiation" of the covergroup inside the DUT (without DUT modification) and the creation of the concrete object via the factory override "pulls" the scope or visibility of the covergroup into the OVM testbench. This "dual citizenship" of the concrete object allows the coverage collector to access the covergroup information from the bind instance inside the DUT via API calls.

7. Conclusions

- Coverage information for coverage driven testing may be gathered from all three desired locations. From within the testbench, from the DUT pins and from inside the DUT itself.
- The two approaches (virtual interfaces and Abstract/Concrete classes) for gathering information from the DUT pins are inadequate in and of themselves for getting coverage information out of the DUT without DUT modification or the use of hierarchical references.
- The bind construct may be used to place a covergroup inside the DUT without modifying it.
- The Abstract/Concrete classes approach combined with the override capability of the OVM factory "pulls" the scope (visibility) of the covergroup instance inside the DUT into the OVM testbench without using hierarchical references.

8. REFERENCES

[1] I. Mahor, Ethernet IP Core Design Document. 2002 http://www.opencores.org/project,ethmac

[2] M. Glasser. Open Verification Methodology Cookbook. 2009, Springer, Inc.

[3] WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores specification. Revision B.3, 2002. http://www.opencores.org

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software. 1995, Addison-Wesley Publishing Company, Reading Massachusetts,

[5] D. Rich, J. Bromley. Abstract BFMs outshine Virtual Interfaces for Advanced SystemVerilog Testbenches. DVCon February 2008

[6] C. Cummings. SystemVerilog Assertions Design Tricks and SVA Bind Files. SNUG SJ 2009. www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf