

Using `bind` for Class-based Testbench Reuse with Mixed-Language Designs

Doug Smith

Doulos
Morgan Hill, California, USA

doug.smith@doulos.com

ABSTRACT

Significant effort goes into building block-level class-based testbenches so reusing them in a full-chip environment has great advantages. The problem arises of which way is best to connect them into a full-chip environment that maximizes reusability. VHDL or mixed-language designs pose greater challenges because hierarchical references are unsupported in VHDL. Alternatively, SystemVerilog offers a simple solution with the `bind` command. Using `bind` and a few simple guidelines, a block-level testbench can be reused without modifications in a mixed-language full-chip environment. This paper demonstrates how to structure a testbench for effortless reuse with nothing more than a single `bind` command.

Table of Contents

1	INTRODUCTION	5
2	CLASS-BASED TESTBENCH REUSE.....	5
2.1.	VIRTUAL INTERFACES	7
3	BLOCK-LEVEL TESTBENCH REUSE	8
3.1.	HIERARCHICAL REFERENCES.....	9
3.2.	CHALLENGE OF MIXED-LANGUAGE DESIGNS	10
4	USING <code>BIND</code>	10
4.1.	TESTBENCH STRUCTURE.....	11
4.1.1.	<i>Binding signals to ports</i>	12
4.1.2.	<i>Connecting the ports to the interface</i>	13
4.1.3.	<i>Port directions</i>	14
4.2.	CONDITIONAL COMPILATION.....	16
4.3.	CONFIGURABLE VERIFICATION COMPONENTS	17
5	REUSE IN FULL-CHIP TESTCASES	19
5.1.	OVM ENVIRONMENTS	19
5.2.	VMM AND OTHER ENVIRONMENTS.....	20
5.3.	VMM AND OVM INTEROPERABILITY.....	22
6	PUTTING IT ALL TOGETHER.....	23
7	CONCLUSIONS	25
8	ACKNOWLEDGEMENTS.....	26
9	REFERENCES	26

Table of Figures

Figure 1. Typical class-based testbench.....	6
Figure 2. A typical class-based testbench consisting of a top-level module, interface, DUT, and testbench objects.....	7
Figure 3. An example use of a virtual interface passed into the testbench constructor.	8
Figure 4. Block-level testbench reused in a full-chip environment.	8
Figure 5. Use of <code>assign</code> statements to connect the block-level testbench to the design unit in the full-chip testbench.	9
Figure 6. Signal aliasing using clocking blocks.	10
Figure 7. Binding the block-level testbench inside the full-chip design.	11
Figure 8. Accessing bound interface in full-chip environment from within the block-level testbench.	11
Figure 9. Binding the block-level testbench into the full-chip design.	11
Figure 10. Port connections specified in the <code>bind</code> statement.	12
Figure 11. Design ports added to block-level testbench.	12
Figure 12. Named mapping of ports in the <code>bind</code> statement.....	12
Figure 13. Diagram of the block-level testbench bound inside the full-chip design.....	13
Figure 14. Automatic port mapping of the block-level testbench ports to the interface.	13
Figure 15. Named port mapping of block-level testbench ports to the interface.	13
Figure 16. Interface example using multiple clocking blocks and <code>inouts</code> for signals no longer driven by the block-level testbench drivers.....	15
Figure 17. Local variables initialized to high impedance in order to avoid driver conflicts.....	16
Figure 18. Generate statements used to conditionally instantiate the design and glue logic.	16
Figure 19. <code>Bind</code> statement using a parameter to conditionally instantiate components.	17
Figure 20. Parameterized class-based testbench component.....	17
Figure 21. Class parameter passed to various active testbench components.	18
Figure 22. Top module parameter passed into parameterized classes.	18
Figure 23. Use of an active flag instead of a parameterized class to disable testbench components.....	19
Figure 24. OVM example using the factory to configure a block-level testbench sequencer.	20
Figure 25. Global package variable provides access to block-level testbench wherever it is bound into the full-chip environment.	20
Figure 26. Global static class member provides access to block-level testbench wherever it is bound into the full-chip environment.	21
Figure 27. Diagram illustrating how a global package variable can be used to reference a testbench wherever it instantiated (bound).....	21
Figure 28. OVM example accessing a VMM environment.	22
Figure 29. Block-level testbench top module template.	23
Figure 30. Testbench to DUT interface template.	24
Figure 31. Block-level class-based environment template.....	25
Figure 32. Full-chip <code>bind</code> statement template.	25

Table of Tables

Table 1. Port directions for testbench components.	14
---------------------------------------------------------	----

1 Introduction

Full-chip verification environments rely on reuse of components and block-level¹ environments to reduce overall effort and increase the effective verification checking and coverage. While the reuse of individual components has significant benefits, the reuse of entire block-level environments affords the greatest benefits since substantial integration effort and maintenance can be avoided. Traditional approaches to reusing block-level environments typically involve using proprietary signal access routines or hierarchical references, which only work in Verilog-based designs or testbench environments. VHDL or mixed-language designs pose particular challenges to integrating block-level environments since hierarchical references are unsupported.

However, SystemVerilog[1] offers a simple, yet elegant, solution for reuse with any kind of design. This paper presents a simple reuse methodology that is compatible with any custom or industry standard class-based verification methodology (like VMM) and allows entire block-level testbench reuse in a full-chip with homogeneous or mixed-language designs. Section 2 will begin by discussing how class-based environments are typically constructed in SystemVerilog and the use of virtual interfaces to provide reusability. Section 3 discusses the challenges of integrating a block-level testbench into an environment containing VHDL components and how the SystemVerilog `bind` command can work-around those difficulties. Section 4 then presents a simple methodology consisting of three basic guidelines to follow in order to make a block-level environment reusable in a full-chip environment using only one `bind` statement. The motivation for these guidelines is discussed as well as code examples and recommendations on using the `bind` statement are given. Section 5 addresses how full-chip testcases can control and configure the block-level environment components although the environments may be instantiated deep within a mixed-language design. Likewise, this section details examples of using this methodology with industry standards like VMM or OVM, and even how to seamlessly mix VMM and OVM testbenches together in the same environment. Lastly, section 6 distills the three guidelines into easy-to-follow templates for a block-level testbench, and section 7 concludes the paper with a few remarks on tool issues to recognize.

2 Class-based testbench reuse

A class-based testbench has significant reuse advantages over a traditional module-based testbench because it offers a more flexible, dynamic, extensible, and configurable environment. Object-oriented programming (OOP) enables testbench elements to be encapsulated into simple components that can be reused throughout your testbench or between multiple environments. It also provides the flexibility to dynamically construct your testbench environment at run-time, and structure the environment differently based on testcase requirements.

Since object creation only occurs at run-time, all testbench environments require some structural stub such as a module or program to dynamically create the class-based environment. A simple initial block will suffice to create the class-based environment, but the design and interface also need to be instantiated.

¹ The term “block” used in this paper refers to the smallest divisible functional design unit typically treated as an independent project with its own sophisticated verification environment.

Connecting the design with the testbench is accomplished by passing a special reference to the interface, called a *virtual interface*, to the constructor of the class-based environment.

Thus, a typical class-based environment consists of a top-level module or program that (1) instantiates the design, (2) instantiates a SystemVerilog interface, (3) declares and allocates the testbench object, and (4) passes a reference to the interface into the environment. Such an environment is shown in the following figures.

```
module block_tb;

    tb_env tb;                // (3) tb_env class instance variable

    apb_intf apb_bus();       // (2) Instantiate the physical interface

    design dut (               // (1) Instantiate the design and connect signals
        .clk    (apb_bus.clk),
        .psel   (apb_bus.psel),
        .penable(apb_bus.penable),
        ...
    );

    // Now create the class-based testbench
    initial begin
        tb = new ( apb_bus, ... ); // (3)/(4) Pass reference to the interface
                                   // called a "virtual interface"
        ...
    end

endmodule : block_tb
```

Figure 1. Typical class-based testbench.

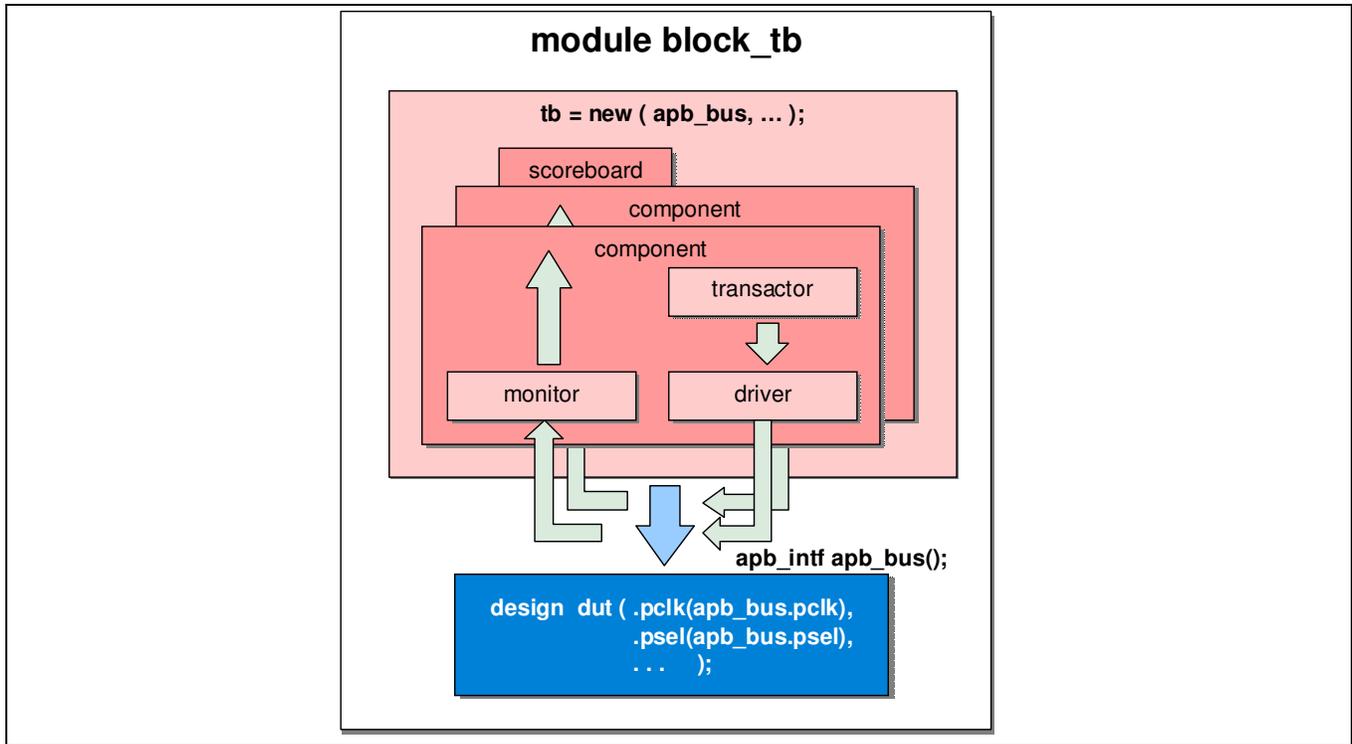


Figure 2. A typical class-based testbench consisting of a top-level module, interface, DUT, and testbench objects.

2.1. Virtual interfaces

Since virtual interfaces are references to an actual interface, they provide an excellent way of connecting a class-based SystemVerilog testbench together with the design-under-test. A testbench refers to the design signals without knowledge of the connections between it and the design. Therefore, interfaces that map signals differently, include different assertions or coverage, or include additional logic like error injection can be easily interchanged as long as they provide the same signal names to the testbench. Consequently, testbenches using virtual interfaces can be constructed to work easily from one environment or project to the next.

A virtual interface is declared by using the keyword `virtual` before an interface declaration inside of the testbench classes as shown in Figure 3. This indicates that only a reference to the interface will be passed into the `tb_env` object. When the testbench is created in the `initial` block by the call to `new()` (see Figure 1), `apb_bus` is passed to the testbench's constructor and the reference is saved locally so that it can be distributed to the various testbench components that interact with the design such as drivers and monitors.

```

class tb_env;
  virtual apb_intf apb_bus; // Variable to hold reference to the interface

  function new ( virtual apb_intf _bus );
    apb_bus = _bus;
  endfunction : new

  ...
endclass : tb_env

```

Figure 3. An example use of a virtual interface passed into the testbench constructor.

3 Block-level testbench reuse

Verification projects typically consist of several block-level testbenches to verify individual design elements, and a full-chip testbench environment to test both functionality and integration. The protocol checking, coverage, and assertions monitored at the block-level are particularly useful in a full-chip environment. So having the ability to readily reuse these components without significant effort and modifications can offer great advantages in finding bugs in the top-level testbench.

Ideally, we want to reuse the entire block-level testbench unmodified in order to maximize the benefits of reuse; however, this is not always so straightforward. In fact, once a design module is integrated into a full-chip design, several things in the block-level testbench become broken in a top-level environment. For example, the block-level design no longer needs to be instantiated in the block-level testbench because it is part of the full-chip design (see Figure 4).

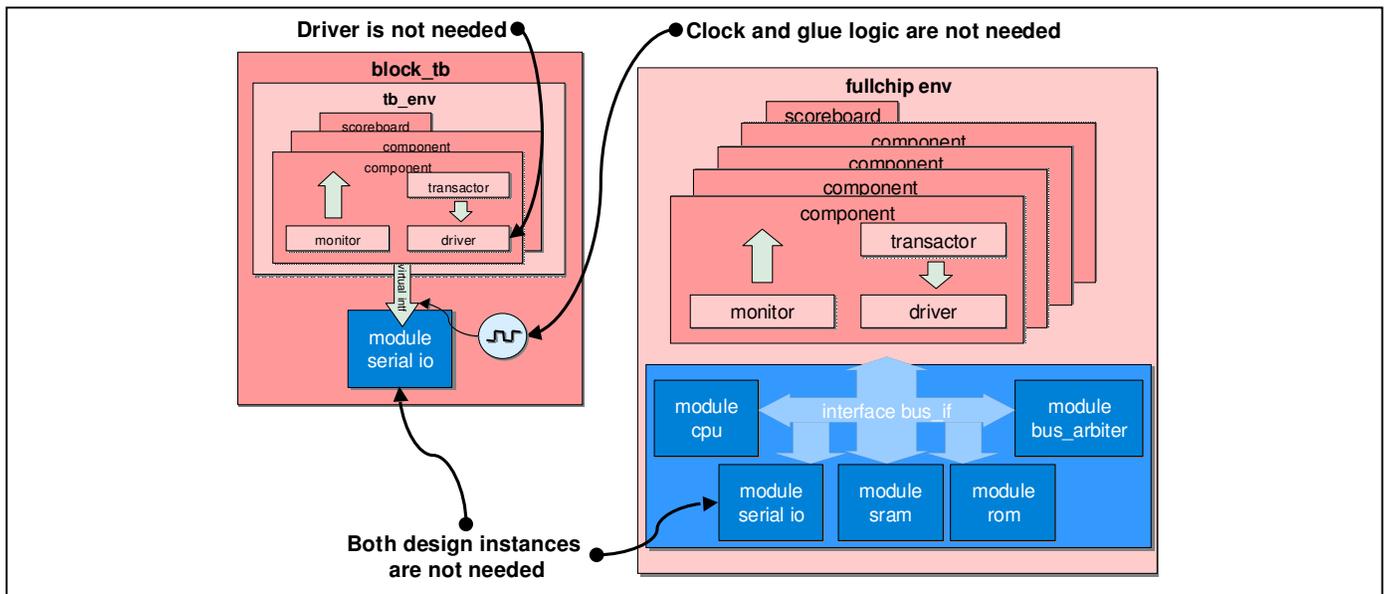


Figure 4. Block-level testbench reused in a full-chip environment.

Consequently, the interface signals of the block-level design unit need to be accessed in a different way by the testbench (usually by hierarchical references), and active transactor/driver components are unnecessary since input stimulus will be driven from other parts of the full-chip design. Glue logic as well (such as clock and reset generation logic) is unnecessary and the design clock needs to be used in the block-level testbench.

So in order to easily reuse a block-level testbench in a full-chip, a block-level testbench must meet the following requirements:

1. **Correctly structured:** it must be structured to access the design signals in either a block or full-chip environment.
2. **Conditional compilation:** it must provide conditional instantiation of the design and supporting logic (like clock and reset generation).
3. **Configurable:** it must be configurable such that the active transactor(s)/driver(s) can be disabled in a full-chip environment.

If a block-level testbench follows these three simple guidelines, then it can be reused without modifications in a full-chip environment.

3.1. Hierarchical references

The first step to reuse is to structure the block-level testbench so that it can access design signals in any kind of environment. The easiest way access signals is to use hierarchical references. Since Verilog allows for multiple top modules (i.e., non-instantiated modules), both the full-chip and block-level modules can simulate in parallel together as top modules. Accessing the design's ports involves just a reference through the full-chip's hierarchy. For example, we could use `assign` statements and poke down through the hierarchy to the design as in Figure 5.

```
module block_tb;

    apb_intf apb_bus();    // Instantiate the physical interface

    assign apb_bus.clk     = toptb.fullchip.design_unit.clk;
    assign apb_bus.psel    = toptb.fullchip.design_unit.psel;
    assign apb_bus.penable = toptb.fullchip.design_unit.penable;

    ...

endmodule : block_tb
```

Figure 5. Use of `assign` statements to connect the block-level testbench to the design unit in the full-chip testbench.

A much better approach is to place the hierarchical references in the interface, which can be easily accomplished using a clocking block structure shown in Figure 6.

```
interface apb_intf();

    logic clk;
    logic psel;
    logic penable;

    clocking cb @(posedge clk);
        input psel    = toptb.fullchip.design_unit.psel; // Specify path reference
        input penable = toptb.fullchip.design_unit.penable;
        input pwrite  = toptb.fullchip.design_unit.pwrite;
        ...
    endclocking

    modport TB (clock cb);

endinterface : apb_intf
```

Figure 6. Signal aliasing using clocking blocks.

Using references in the clocking block means that the testbench can now be used anywhere without modifications since only the interface changes. In general, this approach is commonly used; however, EDA tool support for references in clocking blocks may vary, and hierarchical references are tedious to maintain, typically not portable, and only work in Verilog-based designs.

3.2. Challenge of mixed-language designs

VHDL offers no support for references so for VHDL designs hierarchical references simply are not an option. However, sometimes a mixture of Verilog and VHDL are required for a design. Within a Verilog block, hierarchical references work as usual, but if the Verilog block is wrapped within a VHDL architecture then the references can no longer probe down through the design. Such is often the case in SoC designs where many IPs and legacy blocks are integrated together. Therefore, alternative solutions must be devised.

4 Using `bind`

In order to work around this limitation of VHDL, engineers typically resort to one of the following two methods: either (1) using signal access routines (either proprietary or VHPI), or (2) the SystemVerilog `bind` statement. Signal access routines vary in support across simulators and often end up being complicated and proprietary, which limits portability of your verification environment across EDA tools. A much better approach is using the SystemVerilog `bind` command. `bind` allows you to instantiate a module (the *target* module) inside any other module (the *destination* module), whether it is Verilog or VHDL, without modification to the source code of the destination module. This approach is often used to attach SystemVerilog assertions to a design without embedding them in the RTL code. Likewise, the use of

`bind` is more simulation efficient since hierarchical references limit the amount of optimizations a simulator compiler can perform.

The simplest approach to using the `bind` statement is to bind the testbench interface directly inside the full-chip design:

```
bind fullchip apb_intf apb_bus( ... );
```

Figure 7. Binding the block-level testbench inside the full-chip design.

Now, the block-level testbench only needs a reference to the interface passed into its constructor:

```
module block_tb;
    tb_env tb;                // tb_env class instance variable

    // apb_intf apb_bus();    // Not used in the full-chip environment

    // Now create the class-based testbench
    initial begin
        tb = new ( toptb.fullchip.apbunit.apb_bus, ... );
        ...
    end
endmodule : block_tb
```

Figure 8. Accessing bound interface in full-chip environment from within the block-level testbench.

However, this approach suffers from two issues: (1) the top module of the block-level testbench needs to be modified in order to remove the interface, and (2) a hierarchical reference is used to reference the bound interface, which again fails to work with VHDL or mixed-language designs.

4.1. Testbench structure

The solution to access signals is to bind the entire block-level testbench into the full-chip design. Notice, whenever a target module is bound, it gets instantiated inside the destination module. The block-level testbench gains access to the design signals inside the full-chip design without the use of hierarchical references.

```
bind fullchip block_tb tb_inst ( );    // Bind the block-level testbench inside
                                         // the enclosing fullchip design along
                                         // with the block-level design unit
```

Figure 9. Binding the block-level testbench into the full-chip design.

4.1.1. Binding signals to ports

Since the block-level testbench is instantiated directly inside the design, hierarchical paths are no longer needed to access the signals. However, none of the design signals actually attach to the block-level testbench since none were specified in the `bind` statement. Instead, the `bind` statement needs the port connections specified, which is most easily accomplished using the port connection shorthand:

```
bind fullchip block_tb tb_inst ( .* );
```

Figure 10. Port connections specified in the `bind` statement.

Again, another problem arises because `.*` only connects to *declared* port connections, but the block-level testbench lacks ports since testbenches do not use ports. So to allow the `bind` statement to bind into a design and provide visibility to the design signals, all the design unit signals need to be mirrored in the block-level testbench's top-level module:

```
module block_tb (
    input clk, output psel, penable, pwrite, ...,
    input [`WIDTH:0] prdata
);
    tb_env tb;      // tb_env class instance variable

    apb_intf apb_bus(); // Instantiate the physical interface

    design dut ( ... );

    // Now create the class-based testbench
    initial begin
        tb = new ( apb_bus, ... ); // Virtual interface
        ...
    end

endmodule : block_tb
```

Figure 11. Design ports added to block-level testbench.

If the signal names differ inside the full-chip design, then the individual signals can be directly mapped using the `bind` statement. Figure 13 illustrates this binding inside the full-chip environment.

```
bind fullchip block_tb tb_inst ( .pclk   ( sysclk ),
                                .psel   ( sel_a  ),
                                .penable( ena    ),
                                ... );
```

Figure 12. Named mapping of ports in the `bind` statement.

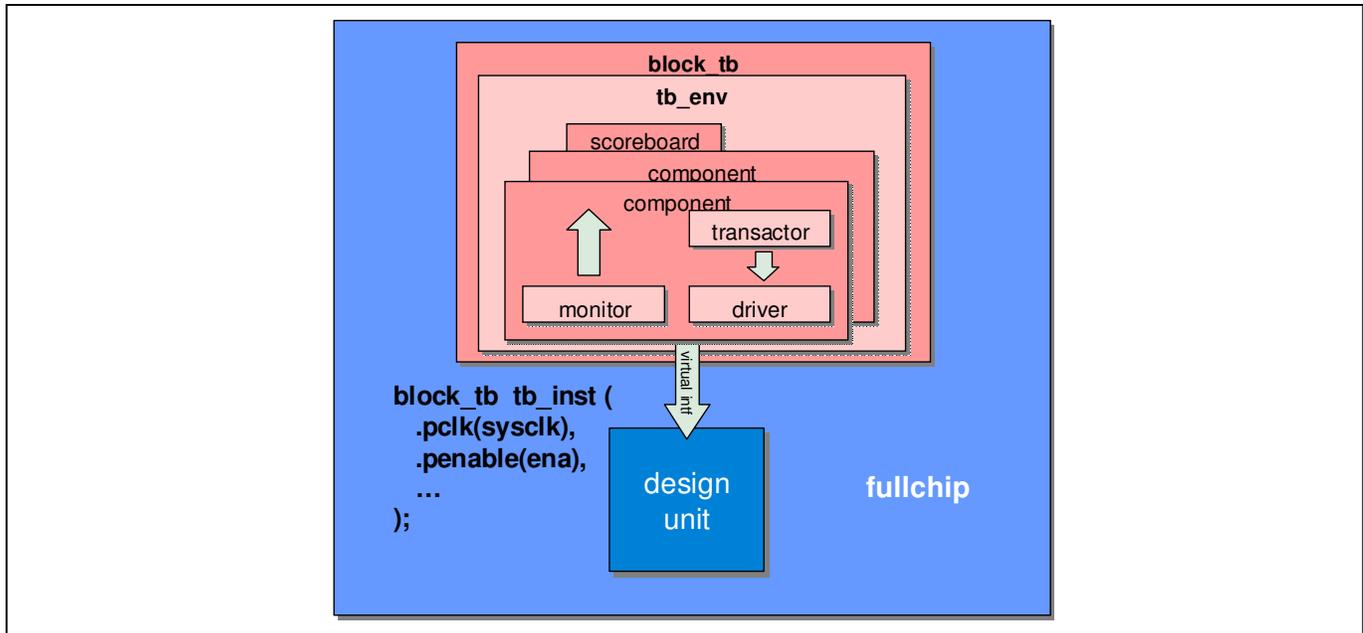


Figure 13. Diagram of the block-level testbench bound inside the full-chip design.

4.1.2. Connecting the ports to the interface

Once the design signals are visible inside the block-level testbench, the ports need to be connected into the interface so the class-based testbench also has access to them through the virtual interface. Continuous `assign` statements can be used, but an easier approach is to use automatic port mapping on the interface:

```

module block_tb (
  input clk, output psel, penable, pwrite, ...,
  input [`WIDTH:0] prdata
);
  tb_env tb; // tb_env class instance variable

  apb_intf apb_bus( .* ); // Automated port mapping
  ...

```

Figure 14. Automatic port mapping of the block-level testbench ports to the interface.

or direct signal name mapping:

```

apb_intf apb_bus( .clk(clk), .psel(psel), .pwrite(pwrite), ... );

```

Figure 15. Named port mapping of block-level testbench ports to the interface.

Now, the `apb_bus` interface is passed into the class-based testbench environment and the testbench has direct access to the signals of the design unit because the `bind` statement connects the two together.

4.1.3. Port directions

It is worth pointing out that the direction of the testbench ports are the mirror opposite of the design-under-test. An output of the design becomes an input into the testbench, and an input becomes an output in the same way that clocking block directions are specified. Table 1 lists the port directions of various testbench components needed to reuse the block-level testbench in a full-chip environment.

Design	Testbench	Interface	Clocking block
output	input	input	input
input	output	output	output

Table 1. Port directions for testbench components.

Using testbench ports like this has a few potential compilation issues. In the block-level environment, the testbench ports are left unconnected since the top module is uninstantiated in simulation. In a full-chip environment, however, these ports may have additional drivers that need consideration. SystemVerilog allows variables to be used for both input and output ports as well as for continuous assignment statements, but variables have no resolution functions and can have at most one driver. Likewise, using a clocking block inside of an interface helps avoid race conditions, but clocking blocks also use variables for driving signals. For this reason, binding the block-level testbench into a full-chip environment may result in multiple drivers to variables, resulting in compilation or elaboration errors. In this case, using wires instead of variables is a better choice to avoid these problems.

Complicating matters further, some active testbench components may be unnecessary in a full-chip environment so the signals that were once outputs of the testbench become only inputs into the testbench monitors. In other words, where the environment resides affects the direction of the testbench ports. Therefore, `inout` ports are preferable for signals shared by both driving and passive components. Creating multiple clocking blocks with different port directions may help simplify the testbench connections (see Figure 16).

```
interface apb_intf (
    input PCLK,
    input [15:0] PRDATA,
    inout [15:0] PADDR,
    inout [15:0] PWDATA,
    inout PSEL,
    inout PENABLE,
    inout PWRITE
);

// Clocking block for active driving components
clocking drv_cb @(posedge PCLK);
    input #1step PRDATA;
    output #1 PADDR, PWDATA, PSEL, PENABLE, PWRITE;
endclocking : drv_cb
```

```

// Clocking block for passive monitor components
clocking mon_cb @(posedge PCLK);
  input #1step PRDATA;
  input #1step PADDR, PWDATA, PSEL, PENABLE, PWRITE;
endclocking : mon_cb

// modport to provide controlled access from testbench
modport test_mp ( clocking drv_cb, clocking mon_cb );

endinterface : apb_intf

```

Figure 16. Interface example using multiple clocking blocks and `inouts` for signals no longer driven by the block-level testbench drivers.

Another problem may be encountered depending on the implementation of your simulator. In Figure 16 above, the `drv_cb` clocking block will be unused in a full-chip environment if the full-chip design drives those particular signals. Clocking blocks create variables called *clocking vars*, which drive continuously as one would expect. When the clocking block is no longer driven in the full-chip environment, these clocking vars drive a strong value of the variable's initial value (X for 4-state and 0 for 2-state), creating a driving conflict on these signals usually resulting in X. Since clocking blocks provide no way to specify an initial clocking var value, a temporary variable can be used that is initialized to a default value of high impedance (Z). Now when they are no longer driven in the full-chip environment, no driver conflicts are created. An example of this is shown in Figure 17. Interestingly, some simulators² exhibit this clocking block driving conflict while others³ do not.

```

interface apb_intf (
  input PCLK,
  input [15:0] PRDATA,
  inout [15:0] PADDR, PWDATA,
  inout PSEL, PENABLE, PWRITE
);
  // Assign high impedance to avoid driver conflicts in full-chip environment
  logic         penable = 'bz;
  logic [15:0]  paddr = 'bz;
  logic [15:0]  pwrdata = 'bz;
  logic         psel = 'bz;
  logic         pwrite = 'bz;

  // Create the drivers to drive the high impedance when not driven by drv_cb below
  assign PENABLE = penable,
         PADDR   = paddr,
         PWDATA  = pwrdata,
         PSEL    = psel,
         PWRITE  = pwrite;

```

² E.g., IUS 8.x[5] and VCS 2008.x[6].

³ E.g., Questasim 6.x[4].

```

// Clocking block for active driving components
clocking drv_cb @(posedge PCLK);
    input #1step prdata = PRDATA;
    output #1 paddr, pwidth, psel, penable, pwrite;
endclocking : drv_cb

...

```

Figure 17. Local variables initialized to high impedance in order to avoid driver conflicts.

4.2. Conditional compilation

Another reuse issue remains in a full-chip environment—what to do with the instantiated design in the block-level testbench since the design-under-test is now part of the entire full-chip design? The obvious answer is to comment out the instantiated design module. Yet if structured correctly, conditional compilation can eliminate the need for modifying or “hacking” the block-level environment.

Verilog provides two simple mechanisms for conditional compilation: (1) pre-processor directives like ``ifdef`, and (2) `generate` statements. While both methods work, `generate` statements are less messy and can take advantage of the same parameters used to parameterize classes in the block-level testbench environment. By parameterizing the top testbench module, a `generate` statement can be added as follows:

```

module block_tb #(parameter ENV = `BLOCK_LEVEL) (
    input clk, inout psel, ..., input [`WIDTH:0] prdata
);
    tb_env    tb;
    bit      local_clk;    // New signal so multiple drivers not on "clk"

    apb_intf apb_bus( .clk( local_clk ), .* );

    generate
        if ( ENV == `BLOCK_LEVEL ) begin : gen_dut
            design dut ( apb_bus );    // Instantiate design unit

            // Clock generation logic needed for block-level TB
            always `PERIOD local_clk = ~local_clk;
        end
        else
            assign local_clk = clk;    // Use input if in sys-level environment
    endgenerate

    initial begin
        tb = new ( apb_bus, ... );
        ...
    end
endmodule : block_tb

```

Figure 18. Generate statements used to conditionally instantiate the design and glue logic.

Using this block-level testbench structure, only a single parameter needs to be changed to use it in a full-chip environment. When used by itself, the testbench instantiates the design and generates the clocking logic; when used in a full-chip verification environment, the testbench uses the internal design clock and no longer instantiates the block-level design unit. Combined with the `bind` statement, the entire block-level environment can be reused in a full-chip verification environment without modification with just one statement:

```
bind fullchip_block_tb #( .ENV( `FULL_CHIP )) tb_inst (
    .clk( global_clk ), .psel( sel_a ), ... );
```

Figure 19. Bind statement using a parameter to conditionally instantiate components.

4.3. Configurable verification components

Conditional compilation solves block-level testbench reuse issues like removing design instantiations, clock generation logic, and multiple drivers on the input/output signals arising from binding into the full-chip design; yet, more drivers remain. The class-based testbench objects also drive the same signals through the virtual interface; therefore, the testbench needs the ability to disable actively driving components like drivers and transactors when used in a full-chip environment.

Since class-based environments are dynamically created at run-time, many solutions are possible for disabling active components (e.g., pre-processor directives, PLI/DPI, parameters, factory mechanisms, etc.). Perhaps the simplest method is to use parameterized classes. This has the distinct advantage of using the same parameter that the testbench's top module uses for conditional compilation. For example, suppose your testbench environment had verification components (e.g., OVM agent / VMM subenv) like this:

```
class apb_subenv #(parameter IS_ACTIVE = `TRUE) extends `VMM_SUBENV;

    apb_transactor    xactor;
    apb_driver        drv;
    apb_monitor       mon;

    function void build();
        super.build();

        if (IS_ACTIVE) begin // Create active component if IS_ACTIVE
            xactor = new ( .. );
            drv    = new ( .. );
        end

        mon = new ( .. );
    endfunction : build
    ...
endclass : apb_subenv
```

Figure 20. Parameterized class-based testbench component.

In this example, the parameter affects the building and configuring of the verification component. When the `IS_ACTIVE` flag is set, the APB component actively drives signals through the virtual interface by the driver; otherwise, it only passively monitors traffic on the interface. The environment class uses the same parameter and passes it on to its many sub-components (Figure 21):

```
class tb_env #(parameter IS_ACTIVE = `TRUE) extends vmm_env;

    apb_subenv    #( IS_ACTIVE ) apb_comp;    // APB component
    mcbbsp_subenv #( IS_ACTIVE ) mcbbsp_comp; // McBSP component
    usbhs_subenv  #( IS_ACTIVE ) usbhs_comp;  // USB High Speed Component
    ...

endclass : tb_env
```

Figure 21. Class parameter passed to various active testbench components.

Now putting everything together, the parameter used to specify whether the environment is used in the block or full-chip environment can be passed directly into the testbench environment:

```
bind fullchip block_tb #( .ENV(`FULL_CHIP)) tb_inst (
    .clk(global_clk ), .psel( sel_a ), ... );

module block_tb #( parameter ENV ) (
    input pclk, ...,
    input [`WIDTH:0] prdata
);

    tb_env #( .IS_ACTIVE( ENV == `BLOCK_LEVEL )) tb;    // Results in
                                                         // IS_ACTIVE = `FALSE
                                                         // which turns off the
                                                         // testbench components

    ...

endmodule : block_tb
```

Figure 22. Top module parameter passed into parameterized classes.

The use of parameterized classes does have the unfortunate drawback of requiring the identical parameter specified whenever the class is declared or used with `typedef`. For example, this means that `apb_subenv #(`TRUE)` is a different class than `apb_subenv #(`FALSE)`. Therefore, using a common base class pointer to `apb_subenv` and calling `$cast()` on it cannot be safely done since the parameter of the object may be incompatible. Alternately, the use of an *active flag* works just as well and an example is shown in Figure 23.

```

bind fullchip block_tb #( .ENV(`FULL_CHIP)) tb_inst ( ... );
module block_tb #( parameter ENV ) ( ... );

    tb_env  tb;

    initial begin
        tb = new ( ... );
        tb.apb_comp.is_active = ( ENV == `BLOCK_LEVEL );    // Set the active flag
        ...
    end
endmodule : block_tb

class apb_subenv extends `VMM_SUBENV;

    bit is_active = 1;    // Active by default

    function void build();
        super.build();

        if (is_active) begin // Create active component only if is_active
            xactor = new ( .. );
            drv     = new ( .. );
        end
    end

    ...

```

Figure 23. Use of an active flag instead of a parameterized class to disable testbench components.

Note that some methodologies recommend the use of callbacks in active transactor components. If a transactor callback is used to pass information to a scoreboard, then disabling the transactor may also disable the scoreboard checking in the block-level environment. Therefore, care should be taken that a block-level checker or scoreboard also work in a full-chip environment, which can easily be accomplished by using only passive components (i.e., monitors) to send information to the scoreboard.

5 Reuse in full-chip testcases

5.1. OVM environments

Significant effort goes into developing block-level transaction generators, drivers, hierarchical stimulus (sequences), etc. so accessing these components in a full-chip testcase would have great reuse benefits. With a factory-based methodology like OVM[3], this is easily accomplished by configuring the factory before building the testbench environment. For example, Figure 24 demonstrates how the block-level sequencer can be configured to use a particular default sequencer. Care should be taken not to call `run_test()` more than once so the block-level testbench should conditionally invoke it.

<pre> class sys_test1 extends ovm_test; sys_env systb; function void build(); // Configure block-level seqr // using the factory mechanism set_config_string ("*.serio_seqr", "default_sequence", "serio_parity_err_seq"); ... endfunction ... endclass : sys_test1 </pre>	<pre> module block_tb #(ENV) ...; tb_env tb; initial begin tb = new (...); // Only invoke run_test() once!! if (ENV == `BLOCK_LEVEL) run_test(); end ... endmodule: block_tb </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 24. OVM example using the factory to configure a block-level testbench sequencer.

5.2. VMM and other environments

In other types of environments like VMM[1], accessing the bound block-level testbench only requires a reference to the testbench object by a global variable. A global variable can easily be created by either defining a variable in a package or through a static class member. When the testbench object is created, the class constructor saves a reference to itself into the global variable. Then, the full-chip top module or testcase accesses the individual class-based components through a hierarchical reference using the block-level testbench reference⁴ (see Figure 25). Since the components are SystemVerilog objects, using a hierarchical reference is not an issue as in VHDL.

<pre> package block_pkg; typedef block_env; // Forward typedef block_env _env; // Global var class block_env extends base_class; function new(); // Save reference to self _env = this; ... endfunction ... endclass </pre>	<pre> module fullchip_tb; block_env block_tb; my_special_xactor xactor; initial begin // Grab reference block_tb = block_pkg::_env; // Reference OK in SysVerilog block_tb.serio_comp.serio_xactor = xactor; ... end </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 25. Global package variable provides access to block-level testbench wherever it is bound into the full-chip environment.

⁴Of course, this only works for one instance of the testbench environment. In environments using multiple testbench objects, a global queue of references would suffice. In practice, however, this is probably not a common concern.

Alternatively, a static class member can be used as shown in Figure 26. So no matter where the block-level testbench is bound, the full-chip testcases still can access and configure the block-level testbench components (see Figure 27).

<pre>class block_env extends base_class; static block_env _env; // Global var function new(); // Save reference to self this._env = this; ... endfunction ... endclass : block_env</pre>	<pre>module fullchip_tb; block_env block_tb; my_special_xactor xactor; initial begin // Grab reference block_tb = block_env::_env; // Reference OK in SysVerilog block_tb.serio_comp.serio_xactor = xactor; end</pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 26. Global static class member provides access to block-level testbench wherever it is bound into the full-chip environment.

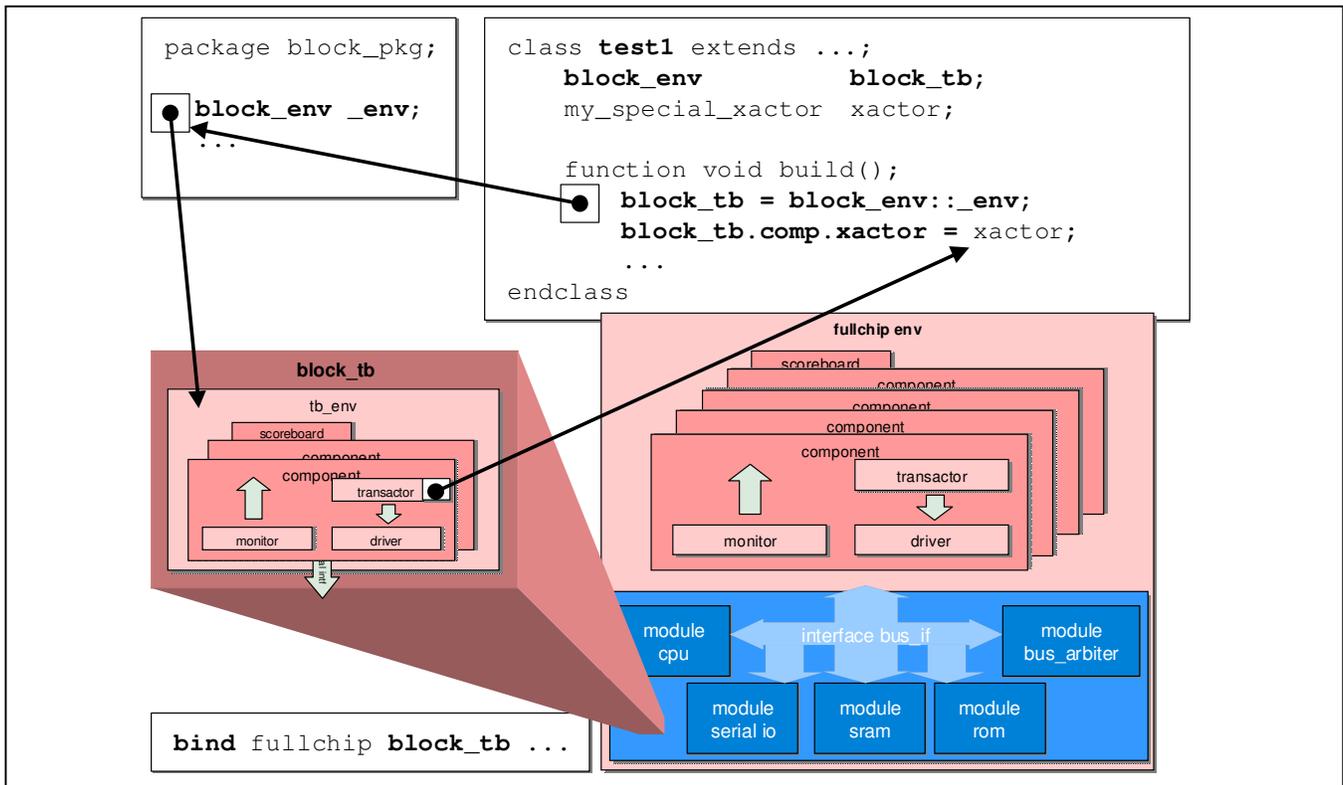


Figure 27. Diagram illustrating how a global package variable can be used to reference a testbench wherever it is instantiated (bound).

5.3. VMM and OVM interoperability⁵

One of the advantages of using this `bind` methodology is that totally heterogeneous verification environments can easily work together. Whether a testbench is written in VHDL, Verilog, or SystemVerilog, it can be bound together with any other environment.

A good application for this is interoperability between VMM and OVM environments. For instance, suppose a block-level environment is written in VMM and needs to be integrated together with an OVM full-chip environment. The `bind` statement can insert the VMM environment within the OVM testbench where both co-exist and execute in parallel without modifications. Each environment can access the other's components via a global reference as described above in Section 4.3. For example, an OVM environment could start the phases in VMM as follows:

```
// Bind the VMM environment into the OVM full-chip environment
bind fullchip vmm_block_tb #( .ENV(`FULL_CHIP)) vmm_tb_inst (
    .clk( global_clk ), .psel( sel_a ), ... );

// The full-chip OVM environment
class ovm_fullchip_env extends ovm_env;
    ...
    // Grab a reference to the VMM environment
    vmm_block_env vmm_block_tb = vmm_block_pkg::_env;

    function void build(); // OVM build phase
        super.build();

        // Access the VMM environment and start the VMM phases
        vmm_block_tb.env.gen_cfg();
        vmm_block_tb.env.build();
    endfunction : build

    task run(); // OVM run phase
        fork vmm_block_tb.env.run(); join_none // Kick off the VMM run phase
        ...
    endtask : run
endclass : ovm_fullchip_env

// Top-level OVM testbench
module ovm_fullchip_tb;
    ...
    initial
        run_test(); // Kick off the OVM machinery
endmodule : ovm_fullchip_tb
```

Figure 28. OVM example accessing a VMM environment.

⁵ Note, some EDA vendors provide compatibility libraries worth considering, which provide an alternative approach for interoperability.

Of course, swapping out components or transaction objects from one environment to another requires using the appropriate base class objects. The appropriate packages or header files need to be included and the appropriate VMM or OVM class types used.

6 Putting it all together

Bringing all the pieces together, a block-level testbench can easily and effortlessly be reused without modifications using a single `bind` statement provided it follows a set of simple templates. The following figures distill and enumerate these essential elements and presents the skeletons to use for the top module, interface, class-based environment, and `bind` statement.

```
//
// Block-level top module
//
module block_tb #( parameter env_flag = `BLOCK_LEVEL ) // (1) Parameterize
                  ( design_port_list ); // (2) Add design signals

// (3) Pass the environmental parameter into
// the class-based testbench environment
tb_class #( .active_flag(env_flag == `BLOCK_LEVEL) ) tb_var;

// (4) Conditionally compile
// Use the generate statement around the bits of code that need to be turned
// off in a full-chip environment. For example, the DUT no longer needs
// instantiated, clock generation logic needs disable, etc.
generate
  if ( env_flag == `BLOCK_LEVEL )
    // Instantiate the design.
    // e.g., design dut ( ... );

    // Local clock, reset, muxing, etc. logic.
    // e.g., always local_clk = `PERIOD ~local_clk;
  else
    // Use input clock, reset, etc.
    // e.g., assign local_clk = <sysclk>;
  endgenerate

// (5) Connect the ports to the interface
intf intf_inst( .clk( local_clk ), .* );

initial begin
  // Create the testbench and pass the virtual interface into it
  tb_var = new ( intf_inst, ... );
  ...
end
endmodule : block_tb
```

Figure 29. Block-level testbench top module template.

```

//
// Interface to DUT used by the class-based testbench
//

interface intf (
    input wire ...,                // (6) Use ports and wires instead
    output wire ...,              // of local variables

    inout SIGNAL1, SIGNAL2, ...   // (7) Use inout ports for signals
                                   // driven by the driver and
                                   // read by the monitor (see
                                   // Section 4.1.3).
);

// (8) Assign high impedance to avoid driver conflicts in full-chip environment
logic      signal1 = 'bz;
logic      signal2 = 'bz;
...

// (9) Create the drivers to drive the high impedance when not driven by drv_cb
assign     SIGNAL1 = signal1,
          SIGNAL2 = signal2,
          ...;

// (10) Create clocking block for active driving components
clocking  drv_cb @(posedge PCLK);
input #1step ...;
output #1 signal1, signal2, ...;
endclocking : drv_cb

// (11) Create clocking block for passive monitor components
clocking  mon_cb @(posedge PCLK);
input #1step ...;
input #1step signal1, signal2, ...;
endclocking : mon_cb

// modport to provide controlled access from testbench
modport test_mp ( clocking drv_cb, clocking mon_cb );

endinterface : apb_intf

```

Figure 30. Testbench to DUT interface template.

<pre> // // Class-based testbench environment // // (12) Parameterize testbench class class tb_env #(parameter active_flag = `TRUE) extends base_class; // (13) // Pass active flag to all // components via a parameter component #(active_flag) comp; /***** OR ... *****/ // Use a flag, which is // more flexible comp.active_flag = active_flag; endclass : tb_env </pre>	<pre> // // Testbench components // // (14) // Parameterize component classes class component #(parameter active_flag = `TRUE) extends base_class; /***** OR ... *****/ // Use active flag instead of // parameter class component extends base_class; bit active_flag = `TRUE; ... // (15) Build active components only // if active flag is set if (active_flag) begin xactor = new (...); driver = new (...); end </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 31. Block-level class-based environment template.

```

//
// Bind statement
//

// (16) Use the bind statement to put the block-level environment into the
// full-chip environment. Set the environmental flag to FULL_CHIP and
// connect the design ports to the testbench.
bind design block_tb #( `FULL_CHIP ) tb_inst( design_port_list );

```

Figure 32. Full-chip bind statement template.

7 Conclusions

The SystemVerilog `bind` statement offers a perfect solution for accessing signals within a mixed-language design. Additionally, it offers an ideal solution for using any block-level testbench in any full-chip environment. Reuse could not be made simpler.

However, this method may pose a few tool-related challenges. Since using a single `bind` statement is simple, the most difficult and time-consuming aspect of this methodology is integrating together the *makefile* tool flows of the different testbench environments. For example, class libraries may differ (e.g., VMM vs. OVM), include paths may change, and tool options may be different between environments.

Similarly, simulator support for SystemVerilog features like the `bind` statement or parameterized classes often pose an obstacle to using this methodology. Not all simulators treat `bind` equally. For example, when using the `bind` statement in its own file, QuestasimTM[4] requires the additional compilation flags `-mfcu -cuname name` in order to place it in its own `$unit` as well as specifying the unit name as a top module when launching the simulator.

Similarly, some simulator versions do not fully or adequately support parameterized classes so using an active flag instead of a parameter is probably a better option for tool portability. The use of static variables in a class or global variables within a package for accessing bound testbench objects may also have limited support in older simulator versions. In general, however, all recent versions of major mixed-language simulators provide adequate support for the methodology presented in this paper.

Therefore, this methodology offers a generic, simple, and effective reuse methodology for any type of class-based testbench environment. With one `bind` statement, much effort rewriting block-level checks, stimulus, and coverage is saved, and the full-chip verification effort can hit the ground running regardless of the HDL language used in the design.

8 Acknowledgements

I would like to thank my colleagues at Doulos for their valuable feedback, especially Jonathan Bromley for his helpful insights and suggestions, and my many students who prompted these ideas.

Any trademarks or other proprietary names mentioned in this paper are acknowledged as the property of their respective owners.

9 References

- [1] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2005, 2005.
- [2] Bergeron, Janick, E. Cerny, A. Hunter, and A Nightingale. Verification Methodology Manual for SystemVerilog. Norwell, MA: Springer, Inc, 2005.
- [3] "OVM Reference Manual Version 2.0", Cadence Design Systems Inc (San Jose, CA) and Mentor Graphics Inc (Beaverton, OR), September 2008.
- [4] "Questasim 6.4a" tool from Mentor Graphics Inc, Beaverton, OR.
- [5] "IUS 8.x" tool from Cadence Design Systems Inc, San Jose, CA.
- [6] "VCS 2008.x" tool from Synopsys Inc, Mountain View, CA.