CHAPTER

3

THE PROCESS

With the emergence of assertion and property language standards, design teams are investigating assertion-based verification techniques and finding that there is generally value in applying these techniques [Foster et al., 2004]. In spite of this general acceptance, there is a huge disconnect between attempting to write a collection of embedded implementation assertions and creating a comprehensive reusable assertion-based IP verification component. For example, if you attempt to create assertion-based monitors for a complex bus interface or a memory controller without approaching the task systematically and planning each step, then it is likely that the quality of your results will be poor.

In this chapter, we introduce a systematic set of steps to help you effectively create your assertion-based IP. Next, we focus on the process of implementing a SystemVerilog module-based assertion monitor. Using a SystemVerilog interface or module-based component (versus a class-based component) is necessary when implementing an assertion-based monitor since general temporal assertions are not allowed within a SystemVerilog class. Nonetheless, as we demonstrate, you can successfully create a testbench that combines both module-based and class-based components. To support this framework, we discuss a class-based communication mechanism, which is used to connect and then establish communication between these module-based and class-based components.

3.1 Guiding principles

As we mentioned in the introduction, the guiding principles we embrace when creating an assertion-based IP monitor are:

- ***modularity***—separate *detection* from *action*
  - Facilitates reusable verification components
  - Simplifies maintenance
  - Supports advanced features such as error injection
- ***clarity***—target your assertions for simulation
  - Initially focus on capturing intent
  - Do not get distracted by formal verification optimizations

Modularity facilitates reusable verification components and simplifies maintenance. A clear separation between assertion (and coverage) *detection* from *action* does not restrict assertion-based IP use. For example, as demonstrated later in this chapter, an assertion-based monitor might detect that a bus protocol violation occurred. Then the assertion-based monitor would pass on error status information (via an analysis port) to other verification components within the testbench. Using an analysis port, the assertion-based monitor does not need to know the details of the testbench architecture (or even who is connected to its analysis port). Hence, this framework supports advanced testbench features such as error injection. For example, one component within the testbench might have injected an error into a transaction, and it is expecting an error condition to be detected. Once it is alerted of this condition by the assertion-based monitor, it can then take an appropriate action.

Specification versus implementation assertions

We are not promoting that you separate detection from action for every assertion in your design. Certainly, this technique would be impractical if applied to the thousands of implementation assertions embedded in an RTL design. However, we are promoting this approach for the case when you create a reusable assertion-based verification IP component that must communicate with other verification components within a testbench.

3.2 Process steps

The systematic steps we follow are based on the work of [Foster et al., 2006b]. However, there are differences in the process we present in this chapter. Dasgupta [2006] also presents a similar process, yet his work was narrowly focused on the testplanning process targeted at formal verification, which often requires introducing advanced strategies (and abstractions) to get the set of assertions to converge during a formal proof. In this chapter, the process we present is focused on creating reusable verification components for simulation. Our steps, illustrated in Figure 3-1, are as follows:

Step 1 **Create a block diagram and interface description.** Create a block diagram and table that describes the details for the design's design component interface signals that must be referenced (monitored) when creating the set of assertions and coverage items. You will use this list to determine completeness of the requirement checklist during the review process.

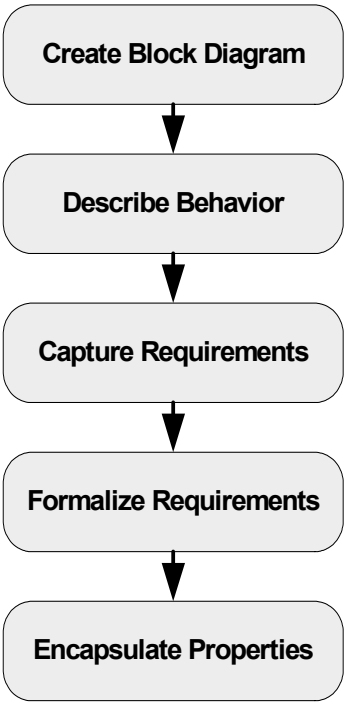
Step 2 **Create an overview description.** Briefly describe the key characteristics of your design's design component. You do not have to make the introduction highly detailed, but it should highlight the major functions and features. Waveform diagrams are useful for describing temporal relationships for temporal signals.

Step 3 **Create a natural language list of properties.** In a natural language, list all properties for your design's design component. We recommend you create a table to capture your list of properties. For each property, use a unique label identifier for each property that helps map the assertions back to the natural language properties.

Step 4 **Convert natural language properties into formal properties.** In this step, convert each of the natural language properties into a set of SVA (or PSL) assertions or coverage properties, using any additional modeling required for describing the intended behavior.

Step 5 **Encapsulate assertions inside a module or interface.** In this step, we turn our set of related properties into a reusable verification component (an assertion-based monitor). We add analysis ports to our monitor for communication with other simulation verification components, providing a clear separation between assertion detection and testbench action.

Figure 3-1 Assertion-based IP creation process steps



Scope of our discussion

We illustrate each of these assertion-based IP creation steps in the following chapters. Obviously, in terms of creating a final VIP product, there are many other deliverables beyond the assertion-based monitor itself (for example, documentation, quality goals, testing strategy). Each one of these important topics requires careful attention—and books could (and should) be written to address each of these topics. However, we have decided to limit the scope of our discussion in this book to the process of creating the assertion-based monitor verification component. We believe

that this is a fundamental skill that should be mastered before attempting to productize VIP.

3.3 Assertion-based IP architecture

In this section, we discuss architectural aspects of creating assertion-based IP verification components. We chose the Advanced Verification Methodology [Glasser et al., 2007], which is a subset of the newly formed OVM, base-class library to demonstrate our assertion-based IP creation process. We chose the OVM for our examples because the source code for the OVM library is openly available and can be downloaded at *http://www.mentor.com/go/cookbook*. Assuredly, there are other testbench base-class libraries available, and we encourage you to choose one that you feel comfortable with when creating your assertion-based IP. The general ideas, processes, and techniques we present in this book are easily extended to other available testbench base-class libraries.

Module versus interface assertion-based IP

We begin by demonstrating how to create assertion-based IP, which is based on a module implementation, and we introduce many key concepts during this discussion (such as interfaces and analysis ports). Once we introduce all the key concepts, we discuss an alternative form of creating assertion-based IP, which is based on a SystemVerilog interface implementation. In general, we favor the interface implementation (for protocol style assertion-based IP); however, not all assertion-based IP is limited to specifying only interfaces or protocols (that is, some assertion-based IP specifies end-to-end behavior and some assertion-based IP involves multiple design components). Furthermore, there are situations where a design project might require module-based forms of assertion-based IP (or have project restrictions on interface content). Hence, we show you how to do both.

class versus module verification components

Class-based versus module-based components. Today, you will find that many modern programming languages are based on object-oriented programming concepts. Even hardware design and verification languages, such as SystemVerilog, support object-oriented capabilities using the class construct. However, what distinguishes an object-oriented program from other programming approaches is its organization. For example, an object-oriented program is a collection of interacting objects (that is, instances of classes)—where each object has its own data space and set of functionality. The object's data is accessed by a collection of methods, which are really functions that serve as an object's interface.

In a way, you can think of each instance of a Verilog module as an object, which has its own data space, set of functionality, and well-defined interface. However, one notable difference between a module and a class is that a module is static in its existence (that is, it cannot be created dynamically during a program's execution), and does not support type parameterization or inheritance.

Inheritance is a powerful technique that lets you achieve productivity and uniformity when creating various program objects—such as different verification components within a testbench. For example, you can use a base class defined within the OVM library to derive (through inheritance) your own customized verification component. This approach takes advantage of all the existing functionality within an OVM base class (that is, without having to recreate it), which improves your productivity. In addition, accessing common methods across various base classes contained within the OVM library minimizes the learning curve involved in creating and maintaining verification components.

connecting module-based components to class-based components

Mixed verification environment. A question often arises: "Should I use classes or modules to create verification components?" The answer is—it depends. Modules are more natural for the HDL user. On the other hand, classes are more flexible with their support of inheritance for

customization, they are easier to randomize than modules, and they allow flexible instantiation.

While class-based verification components have a number of advantages over module-based verification components, there are still situations that require module-based verification components or justify their use. For example, to manage project resources, you might be forced to use some legacy module-based verification components from a previous project—or even purchase some new module-based third-party verification IP. Hence, the ability to mix existing module-based verification components with newly developed class-based verification components is critically important to many projects.

Another example where mixed class-based and module-based verification components are required is the use of assertions within a testbench. That is, SystemVerilog does not allow temporal assertions within a class. Hence, it is necessary to capture our assertions within a module (or SystemVerilog interface), and then integrate our module-based monitor with other class-based components contained within our testbench.

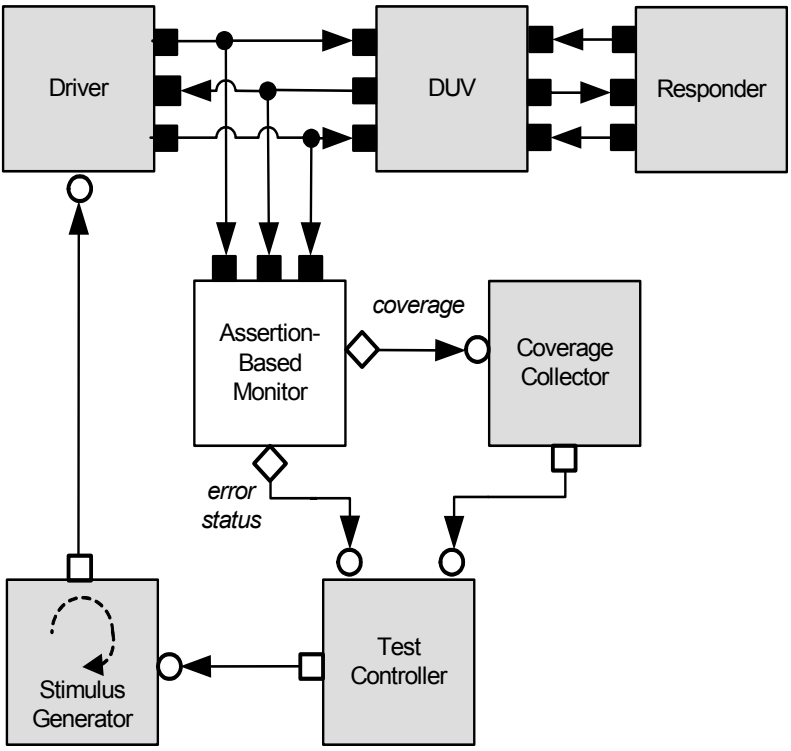
Regardless of whether you choose to implement a class-based or module-based verification component, we can use the same communication mechanism (implemented as a class) to communicate between these mixed implementation techniques. For example, a class can be passed as inputs to a module, which then can be used to connect module-based verification components to class-based verification components.

Testbench organization with assertion-based IP

Figure 3-2 illustrate the organization of a testbench that includes an assertion-based monitor, which contains an RTL design (that is, the DUV) with pin-level, timed-bus interfaces. The testbench in this example consists of a set of verification components that communicate with each other (via classes) using untimed transactions. Untimed transactions (for example, a read or a write request to a specified address) are sent to a driver transactor, whose role

is to convert a stream of untimed transactions into pin-level timed activity.

Figure 3-2 Assertion-based monitor interfaces



3.3.1 Module-based assertion IP

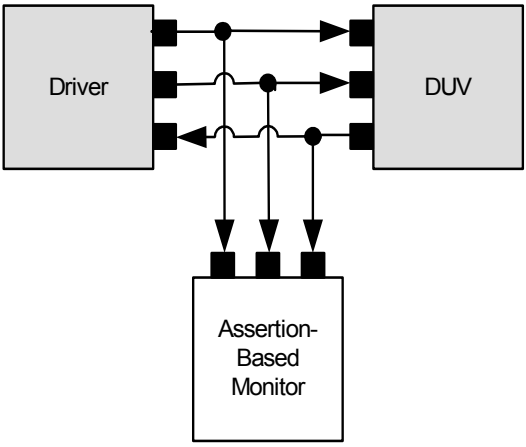
The assertion-based monitor in our example is a module-based transactor. Its role is to monitor the bus pin-level activity, and identify protocol violations and interesting sequences to be used for coverage. All bus protocol violations are reported to the testbench's simulation controller (for appropriate action) through a status analysis port (illustrated by the diamond-shaped connector in Figure 3-2), which is a parameterized class used to transport a status class transaction. In addition, the timed pin-level

activity is converted back into an untimed transaction and is passed to the coverage collector through a different analysis port.

3.3.2 SystemVerilog interface

To establish a connection between our assertion-based monitor, driver transactor, and the DUV, we create a SystemVerilog interface, as illustrated in Figure 3-3. A SystemVerilog interface encapsulates the communication between multiple blocks. Hence, a SystemVerilog interface forms a single connection description that supports bus definition reuse across multiple verification components and the DUV. This means that if you make a change to the interface definition, it is reflected automatically across all bus components that share the same interface structure.

Figure 3-3 SystemVerilog interface



For instance, in Example 3-1 we demonstrate an interface for the design illustrated in Figure 3-3. For this case, our assertion-based monitor would reference the interface signals (for example, `sel` or `en`) via the direction defined by the `monitor_mp` named modport.

Example 3-1 Encapsulate signals inside a SystemVerilog interface

```
interface tb_bus_if( input clk , input rst );

  parameter int DATA_SIZE = 8;
  parameter int ADDR_SIZE = 8;

  bit       sel;
  bit       en;
  bit       write;
  bit [DATA_SIZE-1:0] wdata;
  bit [DATA_SIZE-1:0] rdata;
  bit [ADDR_SIZE-1:0] addr;

  modport driver_mp (
    input        clk , rst ,
    output       sel , en , write , addr ,
    output       wdata ,
    input        rdata
  );

  modport duv_mp (
    input        clk , rst ,
    input        sel , en , write , addr ,
    input        wdata ,
    output       rdata
  );

  modport monitor_mp (
    input        clk , rst ,
    input        sel , en , write , addr ,
    input        wdata ,
    input        rdata
  );

endinterface
```

Top-level interface instantiation and module-based reference

Example 3-2 demonstrate how a SystemVerilog interface is instantiated in the top module of a testbench. The interface `monitor_mp` modport is passed as an argument into our assertion-based monitor example, which then establishes a connection between the signal defined within the SystemVerilog interface and the monitor.

Example 3-2 Instantiated SystemVerilog interface and monitor

```
module top;

  . . .

  tb_bus_if #( .DATA_SIZE(8), .ADDR_SIZE(8)) nonpiped_bus_if (
     .clk( clk_rst_bus.clk ),
     .rst( clk_rst_bus.rst)
  );

  assertion_monitor tb_monitor(
    .monitor_mp( nonpiped_bus_if.monitor_mp )
  );
  . . .
endmodule
```

Example 3-3 demonstrates how the SystemVerilog interface monitor `modport` is referenced inside a module-based component, such as our assertion-based monitor example.

Example 3-3 Interface modport references inside assertion monitor

```
module assertion_monitor ( interface.monitor_mp monitor_mp );

  parameter DATA_SIZE  = 8;
  parameter ADDR_SIZE  = 8;

  bit  [ADDR_SIZE-1:0] bus_addr;
  bit  [DATA_SIZE-1:0] bus_wdata, bus_rdata;
  bit                  bus_write;
  . . .
  always @(posedge monitor_mp.clk) begin

    bus_addr  = monitor_mp.addr;
    bus_wdata = monitor_mp.wdata;
    bus_rdata = monitor_mp.rdata;
    bus_write = monitor_mp.write;
    . . .
  end
. . . .
endmodule
```

class-based reference of an interface

Thus far, we have seen how an interface is used (that is, referenced) inside a module-based component. To complete the connection of our interface, the signals must be accessible to class-based transactor components, such as the driver shown in Figure 3-2. For our testbench class-based components, we encapsulate them within a special class known as an environment class (see Appendix B,

"Complete OVM/AVM Testbench Example" for details), which is then instantiated within our top-level module (see `env` in Example 3-4). Our SystemVerilog interface is passed into the environment class constructor (`new`), as illustrated in Example 3-4.

Example 3-4 Passing a SystemVerilog interface into a class

```
module top;

  . . .

 // System Verilog Interface for clock and reset
  clk_rst_if clk_rst_bus();

 // Module-based driver for clock and reset
  clock_reset clock_reset_gen( clk_rst_bus );

 // Environment class to encapsulate class-based testbench
components
  tb_env env;

 // System Verification Interface
  tb_bus_if #( .DATA_SIZE(8), .ADDR_SIZE(8)) nonpiped_bus_if (
     .clk( clk_rst_bus.clk ),
     .rst( clk_rst_bus.rst)
  );

 // Module-based assertion monitor
  assertion_monitor tb_monitor(
    .monitor_mp( nonpiped_bus_if.monitor_mp )
  );

 // Module-based DUV
  duv my_duv (
    .driver_mp( nonpiped_bus_if.driver_mp ),
    . . .
  );

  initial begin
  // Environment class where SystemVerilog interface is passed into
  // its constructor
    env = new( nonpiped_bus_if, . . .);
    . . .
  end
endmodule
```

Example 3-5 illustrates the relevant details for our environment class and specifically shows the SystemVerilog interface argument for the environment class

constructor (`new`), which is assigned to our testbench class-based driver virtual interface (in the connect method)—effectively completing the connection between the driver, DUV, and assertion-based monitor.

Example 3-5 Environment class constructor

```
class tb_env extends ovm_env;

  protected tb_driver    p_driver;
    . . .
  virtual tb_bus_if #( .DATA_SIZE(8), .ADDR_SIZE(8) )
              p_nonpiped_bus;

  function new(
                virtual tb_bus_if #(.DATA_SIZE(8),.ADDR_SIZE(8))
                      nonpiped_bus,      . . .
               );
      p_nonpiped_bus = nonpiped_bus;
      p_driver = new("drive", this);
      . . .
  endfunction

  function void connect;
    p_driver.m_bus_if = p_nonpiped_bus; // connect to transactor
    . . .
  endfunction
  . . .
endclass
```

For completeness, Example 3-6 sketches out the skeleton of the driver class-based component. The `m_bus_if` is a virtual interface within the driver transactor. The actual interface was assigned to this virtual interface handle within the environment class as previously shown in Example 3-5. This assignment completes our connection between the module-based assertion monitor, module-based DUV, and class-based testbench driver.

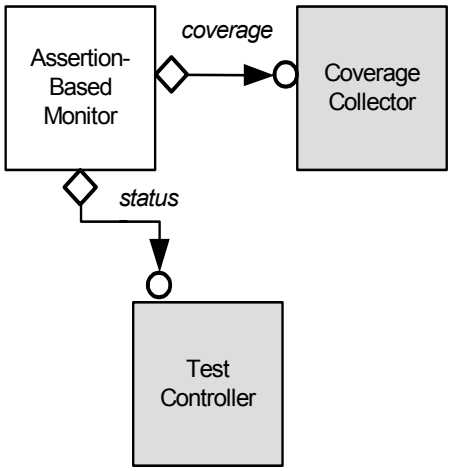
Example 3-6 Class-based driver transactor

```
class tb_driver extends ovm_threaded_component;

  virtual tb_bus_if #( .DATA_SIZE( 8 ), .ADDR_SIZE ( 8 ) ) m_bus_if;

  . . . .

endclass
```

3.3.3 Analysis ports

Within a contemporary testbench, a key observation is that analysis traffic (such as coverage and assertion status) is fundamentally different from operation traffic (such as design data and control information). Hence, for our assertion-based IP architecture, we introduce analysis ports, which enables us to provide a bridge between the operational domain of our testbench, which includes monitoring the DUV, and the analysis domain of our testbench.

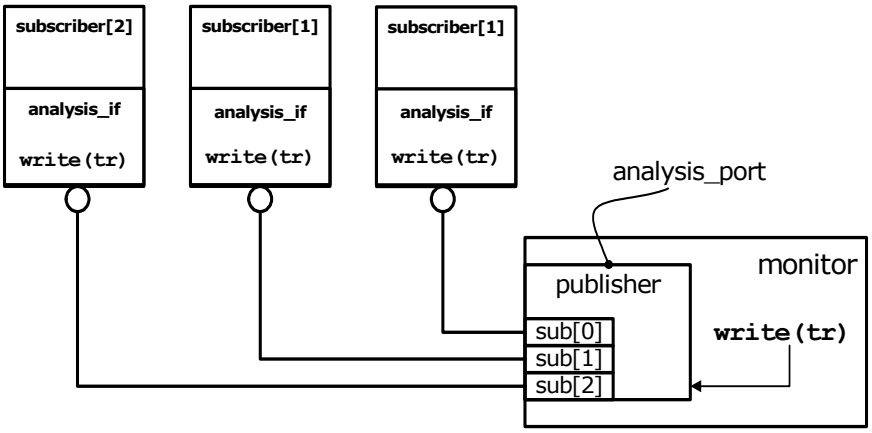
Figure 3-4 Assertion-based monitor analysis ports



An analysis port as implemented in the OVM as a class-based object broadcasts transactions to zero or more listeners within the testbench. In fact, the key benefit of an analysis port is that the assertion-based IP does not need to know the architecture details of the overall testbench to function correctly. For our case, this makes the assertion-based IP truly reusable. For example, in certain testbench architectures, the assertion-based IP might be connected to a coverage collector—while in other architectures the same assertion-based IP might be connected to a scoreboard. Still, there might be other testbench architectures where the assertion-based IP is connected to both a coverage collector and a scoreboard.

To introduce the general concepts of an analysis port, we illustrate the OVM analysis port organization in Figure 3-5. This organization consists of a publisher object and a set of subscriber objects. Each subscriber verification component registers itself with the publisher verification component (such as our assertion-based IP monitor). When the publisher has some new data to publish, it notifies all the registered subscribers.

Figure 3-5 Analysis port organization



At the beginning of simulation, each subscriber registers itself with the publisher and the publisher maintains a list of subscribers. The OVM provides the infrastructure and utilities that automatically handle the connection of the

subscribers to a publisher at the beginning of simulation, so the user does not have to implement these lower-level details. For additional information, Appendix B, "Complete OVM/AVM Testbench Example" provides an overview of the `ovm_analysis_port` class, and you can find more details in [Glasser et al., 2007].

During normal operation, the verification component that owns the analysis port calls `write()`, passing in a transaction object (for example, coverage information or assertion status information). The analysis port then passes a copy of the transaction object to each subscriber.

To help illustrate how the analysis port is used (and connected) within a testbench, we begin by modifying our simple assertion-based monitor example previously shown in Example 3-3 to include an error status analysis port.

Example 3-7 An OVM parameterized analysis port of type tb_status

```
module assertions(
  interface.monitor_mp monitor_mp
);

  ovm_analysis_port #(tb_status) status_ap = new("status_ap",
null);
  . . .
  tb_status status;
  . . .

  property p_valid_inactive_transition;
    @(posedge monitor_mp.clk) disable iff (monitor_mp.rst)
      ( bus_inactive) |=> (( bus_inactive) || (bus_start));
  endproperty

a_valid_inactive_transition:
  assert property (p_valid_inactive_transition) else begin
    status = new();
    status.set_err_trans_inactive();
    status_ap.write(status);
  end
  . . .
endmodule
```

An `ovm_analysis_port` is a parameterized class. For our example (shown in Example 3-7) it is parameterized to be a `tb_status` class, which we defined in Example 3-8. If the

assertion `a_valid_inactive_transition` in Example 3-7 evaluates false, then a `status` object is created. Next, the specific error condition is flagged by calling the appropriate error status method. For example, in our case, the `tb_status` class `err_trans_inactive` method is called. Finally, the `status` object is passed to the analysis port where it is then broadcast to any (and all) registered subscribers.

Note that in terms of assertion status, we never call the analysis port if the assertion evaluates true. Calling an analysis port at every assertion clock edge creates too much activity within a testbench.

Example 3-8 IA tb_status class

```
class tb_status extends ovm_transaction;

typedef enum { ERR_TRANS_RESET ,
               ERR_TRANS_INACTIVE ,
               ERR_TRANS_START ,
               ERR_TRANS_ACTIVE ,
               ERR_TRANS_ERROR ,
               ERR_STABLE_SEL ,
               ERR_STABLE_ADDR ,
               ERR_STABLE_WRITE ,
               ERR_STABLE_WDATA } bus_status_t;

bus_status_t bus_status;
. . . .
function void set_err_trans_inactive;
  bus_status = ERR_TRANS_INACTIVE;
endfunction
. . . .
endclass
```

3.3.4 Interface-based assertion IP

We demonstrated in the previous sections how to create and connect assertion-based IP based on a module implementation. In this section, we demonstrate how to create and connect assertion-based IP based on a

SystemVerilog interface implementation. As we previously mentioned, delivering interface-based assertion IP is the preferred method. However, there are cases (such as end-to-end assertion-based IP or project interface restrictions) that require module-based assertion IP.

Example 3-9 demonstrates assertion-based IP delivered in an interface implementation.

Example 3-9 Encapsulate analysis ports inside interface

```
interface tb_bus_if( input clk , input rst );

  . . .

  ovm_analysis_port #(tb_coverage) cov_ap = new("cov_ap", null);

  property p_burst_size;
    int psize;

    @(posedge clk)
        ((bus_inactive), psize=0)
    ##1 ((bus_start, psize++, build_transaction(psize))
         ##1 (bus_active))[*1:$]
    ##1 (bus_inactive);
  endproperty

  cover property (p_burst_size);

  function void build_transaction(int psize);
    tb_coverage_tr tr;

    tr = new();
    if (bus_write) begin
      tr.set_write();
      tr.data = bus_wdata;
    end
    else begin
      tr.set_read();
      tr.data = bus_rdata;
    end
    tr.burst_count = psize;
    tr.addr        = bus_addr;

    cov_ap.write(tr);
  endfunction
endinterface
```

The SystemVerilog interface contains all required bus interface assertion and coverage properties. We have added analysis ports to the SystemVerilog interface (compared with the simpler interface shown in Example 3-1). We

demonstrate how to connect to these analysis ports in the following examples.

Example 3-9 demonstrates a SystemVerilog interface with a coverage property. In this case, the coverage property reconstructs a multicycle bus sequence into a transaction, which then broadcasts to an analysis port. The transaction contains the type of bus operation (for example, a read or a write), the address, and the data value. In addition, the transaction contains the particular burst transaction sequence number.

Example 3-10 demonstrates the top level module for our testbench.

Example 3-10 Passing an interface with assertions into a class

```
module top;

  . . .

 // System Verilog Interface for clock and reset
  clk_rst_if clk_rst_bus();

 // Module-based driver for clock and reset
  clock_reset clock_reset_gen( clk_rst_bus );

 // Environment class to encapsulate class-based testbench
components
  tb_env env;

 // System Verification Interface
  tb_bus_if #( .DATA_SIZE(8), .ADDR_SIZE(8)) nonpiped_bus_if (
     .clk( clk_rst_bus.clk ),
     .rst( clk_rst_bus.rst)
  );

 // Module-based DUV
  duv my_duv (
    .driver_mp( nonpiped_bus_if.driver_mp ),
    . . .
  );

  initial begin
  // Environment class where SystemVerilog interface is passed into
  // its constructor
    env = new( nonpiped_bus_if, . . .);
    . . .
  end
endmodule
```

Essentially, the only difference between this top-level module and the top-level module in Example 3-4 is that we no longer instantiate an assertion monitor into the top module of the testbench (since our assertions are now encapsulated within the SystemVerilog interface).

Example 3-11 demonstrates how a class-based coverage collector component subscribes to an analysis port contained within a SystemVerilog interface, which was passed into the environment class constructor. Recall that all the testbench class-based components are instantiated within an environment class.

Example 3-11 Environment class constructor with analysis ports

```
class tb_env extends ovm_env;

  protected tb_cov_colr    p_cov_col;
  . . .

  ovm_analysis_port #(tb_coverage) p_if_cov_ap = new("cov_ap",
null);

  function new(
               virtual tb_bus_if #(.DATA_SIZE(8),.ADDR_SIZE(8))
                     nonpiped_bus_if,      . . .
              );
      p_if_cov_ap = nonpiped_bus_if.cov_ap;
      . . .
     // instantiate a class-based coverage collector
      p_cov_col = new("cov_col", this);
  endfunction

  function void connect;
    // coverage collect subscribes to interface analysis port
    p_if_cov_ap.register(p_cov_col.analysis_export);
    . . .
  endfunction
  . . .
endclass
```

3.4 Guidelines and conventions

We highly recommend that you adopt a set of SVA naming conventions associated with declaration names and directive labels, so that when your verification IP consumers read your SystemVerilog assertions, they are able to quickly distinguish internal verification IP names from HDL names. For our examples, we have adopted the naming convention guidelines defined in Table 3-1.

Table 3-1 SVA naming convention guidelines for verification IP

Convention	Use
p_*name*	**p_** is a prefix for property declaration names
s_*name*	**s_** is a prefix for sequence declaration names
a_*label*	**a_** is a prefix for assert directive labels
m_*label*	**m_** is a prefix for assume directive labels (for formal)
c_*label*	**c_** is a prefix for cover directive labels

3.5 Summary

In this chapter, we introduced a set of guiding principles for creating assertion-based IP. Specifically, we discussed the importance of modularity and clarity when creating assertion-based IP. We then presented a systematic set of steps to help you effectively create your assertion-based IP.

To demonstrate how to architect your assertion-based IP to achieve a clear separation between assertion (and coverage) detection from action, we sketched out both a module-based and interface-based assertion IP implementation using the OVM base-class library elements. The concepts we presented in this chapter are easily extended to other base-class libraries that might be available to you.