Open Verification Methodology User Guide

Product Version 2.0
September 2008

Contents

Introduction

This book describes how to use the Open Verification Methodology (OVM) using the OVM Class Library. This book:

- Tells you how this manual is organized and how to use it in your verification work.
- Briefly describes:
  - OVM features.
  - How to install OVM (if necessary).
  - OVM terminology.
- Gives an overview of OVM concepts and verification roles.
- Shows the environment developer how to create reusable verification components.
- Shows the test-environment user how to assemble components into environments and create tests to meet verification goals.
- Provides an example of a OVM environment using the XBus design example.

OVM uses the SystemVerilog OVM Class Library, which is documented in the *SystemVerilog OVM Class Reference*; both are available at www.ovmworld.org.

This chapter contains the following sections:

How to Use this Book

A typical verification team consists of multiple contributors with different skill sets and responsibilities. The different roles of developers and environment users require different depths of verification knowledge. The organization of this manual is based on the way a typical verification team divides its responsibilities.

- Environment developers create the reusable testbench infrastructure.
- Environment users (or integrators) write tests for and configure the testbench infrastructure created by the developer to meet a project's verification goals.

We highly recommend that you read this entire OVM manual—regardless of your individual role—to gain a thorough understanding of OVM. However, this manual is organized in a way that allows the environment users to quickly learn what they need to know without having to read and understand the entire manual. This manual is structured as follows:

Chapter	Description
Introduction	This chapter
OVM Overview	Gives an overview of OVM concepts and verification roles. This chapter should be read by all members of the verification team.
Transaction-Level Modeling (TLM)	Discusses the essential elements of transaction-level communication in OVM and illustrates the mechanics of how to assemble transaction-level components into a verification environment.
Developing Reusable Open Verification Components (OVCs)	Describes to the OVC/environment developer how to create reusable verification components (OVCs). The environment user may also want to read this chapter for a deeper understanding of OVC development.
Using OVCs	Describes to the environment user/integrator how to configure OVCs for a particular verification project and how to create tests and scenarios to meet verification goals.
Advanced Topics	Discusses in greater detail certain topics in the previous chapters.
XBus OVC Example	Steps the user through an example of an OVC based on the XBus specification and OVM.
XBus Specification	The XBus specification

What is OVM?

OVM is a complete verification methodology that codifies the best practices for development of verification environments targeted at verifying large gate-count, IP-based SoCs. Verification productivity stems from the ability to quickly develop individual verification components, encapsulate them into larger reusable verification components (OVCs), and reuse them in different configurations and at different levels of abstraction.OVM supports “bottom-up” reuse by allowing block-level components and environments to be encapsulated and reused as blocks that can be composed into a system. “Top-down” reuse allows transaction-level verification environments to be assembled with system-level models of the design, and then reused as the design is refined down to RTL.

OVM uses a SystemVerilog implementation of standard TLM interfaces for modular communication between components. When coupled with a proven reuse architecture for verification components, OVM delivers a common objected-oriented usage model and ensures that all OVM-compliant OVCs will interoperate regardless of origin or language implementation. Key features of OVM include:

- Data Design—Infrastructure for class property abstracting and simplifying the user code for setting, getting, and printing (text and gui) property variables.
- Stimulus Generation—Classes and infrastructure to enable fine-grain control of sequential data streams for module- and system-level stimulus generation. Users can randomize data based on the current state of the environment, including the DUT state, interface, or previously-generated data. Users are provided out-of-the-box stimulus generation, which can be customized to include user-defined hierarchical transactions and transaction streams.
- Building and Running the Verification Environment—Creating a complete verification environment for an SoC containing different protocols, interfaces and processors is becoming more and more difficult. Base classes are provided for each functional aspect of a verification environment in the SystemVerilog OVM Class Library. The library provides facilities for streamlining the integration of user-defined types into the verification environment. A topology-build infrastructure and methodology provide users flexibility in defining required testbench structures. A common configuration interface enables the user to query and set fields in order to customize run-time behavior and topology.
- Coverage Model Design—Best-known practices for incorporating coverage into a reusable OVC including global and fine-grain control design.
- Built-in Checking Support—Best-known practices for incorporating physical- and functional-layer checks into a reusable OVC, including global and fine-grain control design.
- User Example—A golden example is provided which is based on the XBus protocol. The example includes tests, sequences, testbench structures, and derived OVCs using the methodology and base classes.

Installing OVM

For instructions on installing the OVM kit, please refer to the appropriate README file in the top-level directory of the release kit.

Terminology in This Book

Term	Definition
Agent	A device that contains the standard components necessary to drive HDL signals (the driver), provide stimulus to the driver (the sequencer) and collect data items along with enforcing checks and tabulating coverage (the monitor). An agent is capable of independent operation.
Bus monitor	A verification component responsible for extracting signal information at the bus level and translating it into events, data, and status information
Checks and Coverage	Functionality and behavior analysis that use coverage, covergroup, procedural code, or assertions in a OVM class-based monitor or SystemVerilog interface
Component	The fundamental building block used to create each element of an OVC. Each component (for example, driver, agent, and so on) is derived from the `ovm_component` base class.
Data item	A transaction object generated as stimulus in a verification environment
Driver	A verification component that connects at the pin-level interface to the DUT. It contains one or more transaction-level interfaces to communicate with other transaction-level components in the verification environment.
DUT	Device Under Test. The design (block, subsystem, or system) being verified, which may be a combination of hardware and software.
Env	The environment or “env” is the top-level component of the OVC. It contains one or more agents, as well as other top-level components such as a bus monitor. The environment is configurable to enable reuse. For example, active agents can be changed to be passive when the verification environment is reused in system verification.
Exhaustive sequence	A sequence that randomly selects each sequence that is available in the sequencer and executes it one time
Interface OVC	A reusable verification component focusing on a specific protocol such as PCI, TCP/IP, Ethernet, and so on
Late randomization	Postponement of the generation and randomization of data until the time that it is passed to the DUT

Term	Definition
Layered sequencer	A sequencer that is used in place of a driver at a given layer of a protocol. Layered sequencers execute a sequence that converts a data item into one or more transactions at the lower layer of the protocol. They may also execute other sequences to generate additional streams of lower-layer items in parallel.
Monitor	A verification component that monitors signal-level behavior and communicates transactions to other components in the verification environment. A monitor may also perform specific checking and/or functional coverage gathering as needed.
OVC	OVM Verification Component. An OVC is an encapsulated, reusable, and configurable verification component for an interface protocol, a design sub-module, or a full system.
Public interface	An application programming interface (API) declared as public
Random sequence	A sequence that selects at random one of the sequences available for a specific sequencer and executes it
Sequence	A basic construct associated with a sequencer. Sequences generate data items and other sequences (subsequences) and drive one or more transactions to the DUT via the driver in an OVC. This construct can also be referred to as a driver sequence.
Sequence item	A data item (that is, transaction) generated by a sequence. This item is typically provided to a driver by a sequencer. For layering of stimulus, different data items can be defined for each layer. Lower-layer objects can be provided items by the upper-layer objects.
Sequence library	A collection of sequences used by a sequencer
Sequencer	A verification component that mediates the generation and flow of data between sequences and a driver. The sequencer has a collection of sequences associated with it called a sequence library. This type of component is also referred to as a driver sequencer.
Simple sequence	A sequence that generates a single random data item
Subsequencer	A sequencer that is accessed by a virtual sequencer’s virtual sequence
Test	A class that encapsulates test-specific instructions from the test writer
Testbench (Env)	The top-level container where reusable verification environments are constructed.

Term	Definition
TLM	Transaction-Level Modeling. TLM interfaces provide a standard method for components to exchange transactions instead of signals. Components are characterized by their TLM interfaces and may only be connected to other components with compatible interfaces.
Virtual sequence	Any sequence that coordinates the activity of other sequences in one or more sequencers is referred to as a virtual sequence. Virtual sequences enable centralized data-flow control on multiple interfaces.

Conventions in This Manual

Typeface	Represents
`courier font`	Indicates code. For example: `class simple_item extends ovm_sequence_item;`

`courier bold`	Used to highlight important sections of code. For example:
	`set_type_override``("uart_frame", "short_delay_frame");`
	Also used to identify user input. The following example indicates that you enter the **run** command, in response to which the system displays the results of the command:
	`sim-prompt>` **`run`**
	`SVSEED default: 1`
	`OVM_INFO @ 0 [RNTST] Running test ...`

italic	The italic font represents variables that you must define. For example, the following sentence indicates that *count* is a variable to which you must assign a value: The number of sequences executed depends on the *`count`* field of the sequencer.
`sim-prompt>`	Denotes the prompt for the simulator you are running. For example: `sim-prompt>` **`run`**
>	Denotes a third-party simulator prompt
%	Denotes the UNIX prompt

OVM Overview

This chapter describes:

- How to use the Open Verification Methodology (OVM) for creating SystemVerilog testbenches.
- The recommended architecture of an OVM Verification Component (OVC).

This chapter contains the following sections:

Introduction to OVM

OVM and Coverage Driven Verification (CDV)

OVM provides the best framework to achieve coverage-driven verification (CDV). CDV combines automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time spent verifying a design. The purpose of CDV is to:

- Eliminate the effort and time spent creating hundreds of tests.
- Ensure thorough verification using up-front goal setting.
- Receive early error notifications and deploy run-time checking and error analysis to simplify debugging.

The CDV flow is different than the traditional directed-testing flow. With CDV, you start by setting verification goals using an organized planning process. You then create a smart testbench that generates legal stimuli and sends it to the DUT. Coverage monitors are added to the environment to measure progress and identify non-exercised functionality. Checkers are added to identify undesired DUT behavior. Simulations are launched after both the coverage model and testbench have been implemented. Verification then can be achieved.

Using CDV, you can thoroughly verify your design by changing testbench parameters or changing the randomization seed. Test constraints can be added on top of the smart infrastructure to tune the simulation to meet verification goals sooner. Ranking technology allows you to identify the tests and seeds that contribute to the verification goals, and to remove redundant tests from a test-suite regression.

CDV environments support both directed and constrained-random testing. However, the preferred approach is to let constrained-random testing do most of the work before devoting effort to writing time-consuming, deterministic tests to reach specific scenarios that are too difficult to reach randomly.

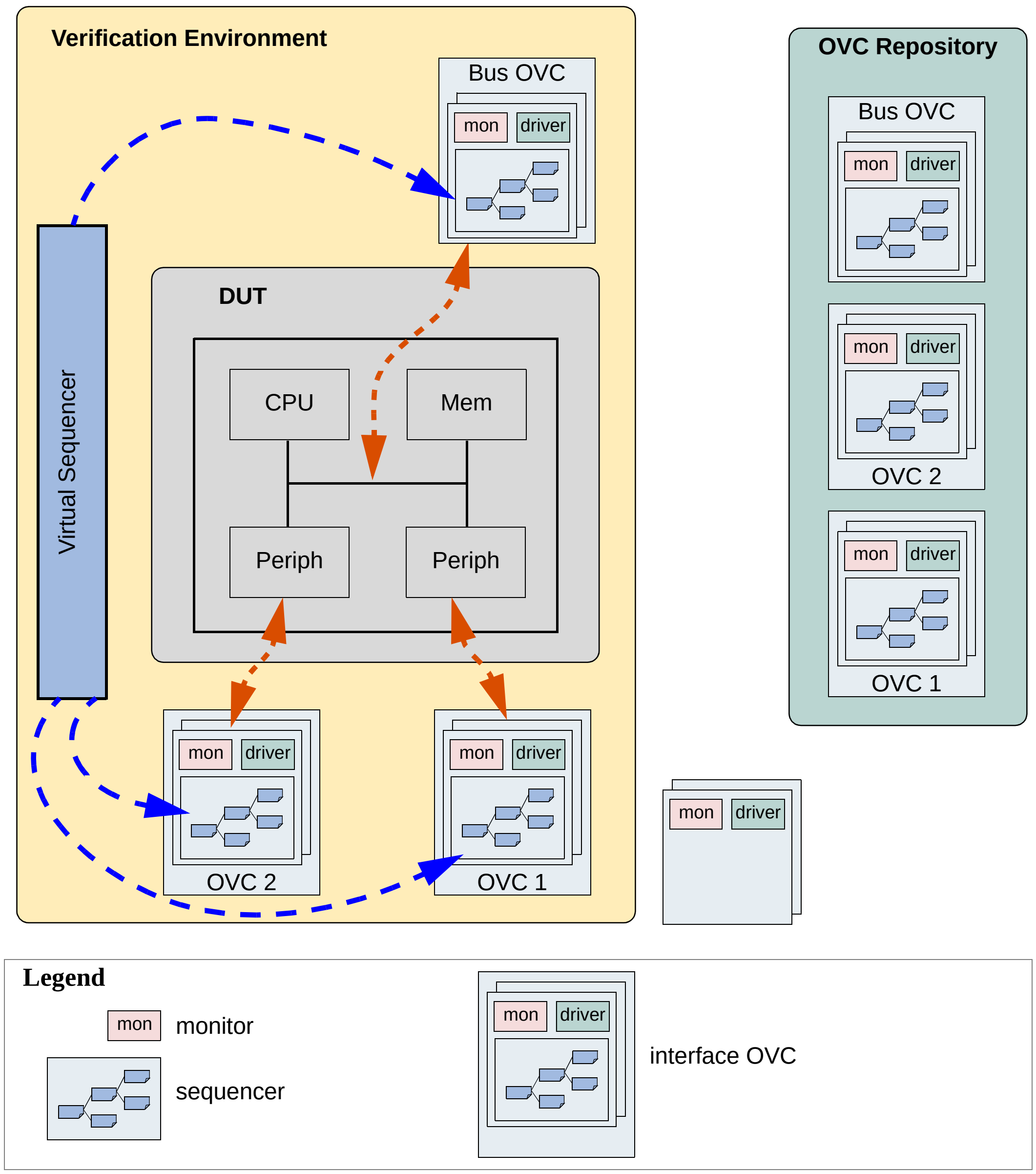
Significant efficiency and visibility into the verification process can be achieved by proper planning. Creating an executable plan with concrete metrics enables you to accurately measure progress and thoroughness throughout the design and verification project. By using this method, sources of coverage can be planned, observed, ranked, and reported at the feature level. Using an abstracted, feature-based approach (and not relying on implementation details) enables you to have a more readable, scalable, and reusable verification plan.

OVM Testbench and Environments

An OVM testbench is composed of reusable verification environments called OVM verification components (OVCs). An OVC is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design sub-module, or a full system. Each OVC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting coverage information for a specific protocol or design. The OVC is applied to the device under test (DUT) to verify your implementation of the protocol or design architecture. OVCs expedite creation of efficient testbenches for your DUT and are structured to work with any hardware description language (HDL) and high-level verification language (HVL) including Verilog, VHDL, ***e***, SystemVerilog, and SystemC.

Figure 2-1 on page 15 shows an example of a verification environment with three interface OVCs. These OVCs might be stored in a company repository and reused for multiple verification environments. The interface OVC is instantiated and configured for a desired operational mode. The verification environment also contains a multi-channel sequence mechanism (that is, virtual sequencer) which synchronizes the timing and the data between the different interfaces and allows fine control of the test environment for a particular test.

Figure 2-1 Verification Environment Example



OVC Overview

The following subsections describe the components of an OVC:

- “Data Item (Transaction)” on page 16
- “Driver (BFM)” on page 16

- "Sequencer" on page 16
- "Monitor" on page 17
- "Agent" on page 18
- "Environment" on page 18

Data Item (Transaction)

Data items represent the input to the DUT. Examples include networking packets, bus transactions, and instructions. The fields and attributes of a data item are derived from the data item's specification. For example, the Ethernet protocol specification defines valid values and attributes for an Ethernet data packet. In a typical test, many data items are generated and sent to the DUT. By intelligently randomizing data item fields using SystemVerilog constraints, you can create a large number of meaningful tests and maximize coverage.

Driver (BFM)

A driver is an active entity that emulates logic that drives the DUT. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving the DUT signals. (If you have created a verification environment in the past, you probably have implemented driver functionality.) For example, a driver controls the read/write signal, address bus, and data bus for a number of clocks cycles to perform a write transfer.

Sequencer

A sequencer is an advanced stimulus generator that controls the items that are provided to the driver for execution. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior allows you to add constraints to the data item class in order to control the distribution of randomized values. Unlike generators that randomize arrays of transactions or one transaction at a time, a sequencer captures important randomization requirements out-of-the-box. A partial list of the sequencer's built-in capabilities includes:

- Ability to react to the current state of the DUT for every data item generated.
- Captures the order between data items in user-defined sequences, which forms a more structured and meaningful stimulus pattern.
- Enables time modeling in reusable scenarios.
- Supports declarative and procedural constraints for the same scenario.

- Allows system-level synchronization and control of multiple interfaces.

For more information about creating and using sequencers, refer to the *SystemVerilog OVM Class Reference* and to the following sections in this manual:

- "Enabling Scenario Creation" on page 56.
- "Using Sequences" on page 83.
- "Creating a Virtual Sequence" on page 92.

Sequencers also can be layered on top of each other to model protocol layering. Refer to "Using Layered Sequencers" on page 125 for more information.

Monitor

A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information and perform checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitors are used instead. A monitor:

- Collects transactions (data items). A monitor extracts signal information from a bus and translates the information into a transaction that can be made available to other components and to the test writer.
- Extracts events. The monitor detects the availability of information (such as a transaction), structures the data, and emits an event to notify other components of the availability of the transaction. A monitor also captures status information so it is available to other components and to the test writer.
- Performs checking and coverage.
  - Checking typically consists of protocol and data checkers to verify that the DUT output meets the protocol specification.
  - Coverage also is collected in the monitor.
- Optionally prints trace information.

A bus monitor handles all the signals and transactions on a bus, while an agent monitor handles only signals and transactions relevant to a specific agent.

Typically, drivers and monitors are built as separate entities (even though they may use the same signals) so they can work independently of each other. However, you can reuse code that is common between a driver and a monitor to save time.

Note: Do not have monitors depend on drivers for information so that an agent can operate passively when only the monitor is present.

Agent

Sequencers, drivers, and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by the test writer, OVM recommends that environment developers create a more abstract container called an agent. Agents can emulate and verify DUT devices. They encapsulate a driver, sequencer, and monitor. OVCs can contain more than one agent. Some agents (for example, master or transmit agents) initiate transactions to the DUT, while other agents (slave or receive agents) react to transaction requests. Agents should be configurable so that they can be either active or passive. Active agents emulate devices and drive transactions according to test directives. Passive agents only monitor DUT activity.

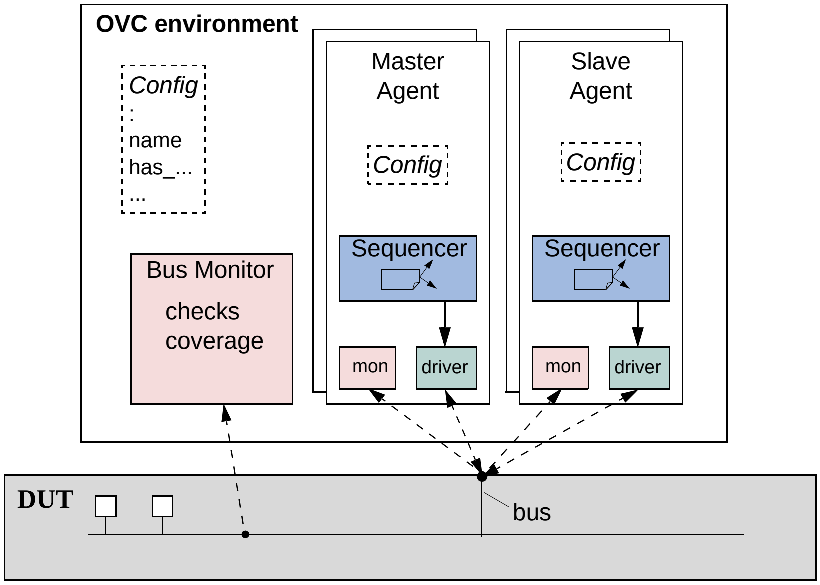
Environment

The environment (env) is the top-level component of the OVC. It contains one or more agents, as well as other components such as a bus monitor. The env contains configuration properties that enable you to customize the topology and behavior and make it reusable. For example, active agents can be changed into passive agents when the verification environment is reused in system verification. Figure 2-2 on page 19 illustrates the structure of a reusable verification environment. Notice that an OVC may contain an environment-level monitor. This bus-level monitor performs checking and coverage for activities that are not necessarily related to a single agent. An agent's monitors can leverage data and events collected by the global monitor.

The environment class (`ovm_env`) is architected to provide a flexible, reusable, and extendable verification component. The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage.

You can use derivation to specialize the existing classes to their specific protocol. This manual describes the process and infrastructure that OVM provides to replace existing component behavior with IP-specific behavior.

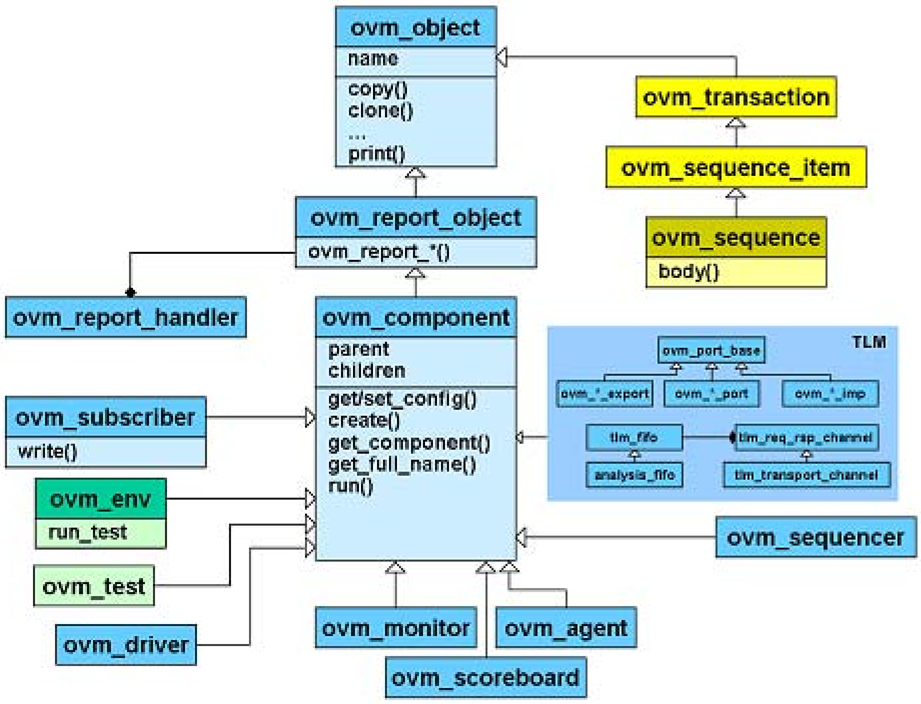
Figure 2-2 Typical OVC Environment



The SystemVerilog OVM Class Library

The SystemVerilog OVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments (see Figure 2-3 on page 20). The library consists of base classes, utilities, and macros. Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. These phases are defined in the base class library but can be extended to meet specific project needs. See the *SystemVerilog OVM Class Reference* for more information.

Figure 2-3 (Partial) OVM Class Hierarchy

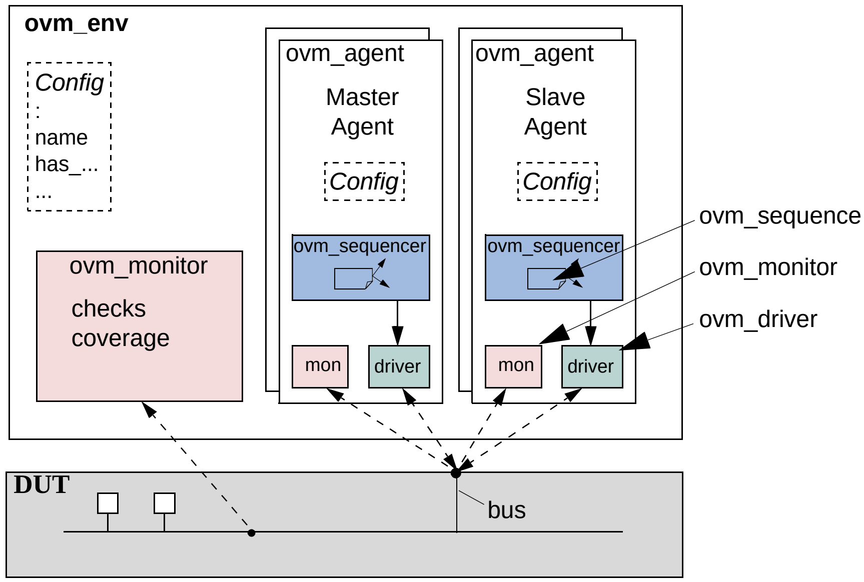


The advantages of using the SystemVerilog OVM Class Library include:

- A robust set of built-in features—The SystemVerilog OVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory methods, and more.

- Correctly-implemented OVM concepts—Each component in the block diagram in Figure 2-2 on page 19 is derived from a corresponding SystemVerilog OVM Class Library component. Figure 2-4 on page 21 shows the same diagram using the derived SystemVerilog OVM Class Library base classes. Using these base-class elements increases the readability of your code since each component's role is predetermined by its parent class.

Figure 2-4 Typical OVM Environment Using OVM Library Classes



Other OVM Facilities

The SystemVerilog OVM Class Library also provides various utilities to simplify the development and use of verification environments. These utilities support debugging by providing a user-controllable messaging utility. They support development by providing a standard communication infrastructure between verification components (TLM) and flexible verification environment construction (OVM factory).

The SystemVerilog OVM Class Library provides global messaging facilities that can be used for failure reporting and general reporting purposes. Both messages and reporting are important aspects of ease of use.

This section includes the following:

- "OVM Factory" on page 21
- "Transaction-Level Modeling" on page 22

OVM Factory

The factory method is a classic software design pattern that is used to create generic code, deferring to run time the exact specification of the object that will be created. In functional verification, introducing class variations is frequently needed. For example, in many tests you might want to derive from the generic data item definition and add more constraints or fields to it; or you might want to use the new derived class in the entire environment or only in a single interface; or perhaps you must modify the way data is sent to the DUT by deriving a new driver. The factory allows you to substitute the verification component without having to provide a derived version of the parent component as well.

The SystemVerilog OVM Class Library provides a built-in central factory that allows:

- Controlling object allocation in the entire environment or for specific objects.
- Modifying stimulus data items as well as infrastructure components (for example, a driver).

Use of the OVM built-in factory reduces the effort of creating an advanced factory or implementing factory methods in class definitions. It facilitates reuse and adjustment of predefined verification IP in the end-user's environment. One of the biggest advantages of the factory is that it is transparent to the test writer and reduces the object-oriented expertise required from both developers and users.

Transaction-Level Modeling

OVM components communicate via standard TLM interfaces, which improves reuse. Using a SystemVerilog implementation of TLM in OVM, a component may communicate via its interface to any other component that implements that interface. Each TLM interface consists of one or more methods used to transport data. TLM specifies the required behavior (semantic) of each method but does not define their implementation. Classes inheriting a TLM interface must provide an implementation that meets the specified semantic. Thus, one component may be connected at the transaction level to others that are implemented at multiple levels of abstraction. The common semantics of TLM communication permit components to be swapped in and out without affecting the rest of the environment.

Transaction-Level Modeling (TLM)

Transaction-Level Modeling Overview

One of the keys to verification productivity is to think about the problem at a level of abstraction that makes sense. When verifying a device under test (DUT) that handles packets flowing back and forth, or processes instructions, or performs other types of functionality, you must create a verification environment that supports the appropriate abstraction level. While the actual interface to the DUT ultimately is represented by signal-level activity, experience has shown that it is necessary to manage most of the verification tasks, such as generating stimulus and collecting coverage data, at the transaction level, which is the natural way engineers tend to think of the activity of a system.

OVM provides a set of transaction-level communication interfaces and channels that you can use to connect components at the transaction level. The use of TLM interfaces isolates each component from changes in other components throughout the environment. When coupled with the phased, flexible build infrastructure in OVM, TLM promotes reuse by allowing any component to be swapped for another, as long as they have the same interfaces. This concept also allows OVM verification environments to be assembled with a transaction-level model of the DUT, and the environment to be reused as the design is refined to RTL. All that is required is to replace the transaction-level model with a thin layer of compatible components to convert between the transaction-level activity and the pin-level activity at the DUT.

The well-defined semantics of TLM interfaces between components also provide the ideal platform for implementing mixed-language verification environments. In addition, TLM provides the basis for easily encapsulating components into reusable components, called OVM verification components (OVCs), to maximize reuse and minimize the time and effort required to build a verification environment.

This chapter discusses the essential elements of transaction-level communication in OVM, and illustrates the mechanics of how to assemble transaction-level components into a verification environment. Later in this document we will discuss additional concerns in order to address a wider set of verification issues. For now, it is important to understand these foundational concepts first.

TLM Basics

Before you can fully understand how to model verification at the transaction level, you must understand what a transaction is.

Transactions

In OVM, a transaction is a class object, `ovm_transaction` (extended from `ovm_object`), that includes whatever information is needed to model a unit of communication between two components. In the most basic example, a simple bus protocol transaction would be modeled as follows:

```
class simple_trans extends ovm_transaction;
  rand data_t data;
  rand addr_t addr;
  rand enum {WRITE,READ} kind;
  constraint c1 { addr < 16'h2000; }
  ...
endclass
```

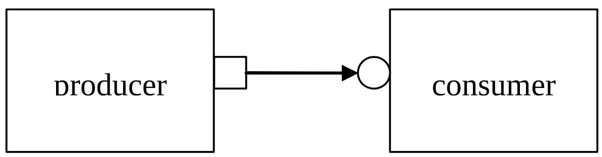
The transaction object includes variables, constraints, and other fields and methods necessary for generating and operating on the transaction. Obviously, there is often more than just this information that is required to fully specify a bus transaction. The amount and detail of the information encapsulated in a transaction is an indication of the abstraction level of the model. For example, the `simple_trans` transaction above could be extended to include more information, such as the number of wait states to inject, the size of the transfer, or any number of other properties. The transaction could also be extended to include additional constraints. It is also possible to define higher-level transactions that include some number of lower-level transactions. Transactions can thus be composed, decomposed, extended, layered, and otherwise manipulated to model whatever communication is necessary at any level of abstraction.

Transaction-Level Communication

Transaction-level interfaces define a set of methods that use transaction objects as arguments. A TLM *port* defines the set of methods (the API) to be used for a particular connection, while a TLM *export* supplies the implementation of those methods. Connecting a port to an export allows the implementation to be executed when the port method is called.

Basic TLM Communication

Figure 3-1 Simple Producer/Consumer



The most basic transaction-level operation allows one component to *put* a transaction to another. Consider Figure 3-1 on page 24.

The square box on the producer indicates a port, and the circle on the consumer indicates the export. The producer generates transactions and sends them out its `put_port`:

```
class producer extends ovm_component;
  ovm_blocking_put_port #(simple_trans) put_port; // 1 parameter
  function new( string name, ovm_component parent);
    put_port = new(“put_port”, this);
    ...
  endfunction
  virtual task run();
    simple_trans t;
    for(int i = 0; i < N; i++) begin
      // Generate t.
      put_port.put(t);
    end
  endtask
```

Note: The `ovm_*_port` is parameterized by the transaction type that will be communicated. This may either be specified directly, or it may be a parameter of the parent component.

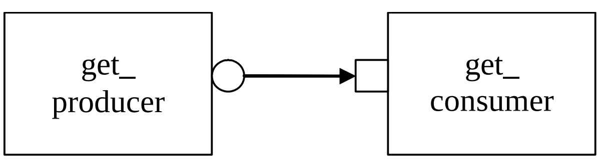
The actual implementation of the put() call is supplied by the consumer.

```
class consumer extends ovm_component;
  ovm_blocking_put_imp #(simple_trans, consumer) put_export; // 2 parameters
  ...
  task put(simple_trans t);
    case(t.kind)
      READ: // Do read.
      WRITE: // Do write.
    endcase
  endtask
endclass
```

Note: The ovm_*_imp takes two parameters: the type of the transaction and the type of the object that declares the method implementation.

Note: The semantics of the put operation are defined by TLM. In this case, the `put()` call in the producer will block until the consumer’s `put` implementation is complete. Other than that, the operation of `producer` is completely independent of the `put` implementation (`ovm_put_imp`). In fact, `consumer` could be replaced by another component that also implements `put` and `producer` will continue to work in exactly the same way. The modularity provided by TLM fosters an environment in which components may be easily reused since the interfaces are well defined.

Figure 3-2 Consumer gets from Producer



The converse operation to put is *get*. Consider Figure 3-2 on page 25.

In this case, the consumer requests transactions from the producer via its get port:

```
class get_consumer extends ovm_component;
  ovm_blocking_get_port #(simple_trans) get_port;
  function new( string name, ovm_component parent);
    get_port = new("get_port", this);
    ...
  endfunction
  virtual task run();
    simple_trans t;
    for(int i = 0; i < N; i++) begin
      // Generate t.
      get_port.get(t);
    end
  endtask
```

The `get()` implementation is supplied by the producer.

```
class get_producer extends ovm_component;
  ovm_blocking_get_imp #(simple_trans, get_producer) get_export;
  ...
  task get(output simple_trans t);
    simple_trans tmp = new();
    // Assign values to tmp.
    t = tmp;
  endtask
endclass
```

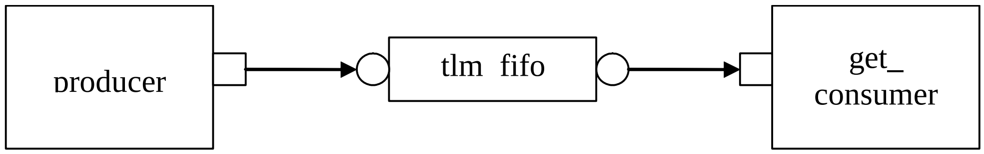
As with `put()` above, the `get_consumer`'s `get()` call will block until the `get_producer`'s method completes. In TLM terms, `put()` and `get()` are *blocking* methods.

Note: In both these examples there is a single process running, with control passing from the port to the export and back again. The direction of data flow (from producer to consumer) is the same in both examples.

Communicating Between Processes

In the basic put example above, the consumer will only be active when its `put()` method is called. In many cases, it may be necessary for components to operate independently, where the producer is creating transactions in one process while the consumer needs to operate on those transactions in another. OVM provides the `tlm_fifo` channel to facilitate such communication. The `tlm_fifo` implements all of the TLM interface methods, so the producer puts the transaction into the `tlm_fifo`, while the consumer independently gets the transaction from the fifo, as shown in Figure 3-3 on page 27.

Figure 3-3 Using a tlm_fifo



When the producer puts a transaction into the fifo, it will block if the fifo is full, otherwise it will put the object into the fifo and return immediately. The get operation will return immediately if a transaction is available (and will then be removed from the fifo), otherwise it will block until a transaction is available. Thus, two consecutive `get()` calls will yield different transactions to the consumer. The related `peek()` method returns a copy of the available transaction without removing it. Two consecutive `peek()` calls will return copies of the same transaction.

Blocking vs. Nonblocking

The interfaces that we have looked at so far are *blocking*. That means that the tasks block execution until they complete. They are not allowed to fail. There is no mechanism for any blocking call to terminate abnormally or otherwise alter the flow of control. They simply wait until the request is satisfied. In a timed system, this means that time may pass between the time the call was initiated and the time it returns.

In contrast, a *nonblocking* call returns immediately. The semantics of a nonblocking call guarantee that the call returns in the same delta cycle in which it was issued, that is, without consuming any time, not even a single delta cycle. In OVM, nonblocking calls are modeled as functions.

```
class consumer extends ovm_component;
  ovm_get_port #(simple_trans) get_port;
  task run;
        ...
        for(int i=0; i<10; i++)
        if(get_port.try_get(t))
        //Do something with t.
        ...
      endtask
endclass
```

If a transaction exists, it will be returned in the argument and the function call itself will return TRUE. If no transaction exists, the function will return FALSE. Similarly, with `try_peek()`. The `try_put()` method will return TRUE if the transaction is sent.

Connecting Transaction-Level Components

With ports and exports defined for transaction-level components, the actual connection between them is accomplished via the `connect()` method in the parent (component or env), with an argument that is the object (port or export) to which it will be connected. In a verification environment, the series of `connect()` calls between ports and exports establishes a netlist of peer-to-peer and hierarchical connections, ultimately terminating at an implementation of the agreed-upon interface. The resolution of these connections causes the collapsing of the netlist, which results in the initiator's port being assigned to the target's implementation. Thus, when a component calls

```
put_port.put(t);
```

the connection means that it actually calls

```
target.put_export.put(t);
```

where `target` is the connected component.

Peer-to-Peer connections

When connecting components at the same level of hierarchy, ports are always connected to exports. All `connect()` calls between components will be done in the parent's `connect()` method.

```
class my_env extends ovm_env;
  ...
  virtual function void connect();
    // component.port.connect(target.export);
    producer.blocking_put_port.connect(fifo.put_export);
    get_consumer.get_port.connect(fifo.get_export);
    ...
  endfunction
endclass
```

Port/Export Compatibility

Another advantage of TLM communication in OVM is that all TLM connections are checked for compatibility before the test runs. In order for a connection to be valid, the export must provide implementations for *at least* the set of methods defined by the port, and the transaction type parameter for the two must be identical. For example, a `blocking_put_port`, which requires an implementation of `put()` may be connected to either a `blocking_put_export` or a `put_export`. Both exports supply an implementation of `put()`, although the `put_export` also supplies implementations of `try_put()` and `can_put()`.

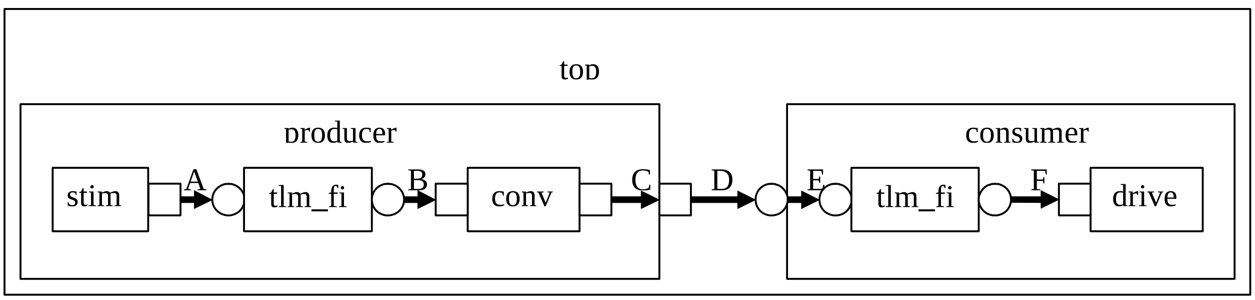
Encapsulation and Hierarchy

The use of TLM interfaces isolates each component in a verification environment from the others. The environment instantiates a component and connects its ports/exports to its neighbor(s), independent of any further knowledge of the specific implementation. Smaller components may be grouped hierarchically to form larger components (see Developing Reusable Open Verification Components (OVCs) on page 35). Access to child components is achieved by making their interfaces visible at the parent level. At this level, the parent simply looks like a single component with a set of interfaces on it, regardless of its internal implementation.

Hierarchical Connections

Making connections across hierarchical boundaries involves some additional issues, which are discussed in this section. Consider the hierarchical design shown in Figure 3-4 on page 29.

Figure 3-4 Hierarchy in TLM



The hierarchy of this design contains two components, `producer` and `consumer`. `producer` contains three components, `gen`, `fifo`, and `conv`. `consumer` contains two components, `fifo` and `driver`. Notice that, from the perspective of `top`, the producer and consumer appear identical to those in Figure 3-1 on page 24, in which the producer's `put_port` is connected to the consumer's `put_export`. The two FIFOs are both unique instances of the same `tlm_fifo` component.

In Figure 3-4 on page 29, connections A, B, D, and F are standard peer-to-peer connections as discussed above. As an example, connection A would be coded in the producer's `connect()` method as:

```
gen.put_port.connect(fifo.put_export);
```

Connections C and E are of a different sort than what have been shown. Connection C is a port-to-port connection, and connection E is an export-to-export connection. These two kinds of connections are necessary to complete hierarchical connections. Connection C *imports* a port from the outer component to the inner component. Connection E *exports* an export upwards in the hierarchy from the inner component to the outer one. Ultimately, every transaction-level connection must resolve so that a port is connected to an export. However, the port and export terminals do not need to be at the same place in the hierarchy. We use port-to-port and export-to-export connections to bring connectors to a hierarchical boundary to be accessed at the next-higher level of hierarchy.

For connection E, the implementation resides in the fifo and is exported up to the interface of consumer. All export-to-export connections in a parent component are of the form

```
export.connect(subcomponent.export)
```

so connection E would be coded as:

```
class consumer extends ovm_component;
  ovm_put_export #(trans) put_export;
  tlm_fifo #(trans) fifo;
  ...
  function void connect();
    put_export.connect(fifo.put_export); // E
    bfm.get_port.connect(fifo.get_export); // F
endfunction
...
endclass
```

Conversely, port-to-port connections are of the form

```
subcomponent.port.connect(port);
```

so connection C would be coded as:

```
class producer extends ovm_component;
  ovm_put_port #(trans) put_port;
  conv c;
  ...
  function void connect();
    c.put_port.connect(put_port);
    ...
  endfunction
```

The following table summarizes connection types and elaboration functions.

connection type	**connect() form**
port-to-export	`comp1.port.connect(comp2.export);`
port-to-port	`subcomponent.port.connect(port);`
export-to-export	`export.connect(subcomponent.export);`

Note: The argument to the `port.connect()` method may be either an export or a port, depending on the nature of the connection (that is, peer-to-peer or hierarchical). The argument to `export.connect()` is always an export of a child component.

Analysis Communication

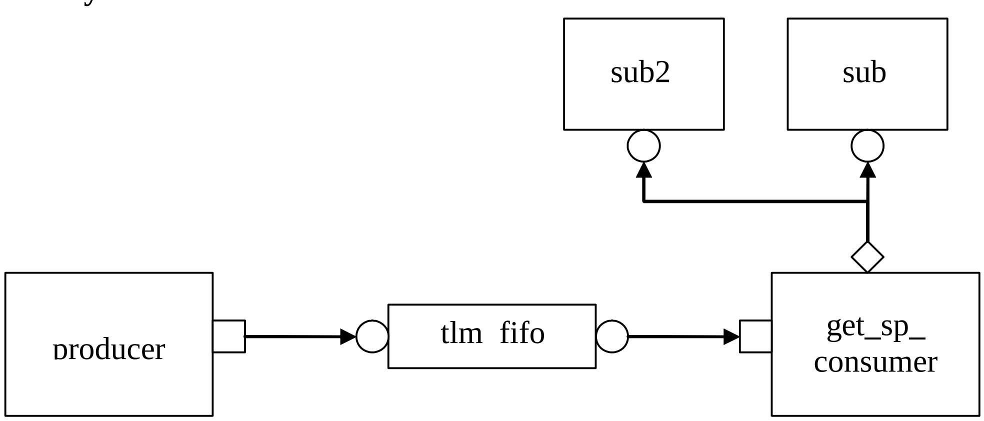
The put/get communication as described above allows verification components to be created that model the “operational” behavior of a system. Each component is responsible for communicating through its TLM interface(s) with other components in the system in order to stimulate activity in the DUT and/or respond its behavior. In any reasonably complex verification environment, however, particularly where randomization is applied, a collected transaction should be distributed to the rest of the environment for end-to-end checking (scoreboard), or additional coverage collection.

The key distinction between the two types of TLM communication is that the put/get ports typically require a corresponding export to supply the implementation. For analysis, however, the emphasis is on a particular component, such as a monitor, being able to produce a stream of transactions, regardless of whether there is a target actually connected to it. Modular analysis components are then connected to the analysis_port, each of which processes the transaction stream in a particular way.

Analysis Ports

The `ovm_analysis_port` (represented as a diamond on the monitor in Figure 3-5 on page 31) is a specialized TLM port whose interface consists of a single function, `write()`. The analysis port contains a list of analysis_exports that are connected to it. When the component calls `analysis_port.write()`, the analysis_port cycles through the list and calls the `write()` method of each connected export. If nothing is connected, the `write()` call simply returns. Thus, an analysis port may be connected to 0, 1, or many analysis exports, but the operation of the component that writes to the analysis port does not depend on the number of exports connected. Because `write()` is a void function, the call will always complete in the same delta cycle, regardless of how many components (for example, scoreboards, coverage collectors, and so on) are connected.

Figure 3-5 Analysis Communication




```
class get_ap_consumer extends get_consumer;
  ovm_analysis_port #(my_trans) ap;
  function new(...);
```

```
        super.new()
        ap = new("analysis_port", this);
        ...
      endfunction
        task run;
            ...
            for(int i=0; i<10; i++)
              if(get_port.try_get(t)) begin
                //Do something with t.
                ap.write(t); // Write transaction.
                ...
              end
        endtask
```

In the parent environment, the analysis port gets connected to the analysis export of the desired components, such as coverage collectors and scoreboards.

Analysis Exports

As with other TLM connections, it is up to each component connected to an analysis port to provide an implementation of `write()` via an analysis_export. OVM provides the `ovm_subscriber` base component to simplify this operation, so a typical analysis component would extend `ovm_subscriber` as:

```
class sub1 #(type T = simple_trans) extends ovm_subscriber #(T);
...
  function void write(T t);
    // Record coverage information of t.
  endfunction
endclass
```

As with `put()` and `get()` described above, the TLM connection between an analysis port and export, allows the export to supply the implementation of `write()`. If multiple exports are connected to an analysis port, the port will call the `write()` of each export, in order. Since all implementations of `write()` must be functions, the analysis port's `write()` function completes immediately, regardless of how many exports are connected to it.

```
class my_env extends ovm_env;
  get_ap_component g;
  sub1 s1;
  sub2 s2;
  ...
  function void connect();
    g.ap.connect(s1.analysis_export);
    g.ap.connect(s2.analysis_export);
    ...
  endfunction
endclass
```

When multiple subscribers are connected to an analysis_port, each is passed a pointer to the same transaction object, the argument to the `write()` call. Each `write()` implementation must make a local copy of the transaction and then operate on the copy to avoid corrupting the transaction contents for any other subscriber that may have received the same pointer.

OVM also includes an analysis_fifo, which is a tlm_fifo that also includes an analysis export, to allow blocking components access to the analysis transaction stream. The analysis_fifo is unbounded, so the monitor's `write()` call is guaranteed to succeed immediately. The analysis component may then get the transactions from the analysis_fifo at its leisure.

Developing Reusable Open Verification Components (OVCs)

This chapter describes the basic concepts and components that make up a typical verification environment. It also shows how to combine these components using a proven hierarchical architecture to create reusable OVCs. The sections in this chapter follow the same order you should follow when developing an OVC:

Note: This chapter builds upon concepts described in OVM Overview on page 13 and Transaction-Level Modeling (TLM) on page 23.

Modeling Data Items for Generation

Data items:

- Are transaction objects used as stimulus to the device under test (DUT).
- Represent transactions that are processed by the verification environment.
- Are classes that you define ("user-defined" classes).
- Capture and measure transaction-level coverage and checking.

Note: The SystemVerilog OVM Class Library provides the `ovm_sequence_item` base class. Every user-defined data item must be derived directly or indirectly from this base class.

To create a user-defined data item:

1. Review your DUT's transaction specification and identify the application-specific properties, constraints, tasks, and functions.
2. Derive a data item class from the `ovm_sequence_item` base class (or a derivative of it).
3. Define a constructor for the data item.
4. Add control fields ("knobs") for the items identified in Step 1 to enable easier test writing.
5. Use OVM field macros to enable printing, copying, comparing, and so on.

OVM has built-in automation for many service routines that a data item needs. For example, you can use:

- `print()` to print a data item.
- `copy()` to copy the contents of a data item.
- `compare()` to compare two similar objects.

OVM allows you to specify the automation needed for each field and to use a built-in, mature, and consistent implementation of these routines.

To assist in debugging and tracking transactions, the `ovm_transaction` base class includes the `m_transaction_id` field. In addition, the `ovm_sequence_item` base class (extended from `ovm_transaction`) also includes the `m_sequence_id` field, allowing sequence items to be correlated to the sequence that generated them originally. This is necessary to allow the sequencer to route response transactions back to the correct sequence in bidirectional protocols.

The class `simple_item` in this example defines several random variables and class constraints. The OVM macros implement various utilities that operate on this class, such as copy, compare, print, and so on. In particular, the `` `ovm_object_utils `` macro registers the class type with the common factory.

```
class simple_item extends ovm_sequence_item;
  rand int unsigned addr;
  rand int unsigned data;
  rand int unsigned delay;
  constraint c1 { addr < 16'h2000; }
  constraint c2 { data < 16'h1000; }
  // OVM automation macros for general objects
  `ovm_object_utils_begin(simple_item)
    `ovm_field_int(addr, OVM_ALL_ON)
    `ovm_field_int(data, OVM_ALL_ON)
    `ovm_field_int(delay, OVM_ALL_ON)
```

```
12   `ovm_object_utils_end
13   // Constructor
14   function new (string name = "simple_item");
15     super.new(name);
16   endfunction : new
17 endclass : simple_item
```

Line 1 Derive data items from `ovm_sequence_item` so they can be generated in a procedural sequence. See “Generating Stimulus with Sequences and Sequence Items” on page 58 for more information.

Lines 5-6 Add constraints to a data item definition in order to:

- Reflect specification rules. In this example, the address must be less than 16'h2000.
- Specify the default distribution for generated traffic. For example, in a typical test most transactions should be legal.

Lines 7-12 Use the OVM macros to automatically implement functions such as `copy()`, `compare()`, `print()`, `pack()`, and so on. Refer to OVM Macros in the *SystemVerilog OVM Class Reference* for information on the `` `ovm_object_utils_begin, `ovm_object_utils_end, `ovm_field_* ``, and their associated macros.

Note: OVM provides built-in macros to simplify development of the verification environment. The macros automate the implementation of functions defined in the base class, such as `copy()`, `compare()`, and `print()`, thus saving many lines of code. Use of these macros is optional but recommended.

Inheritance and Constraint Layering

In order to meet verification goals, the OVC user might need to adjust the data-item generation by adding more constraints to a class definition. In SystemVerilog, this is done using inheritance. The following example shows a derived data item, `word_aligned_item`, which includes an additional constraint to select only word-aligned addresses.

```
class word_aligned_item extends simple_item;
  constraint word_aligned_addr { addr[1:0] == 2'b00; }
  `ovm_object_utils(word_aligned_item)
  // Constructor
  function new (string name = "word_aligned_item");
    super.new(name);
  endfunction : new
endclass : word_aligned_item
```

To enable this type of extensibility:

- The base class for the data item (`simple_item` in this chapter) should use virtual methods to allow derived classes to override functionality.

- Make sure constraint blocks are organized so that they are able to override or disable constraints for a random variable without having to rewrite a large block.
- Do not use the `protected` or `local` keyword to restrict access to properties that may be constrained by the user. This will limit your ability to constrain them with an inline constraint.

Defining Control Fields ("Knobs")

The generation of all values of the input space is often impossible and usually not required. However, it is important to be able to generate a few samples from ranges or categories of values. In the `simple_item` example in Modeling Data Items for Generation on page 35 above, the delay property could be randomized to anything between zero and the maximum unsigned integer. It is not necessary (nor practical) to cover the entire legal space, but it is important to try back-to-back items along with short, medium, and large delays between the items, and combinations of all of these.To do this, define control fields (often called "knobs") to enable the test writer to control these variables. These same control knobs can also be used for coverage collection. For readability, use enumerated types to represent various generated categories.

Knobs Example

```
typedef enum {ZERO, SHORT, MEDIUM, LARGE, MAX} simple_item_delay_e;
class simple_item extends ovm_sequence_item;
  rand int unsigned addr;
  rand int unsigned data;
  rand int unsigned delay;
  rand simple_item_delay_e delay_kind; // Control field
  // OVM automation macros for general objects
  `ovm_object_utils_begin(simple_item)
    `ovm_field_int(addr, OVM_ALL_ON)
    `ovm_field_enum(delay_kind, simple_item_delay_e, OVM_ALL_ON)
  `ovm_object_utils_end
  constraint delay_order_c { solve delay_kind before delay; }
  constraint delay_c {
    (delay_kind == ZERO) -> delay == 0;
    (delay_kind == SHORT) -> delay inside { [1:10] }
    (delay_kind == MEDIUM) -> delay inside { [11:99] }
    (delay_kind == LONG) -> delay inside { [100:999] }
    (delay_kind == MAX ) -> delay == 1000;
    delay >=0; delay <= 1000; }
endclass : simple_item
```

Using this method allows you to create more abstract tests. For example, you can specify distribution as:

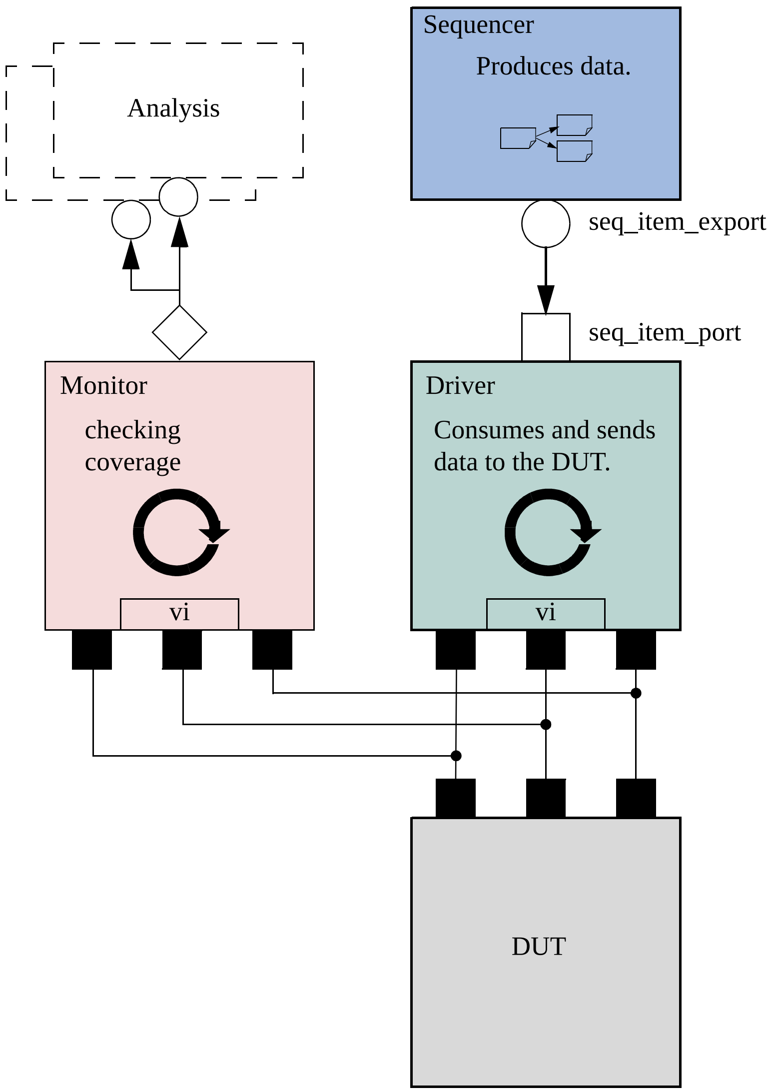
```
constraint delay_kind_d {delay_kind dist {ZERO:=2, SHORT:=1,
   MEDIUM:=1, LONG:=1, MAX:=2};}
```

When creating data items, keep in mind what range of values are often used or which categories are of interest to that data item. Then add knobs to the data items to simplify control and coverage of these data item categories.

Transaction-Level Components

As discussed in Transaction-Level Modeling (TLM) on page 23, TLM interfaces in OVM provide a consistent set of communication methods for sending and receiving transactions between components. The components themselves are instantiated and connected in the testbench, to perform the different operations required to verify a design. A simplified testbench is shown in Figure 4-1 on page 39.

Figure 4-1 Simplified Transaction-Level Testbench



The basic components of a simple transaction-level verification environment are:

1. A stimulus generator (sequencer) to create transaction-level traffic to the DUT

2. A driver to convert these transactions to signal-level stimulus at the DUT interface
3. A monitor to recognize signal-level activity on the DUT interface and convert it into transactions
4. An analysis component, such as a coverage collector or scoreboard, to analyze transactions

As we shall see, the consistency and modularity of the TLM interfaces in OVM allow components to be reused as other components are replaced and/or encapsulated. Every component is characterized by its interfaces, regardless of its internal implementation. This chapter will discuss how to encapsulate these types of components into a proven architecture, an OVC, to improve reuse even further.

Figure 4-2 Highly-Reusable OVC Agent

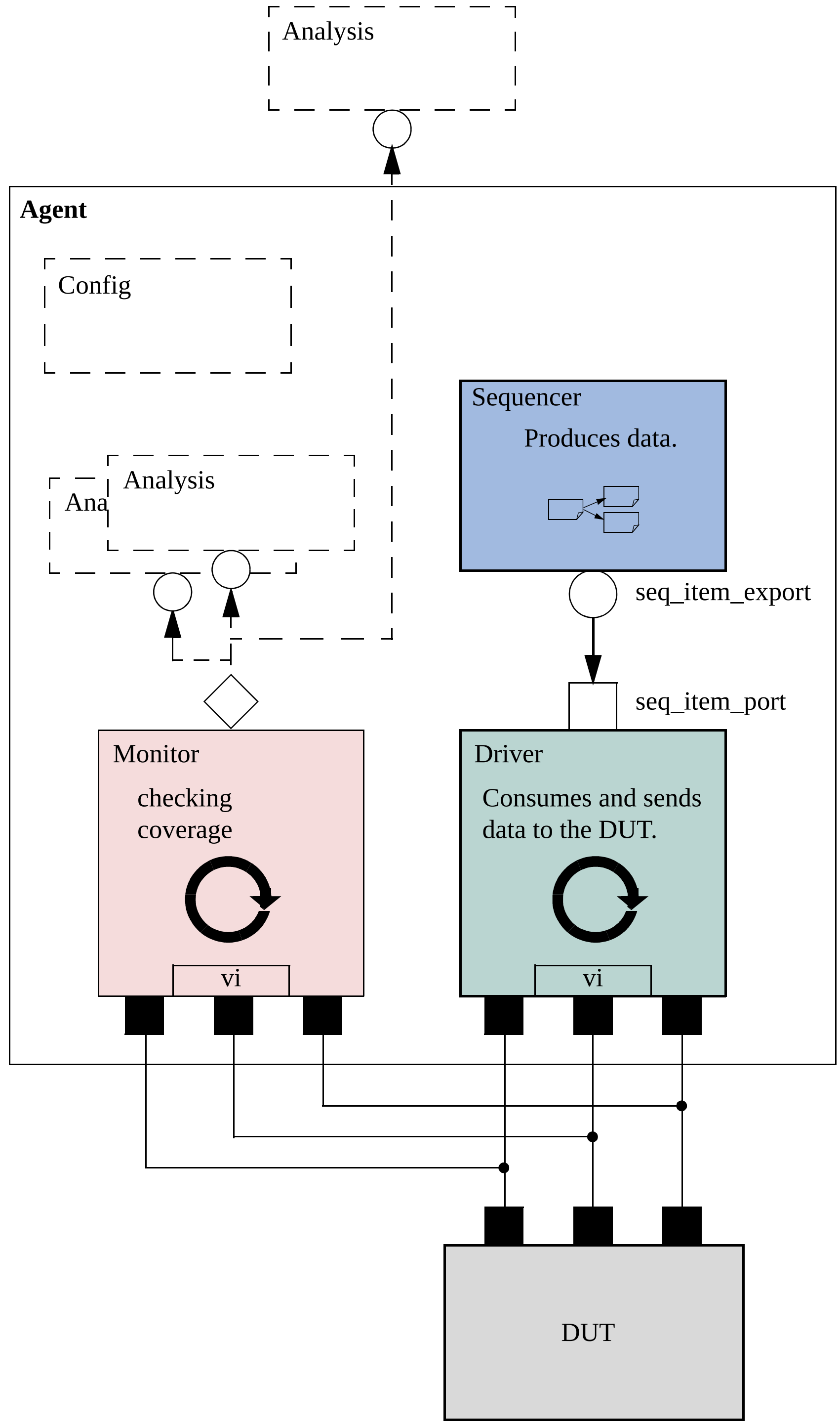


Figure 4-2 on page 41 shows the recommended grouping of individual components into a reusable interface-level OVC agent. Instead of reusing the low-level classes individually, the developer creates a component that encapsulates it’s sub-classes in a consistent way. Promoting a consistent architecture makes these components easier to learn, adopt, and configure.

Creating the Driver

The driver's role is to drive data items to the bus following the interface protocol. The driver obtains data items from the sequencer for execution. The SystemVerilog OVM Class Library provides the `ovm_driver` base class, from which all driver classes should be extended, either directly or indirectly. The driver has a `run()` method that defines its operation, as well as a TLM port through which it communicates with the sequencer (see example below).

To create a driver:

1. Derive a driver from the `ovm_driver` base class.
2. If desired, add OVM infrastructure macros for class properties to implement utilities for printing, copying, comparing, and so on.
3. Obtain the next data item from the sequencer and execute it as outlined above.
4. Declare a virtual interface in the driver to connect the driver to the DUT.

Refer to “Generating Stimulus with Sequences and Sequence Items” on page 58 for a description of how a sequencer, driver, and sequences synchronize with each other to generate constrained random data.

The class `simple_driver` in the example below defines a driver class. The example derives `simple_driver` from `ovm_driver` (parameterized to use the `simple_item` transaction type) and uses the methods in the `seq_item_port` object to communicate with the sequencer. As always, include a constructor and the `` `ovm_component_utils `` macro to register the driver type with the common factory.

```
class simple_driver extends ovm_driver #(simple_item);
  simple_item s_item;
  virtual dut_if vif;
  // OVM automation macros for general components
  `ovm_component_utils(simple_driver)
  // Constructor
  function new (string name = "simple_driver", ovm_component parent);
    super.new(name, parent);
   endfunction : new
  task run();
    forever begin
      // Get the next data item from sequencer (may block).
      seq_item_port.get_next_item(s_item);
      // Execute the item.
```

```
15       drive_item(s_item);
16       seq_item_port.item_done(); // Consume the request.
17     end
18   endtask : run
19
20   task drive_item (input simple_item item);
21     ... // Add your logic here.
22   endtask : drive_item
23 endclass : simple_driver
```

Line 1 Derive the driver.

Line 5 Add OVM infrastructure macro.

Line 13 Call `get_next_item()` to get the next data item for execution from the sequencer.

Line 16 Signal the sequencer that the execution of the current data item is done.

Line 21 Add your application-specific logic here to execute the data item.

More flexibility exists on connecting the drivers and the sequencer see more on connecting driver

Creating the Sequencer

The sequencer generates stimulus data and passes it to a driver for execution. The SystemVerilog OVM Class Library provides the `ovm_sequencer` base class, which is parameterized by the `request` and `response` item types. You should derive all sequencer classes directly or indirectly from this class.

To create a sequencer:

1. Derive a sequencer from the `ovm_sequencer` base class and specify the `request` and `response` type parameters.
2. Use `` `ovm_sequencer_utils `` and `` `ovm_update_sequence_lib_and_item `` to indicate the generated data item type and field desired automation.

This is all that is required to define baseline behavior for a sequencer. Refer to "Generating Stimulus with Sequences and Sequence Items" on page 58 for a description of how a sequencer, driver, and sequences synchronize with each other to generate constrained-random data.

The class `simple_sequencer` in the example below defines a sequencer class. The example derives it from `ovm_sequencer` and parameterizes it to use the `simple_item` type.

```
class simple_sequencer extends ovm_sequencer #(simple_item);
  // OVM automation macro for sequencers
  `ovm_sequencer_utils(simple_sequencer)
  // Constructor
```

```
  function new (string name="simple_sequencer", ovm_component parent);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(simple_item)
  endfunction : new
endclass : simple_sequencer
```

Note:

- In the class definition, by default, the response type is the same as the request type. If a different response type is desired, the optional second parameter must be specified for the ovm_sequencer base type:

  ```
  class simple_sequencer extends ovm_sequencer #(simple_item, simple_rsp);
  ```

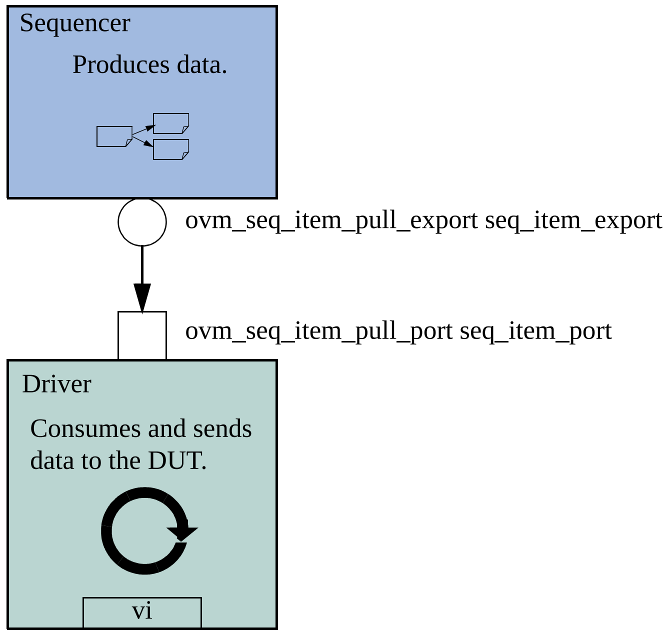
- The `` `ovm_component_utils `` macro should not be used here because its functionality is embedded in `` `ovm_sequencer_utils ``. Instead of using the `` `ovm_component_utils `` use `` `ovm_sequencer_utils ``, as well as the regular general automation this macro provides sequencer-specific infrastructure. Refer to OVM Macros in the *SystemVerilog OVM Class Reference* for more information.

- Call `` `ovm_update_sequence_lib_and_item `` macro from the constructor of your sequencer class. This macro registers all the sequence types that are associated with the current sequencer and indicates the sequencer's generated transaction type as a parameter. Refer to OVM Macros in the *SystemVerilog OVM Class Reference* for more information.

Connecting the Driver and Sequencer

The driver and the sequencer are connected via TLM, with the driver's `seq_item_port` connected to the sequencer's `seq_item_export` (see Figure 4-3 on page 45 below). The sequencer produces data items to provide via the export. The driver consumes data items through its `seq_item_port`, and optionally provides responses. The component that contains the instances of the driver and sequencer makes the connection between them. See "Creating the Agent" on page 51 below.

Figure 4-3 Sequencer-Driver Interaction



The `seq_item_port` in `ovm_driver` defines the set of methods used by the driver to obtain the next item in the sequence. An important part of this interaction is the driver's ability to synchronize to the bus, and to interact with the sequencer to generate data items at the appropriate time. The sequencer implements the set of methods that allows flexible and modular interaction between the driver and the sequencer.

Basic Sequencer and Driver Interaction

Basic interaction between the driver and the sequencer is done using the tasks `get_next_item()` and `item_done()`. As demonstrated in the example in Creating the Driver on page 42, the driver uses `get_next_item()` to fetch the next randomized item to be sent. After sending it to the DUT, the driver signals the sequencer that the item was processed using `item_done()`.Typically, the main loop within a driver resembles the following pseudo code.

```
get_next_item(req);
// Send item following the protocol.
item_done();
```

Note: `get_next_item()` is blocking.

Querying for the Randomized Item

In addition to the `get_next_item()` task, the ovm_seq_item_pull_port class provides another task, `try_next_item()`. This task will return in the same simulation step if no data items are available for execution. You can use this task to have the driver execute some idle transactions, such as when the DUT has to be stimulated when there are no meaningful data to transmit. The following example shows a revised implementation of the `run()` task in the previous example (in Creating the Driver on page 42), this time using `try_next_item()` to drive idle transactions as long as there is no real data item to execute:

```
task run();
  forever begin
    // Try the next data item from sequencer (does not block).
    seq_item_port.try_next_item(s_item);
    if (s_item == null) begin
      // No data item to execute, send an idle transaction.
      ...
    end
    else begin
      // Got a valid item from the sequencer, execute it.
      ...
      // Signal the sequencer; we are done.
      seq_item_port.item_done();
    end
  end
endtask: run
```

Fetching Consecutive Randomized Items

In some protocols, such as pipelined protocols, the driver gets a few generated items to fill the pipeline before the first items were completely processed. In such cases, the driver calls `item_done()` without providing the response to the sequencer. In such scenarios the driver logic may look like the following pseudo code:

```
while the pipeline is not empty{
  get_next_item(req);
  fork;
    logic that sends item to the pipeline
  join_none;
  item_done();
  for each completed process call{
    ...
  }
}
```

Sending Processed Data Back to the Sequencer

In some sequences, a generated value depends on the response to previously generated data. By default the data items between the driver and the sequencer are copied by reference, which means that changes the driver makes to the data item will visible inside the sequencer. In cases where the data item between the driver and the sequencer is copied by value, the driver needs to return the processed response back to the sequencer. Do this using the optional argument to `item_done()`.

```
item_done(rsp);
```

or using the built-in analysis port in `ovm_driver`:

```
rsp_port.write(rsp);
```

Note: Before providing the response, the response's sequence and transaction id must be set to correspond to the request transaction using `rsp.set_id_info(req)`.

With the basic functionality of driver-sequencer communication outlined above, the steps required to create a driver are straightforward.

Using TLM-Based Drivers

The `seq_item_port`, which is built into `ovm_driver`, is a bidirectional port. It also includes standard TLM methods `get()` and `peek()` for requesting an item from the sequencer, and `put()` to provide a response. Thus other components, which may not necessarily be derived from `ovm_driver`, may still connect to and communicate with the sequencer. As with the `seq_item_port`, the methods to use depend on the interaction desired.

```
// Pause sequencer operation while the driver operates on the transaction.
peek(req);

// Process req operation.
get(req);
```

Note:

- `peek()` is a blocking method, so the driver may block waiting for an item to be returned.
- The `get()` operation notifies the sequencer to proceed to the next transaction. It returns the same transaction as the `peek()`, so the transaction may be ignored.

```
// Allow sequencer to proceed immediately upon driver receiving transaction.
get(req);

// Process req operation.
```

To provide a response using the `blocking_slave_port`, the driver would call

```
seq_item_port.put(rsp);
```

The response may also be sent back using an `analysis_port` as well.

Creating the Monitor

The monitor is responsible for extracting signal information from the bus and translating it into events, structs, and status information. This information is available to other components and to the test writer via standard TLM interfaces and channels. The monitor should never rely on state information collected by other components, such as a driver, but it may need to rely on request-specific id information in order to properly set the sequence and transaction id information for the response.

The monitor functionality should be limited to basic monitoring that is always required. This can include protocol checking—which should be configurable so it can be enabled or disabled—and coverage collection. Additional high-level functionality such as scoreboards should be implemented separately on top of the monitor.

If you want to verify an abstract model or accelerate the pin-level functionality, you should separate the signal-level extraction, coverage, checking, and the transaction-level activities. An analysis port should allow communication between the sub-monitor components (see Built-In TLM Channels in the *SystemVerilog OVM Class Reference*).

Monitor Example

The following example shows a simple monitor which has the following functions:

- The monitor collects bus information through a virtual interface (`xmi`).
- The collected data is used in coverage collection and checking.
- The collected data is exported on an analysis port (`item_collected_port`).

Actual code for collection is not shown in this example. A complete example can be found in the XBus example in xbus_master_monitor.sv.

```
class master_monitor extends ovm_monitor;
  virtual bus_if xmi; // SystemVerilog virtual interface
  bit checks_enable = 1; // Control checking in monitor and interface.
  bit coverage_enable = 1; // Control coverage in monitor and interface.
  ovm_analysis_port #(simple_item) item_collected_port;
  event cov_transaction; // Events needed to trigger covergroups
  protected simple_item trans_collected;
  `ovm_component_utils_begin(master_monitor)
    `ovm_field_int(checks_enable, OVM_ALL_ON)
    `ovm_field_int(coverage_enable, OVM_ALL_ON)
  `ovm_component_utils_end
  function new (string name, ovm_component parent);
    super.new(name, parent);
    cov_trans = new();
    cov_trans.set_inst_name({get_full_name(), ".cov_trans"});
    trans_collected = new();
    item_collected_port = new("item_collected_port", this);
  endfunction : new
```

```
    virtual task run();
      fork
        collect_transactions(); // Spawn collector task.
      join
    endtask : run

    covergroup cov_trans @cov_transaction;
      option.per_instance = 1;
      ... // Coverage bins definition
    endgroup : cov_trans

    virtual protected task collect_transactions();
      forever begin
        @(posedge xmi.sig_clock);
        ...// Collect the data from the bus into trans_collected.
        if (checks_enable)
          perform_transfer_checks();
        if (coverage_enable)
          perform_transfer_coverage();
        item_collected_port.write(trans_collected);
      end
    endtask : collect_transactions

    virtual protected function void perform_transfer_coverage();
      -> cov_transaction;
    endfunction : perform_transfer_coverage

    virtual protected function void perform_transfer_checks();
      ... // Perform data checks on trans_collected.
    endfunction : perform_transfer_checks

  endclass : master_monitor
```

The collection is done in a task (`collect_transaction`) which is spawned at the beginning of the `run()` phase. It runs in an endless loop and collects the data as soon as the signals indicate that the data is available on the bus.

As soon as the data is available it is sent to the analysis port (`item_collected_port`) for other components waiting for the information.

Coverage collection and checking are conditional because they can affect simulation run-time performance. If not needed, they can be turned off by setting `coverage_enable` or `checks_enable` to 0, using the configuration mechanism. For example:

```
set_config_int(“master0.monitor”, “checks_enable”, 0);
```

If checking is enabled, the task calls the `perform_transfer_checks` function, which performs the necessary checks on the collected data (`trans_collected`). If coverage collection is enabled, the task emits the coverage sampling event (`cov_transaction`) which results in collecting the current values.

Note: SystemVerilog does not allow concurrent assertions in classes, so protocol checking can also be done using assertions in a SystemVerilog interface.

Instantiating Components

The isolation provided by object-oriented practices and TLM interfaces between components facilitate reuse in OVM enabling a great deal of flexibility in building environments. Because each component is independent of the others, a given component can be replaced by a new component with the same interfaces without having to change the parent's `connect()` method. This flexibility is accomplished through the use of the *factory* in OVM.

When instantiating components in OVM, rather than calling its constructor (in bold below),

```
class my_component extends ovm_component;
  my_driver driver;
  ...
  function build();
    driver = new("driver",this);
    ...
  endfunction
endclass
```

components are instantiated using the `create()` method.

```
class my_component extends ovm_component;
  my_driver driver;
    ...
  function build();
    driver = my_driver::type_id::create("driver",this);
      ...
  endfunction
endclass
```

The factory operation is explained in "The Built-In Factory and Overrides" on page 108. The `type_id::create()` method is a type-specific static method that returns an instance of the desired type (in this case, `my_driver`) from the factory. The arguments to `create()` are the same as the standard constructor arguments, a string name and a parent component.The use of the factory allows the developer to derive a new class extended from `my_driver` and cause the factory to return the extended type in place of `my_driver`. Thus, the parent component can use the new type without modifying the parent class.

For example, for a specific test, an environment user may want to change the driver.

To change the driver for a specific test:

1. Declare a new driver extended from the base component and add or modify functionality as desired.

   ```
   class new_driver extends my_driver;
     ... // Add more functionality here.
   endclass: new_driver
   ```

2. In your test, environment, or testbench, override the type to be returned by the factory.

```
virtual function build();
  set_type_override_by_type(my_driver::get_type(),
     new_driver::get_type());
endfunction
```

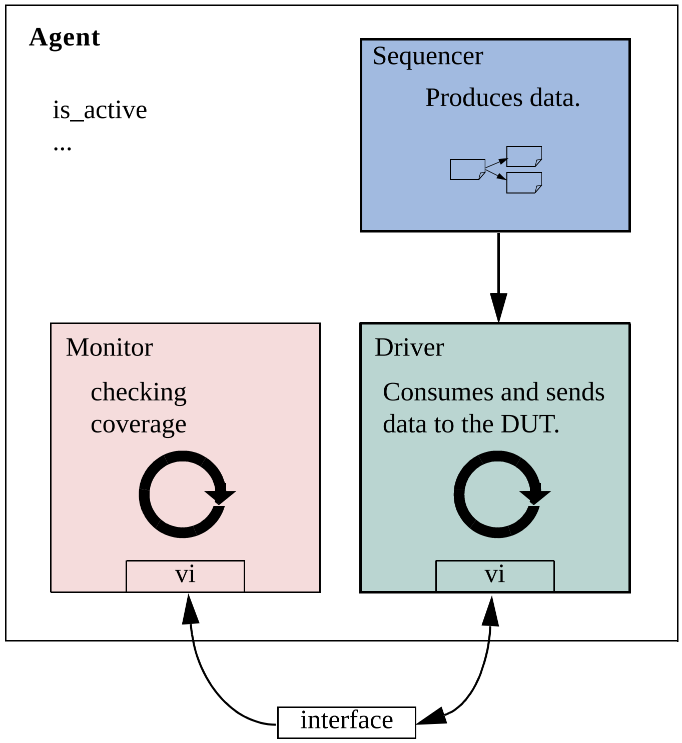
The factory also allows a new type to be returned for the creation of a specific instance as well. In either case, because `new_driver` is an extension of `my_driver`, and the TLM interfaces are the same, the connections defined in the parent remain unchanged.

Creating the Agent

The agent (Figure 4-4 on page 52) instantiates and connects together a driver, monitor, and sequencer using TLM connections as described in the preceding sections. To provide greater flexibility, the agent also contains configuration information and other parameters. As discussed in "Agent" on page 18, OVM recommends that the OVC developer create an agent that provides protocol-specific stimuli creation, checking, and coverage for a device. In a bus-based environment, an agent models either a master or a slave component. An agent has two basic operating modes:

- Active mode—the agent emulates a device in the system and drives DUT signals. This mode requires that the agent instantiate a driver and sequencer. A monitor also is instantiated for checking and coverage.
- Passive mode—the agent does not instantiate a driver or sequencer and operates passively. Only the monitor is instantiated and configured. Use this mode when only checking and coverage collection is desired.

Figure 4-4 Agent



The class `simple_agent` in the example below instantiates a sequencer, a driver, and a monitor in the recommended way. Instead of using the constructor, the OVM `build()` phase is used to configure and construct the subcomponents of the agent. Unlike constructors, this virtual function can be overridden without any limitations. Also, instead of hard coding the allocation `create_component()` is used to instantiate the subcomponents. The example in "To change the driver for a specific test:" on page 50 illustrates how you can override existing behavior using `extends`.

```
class simple_agent extends ovm_agent;
  ovm_active_passive_enum is_active;
  ... // Constructor and OVM automation macros
  simple_sequencer sequencer;
  simple_driver driver;
  simple_monitor monitor;
  // Use build() phase to create agents's subcomponents.
  virtual function void build();
    super.build()
    monitor = simple_monitor::type_id::create("monitor",this);
    if (is_active == OVM_ACTIVE) begin
      // Build the sequencer and driver.
      sequencer = simple_sequencer::type_id::create("sequencer",this);
      driver = simple_driver::type_id::create("driver",this);
    end
```

```
16   endfunction : build
17   virtual function void connect();
18     if(is_active == OVM_ACTIVE) begin
19       driver.seq_item_port.connect(sequencer.seq_item_export);
20     end
21   endfunction : connect
22 endclass : simple_agent
```

Note: You should always call `super.build()` (see Line 9) to update the given component's configuration overrides. This is crucial to providing the capability for an enclosing component to be able to override settings of an instance of this component.

Line 10 The monitor is created using `create()`.

Lines 11-15 The `if` condition tests the `is_active` property to determine whether the driver and sequencer are created in this agent. If the agent is set to active (`is_active = OVM_ACTIVE`), the driver and sequencer are created using additional `create()` calls.

Both the sequencer and the driver follow the same creation pattern as the monitor.

This example shows the `is_active` flag as a configuration property for the agent. You can define any control flags that determine the component's topology. At the environment level, this could be a `num_masters` integer, a `num_slaves` integer, or a `has_bus_monitor` flag. See "XBus OVC Example" on page 133 for a complete interface OVC example that uses all the control fields previously mentioned.

Note: Calling `create()` from the `build()` method is the recommended way to create any multi-hierarchical component.

Lines 18-20 The `if` condition should be checked to see if the agent is active and, if so, the connection between the sequencer and driver is made using `connect()`.

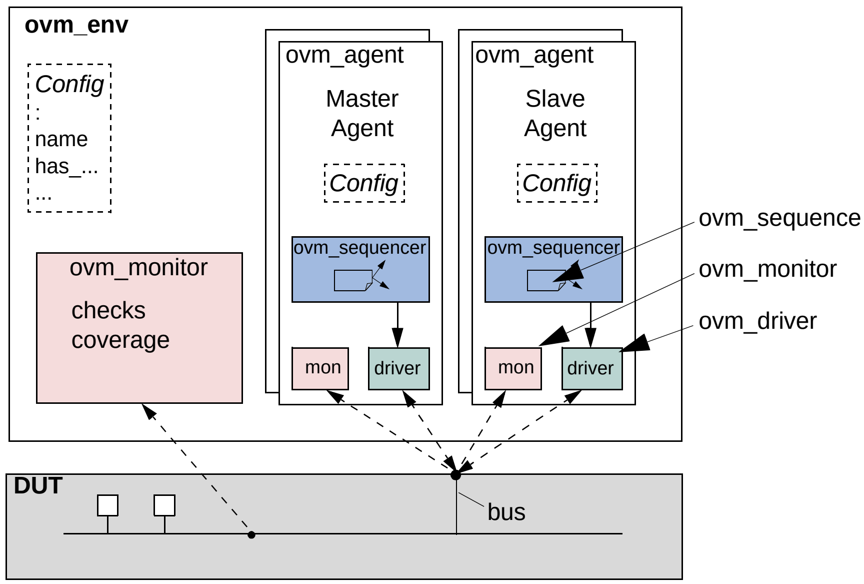
Using connect() to Connect Components

The `connect()` phase, which happens after the build is complete, should be used to connect the components inside the agent. See Lines 18-20 in the example above.

Creating the Environment

Having covered the basic operation of transaction-level verification components in a typical environment above, this section describes how to assemble these components into a reusable environment (Figure 4-5 on page 54). By following the guidelines here, you can ensure that your environment will be architecturally correct, consistent with other OVCs, and reusable. The following sections describe how to create and connect environment sub-components.

Figure 4-5 Typical OVM Environment Architecture



The Environment Class

The environment class is the top container of reusable components. It instantiates and configures all of its subcomponents. Most verification reuse occurs at the environment level where the user instantiates an environment class and configures it and its agents for specific verification tasks. For example, a user might need to change the number of masters and slaves in a new environment as shown below.

```
class ahb_env extends ovm_env;
  int num_masters;
  ahb_master_agent masters[];
  `ovm_component_utils_begin(ahb_env)
    `ovm_field_int(num_masters, OVM_ALL_ON)
  `ovm_component_utils_end
  virtual function void build();
    string inst_name;
    super.build();
    masters = new[num_masters];
    for(int i = 0; i < num_masters; i++) begin
      $sformat(inst_name, "masters[%0d]", i);
      masters[i] = xbus_master_agent::type_id::create(inst_name,this);
    end
    // Build slaves and other components.
  endfunction
  function void assign_vi(virtual interface ahb_bus  ahb_all);
    // Based on the configuration, assign master, slave, decoder and
    // arbiter signals.
  endfunction
```

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

Note: Similarly to the agent, `create` is used to allocate the environment sub-components. This allows introducing derivations of the sub-components later.

The user is not required to call `build()` explicitly. The SystemVerilog OVM Class Library will do this for all created components. Once all the components' `build()` functions are complete, the library will call each component's `connect()` function. Any connections between child components should be made in the `connect()` function of the parent component.

The OVM Configuration Mechanism

An OVC is created on a per-protocol basis for general-purpose protocol-related use. It may support various features or operation modes that are not required in a particular project. OVM provides a standard configuration mechanism which allows you to define the OVC's configuration to suit the current project's requirements. The OVC can get the configuration during run time or during the build process. Doing this during the build allows you to modify the environment object structure without touching multiple classes.

Properties that are registered as OVM fields using the `ovm_field_*` macros will be automatically updated by the component's `super.build()` method. These properties can then be used to determine the `build()` execution for the component.

It is not required to call a created component's `build()` function. The SystemVerilog OVM Class Library will do this for the user for all components that have not had their `build()` function called explicitly by the user. However it is possible, if the user requires, to call the component's `build()` function explicitly.

Connections among the created components is made in the `connect()` function of the component. Since `connect()` happens after `build()`, the user can assume the environment topology is fully created. With the complete topology, the user can then make the necessary connections.

Making the OVC Reusable

There are times when you as the developer know the context in which the OVC you are developing will be used. In such cases you should take care to separate the requirements of the OVC's protocol from those of the project. It is strongly recommended that you use only the interface-protocol documentation in developing the OVC. Later, you can consult your project's documentation to see if there are some generic features which might be useful to implement. For example, you should be able to configure slave devices to reside at various locations within an address space.

As another example, if within a protocol frame a few bits are defined as reserved, they should stay reserved within the OVC. The verification logic that understands how a specific implementation uses these bits should be defined outside the global generic code.

As a developer, it is critical to identify these generic parameters and document them for the environment users.

How to Create a Configurable Attribute

Making an attribute configurable is part of the built-in automation that the SystemVerilog OVM Class Library provides. Using the automation macros for `copy()`, `print()`, `compare()`, and so on, also introduces these attributes to the configuration mechanism. In the example in "The Environment Class" on page 54, `num_master` is a configuration parameter that allows changing the master agent numbers as needed. Since the `` `ovm_field_int `` declaration is already provided for printing, there is no further action needed to allow the users to configure it.

For example, to get three master agents, you can would specify:

```
set_config_int("my_env", "num_masters", 3);
```

This can be done in procedural code within the testbench. For more information, see "OVC Configuration" on page 76.

Note:

- The values of parameters are automatically updated in the `super.build()` phase. Make sure that you call `super.build()` before accessing these values.
- If you prefer not to use the automation macros, you can use `get_config_int()` to fetch the configuration value of a parameter. You can also do this if you are concerned that the `num_masters` field was overridden and you want to re-fetch the original configuration value for it.
- A larger environment can integrate smaller ones and reconfigure their parameters to suit the needs of the parent environment. In this case, if there are contradicting configuration directives, the first `set_config` directives from the parent environment will take precedence.

Enabling Scenario Creation

The environment user will need to create many test scenarios to verify a given DUT. Since the OVC developer is usually more familiar with the DUT's protocol, the developer should facilitate the test writing (done by the OVC's user) by doing the following:

- Place knobs in the data item class to simplify declarative test control.

- Create a library of interesting reusable sequences.

Note: The environment user controls the environment-generated patterns configuring its sequencers. The user can:

- Add a sequence of transactions to a sequencer.
- Modify the sequencer to use specific sequences more often than others.
- Override the sequencer's main loop to start with a user-defined sequence instead.

In this section we describe how to create a library of reusable sequences and review their use. For more information on how to control environments, see "Creating Meaningful Tests" on page 80.

Declaring User-Defined Sequences

Sequences are made up of several data items, which together form an interesting scenario or pattern of data. Verification components can include a library of basic sequences (instead of single-data items), which test writers can invoke. This approach enhances reuse of common stimulus patterns and reduces the length of tests. In addition, a sequence can call upon other sequences, thereby creating more complex scenarios.

Note: The SystemVerilog OVM Class Library provides the `ovm_sequence` base class. You should derive all sequence classes directly or indirectly from this class.

To create a user-defined sequence:

1. Derive a sequence from the `ovm_sequence` base class and specify the request and response item type parameters. In the example below, only the request type is specified, `simple_item`. This will result in the response type also being of type `simple_item`.
2. Use the `` `ovm_sequence_utils `` macro to associate the sequence with the relevant sequencer type and to declare the various automation utilities. This macro also provides a `p_sequencer` variable that is of the type specified by the second argument of the macro. This allows access to derived type-specific sequencer properties.
3. Implement the sequence's `body` task with the specific scenario you want the sequence to execute. In the `body` task, you can execute data items and other sequences using "`ovm_do" on page 60 and "`ovm_do_with" on page 61.

Example

The class `simple_seq_do` in the following example defines a simple sequence. It is derived from `ovm_sequence` and uses the `` `ovm_sequence_utils `` macro to associate this sequence with `simple_sequencer`, and to declare the various utilities `` `ovm_object_utils `` would provide.

```
class simple_seq_do extends ovm_sequence #(simple_item);
  rand int count;
  constraint c1 { count >0; count <50; }
  // Constructor
  function new(string name="simple_seq_do");
    super.new(name);
  endfunction
  // OVM automation macros for sequences
  `ovm_sequence_utils(simple_seq_do, simple_sequencer)
  // The body() task is the actual logic of the sequence.
  virtual task body();
    repeat(count)
      `ovm_do(req)
  endtask : body
endclass : simple_seq_do
```

Once you define a sequence, it is registered inside its sequencer and may be generated by the sequencer's default generation loop. The `` `ovm_sequence_utils `` macro creates the necessary infrastructure to associate this sequence with the relevant sequencer type, and declares the various automation utilities. This macro is similar to the `` `ovm_object_utils `` macro (and its variations) except that it takes a second argument, which is the sequencer type name this sequence is associated with.

Note: Do not use the `` `ovm_object_utils `` macro when using the `` `ovm_sequence_utils `` macro. The functionality of `` `ovm_object_utils `` is included in `` `ovm_sequence_utils ``.

Generating Stimulus with Sequences and Sequence Items

Sequences allow you to define:

- Streams of data items sent to a DUT.
- Streams of actions performed on a DUT interface.

You can also use sequences to generate static lists of data items with no connection to a DUT interface.

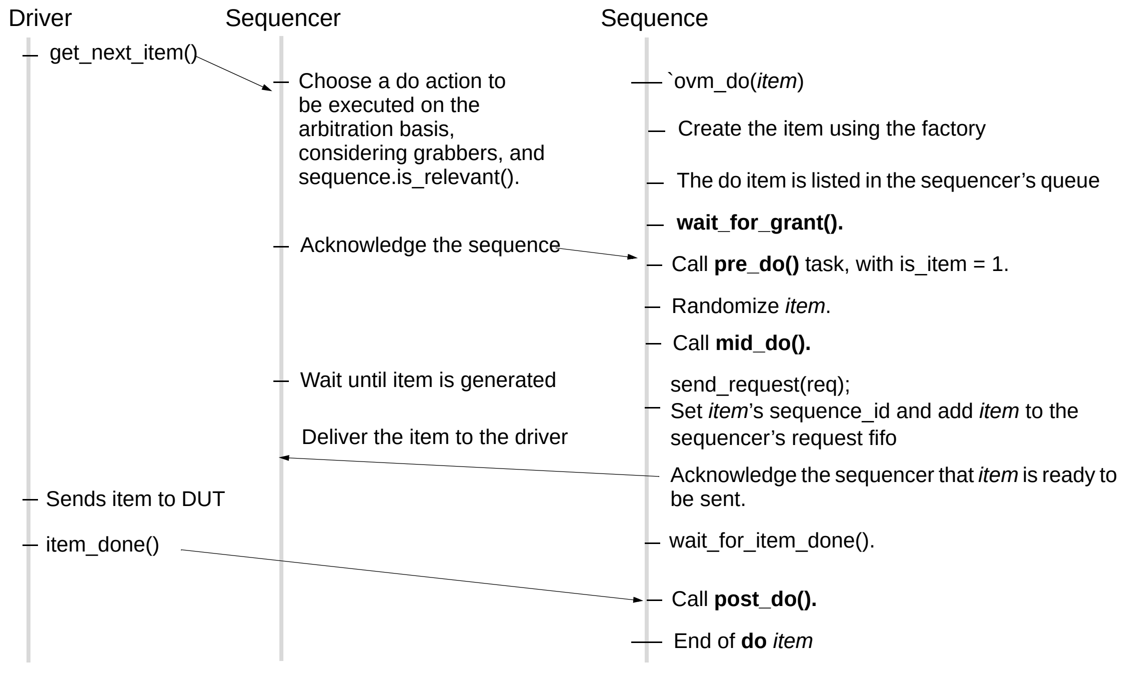
Getting Started with Sequences

Previous sections have discussed the basics of creating sequences and sequence items using the SystemVerilog OVM Class Library. This section discusses how to generate stimulus using the sequence and sequence item macros provided in the class library.

Figure 4-6 on page 59 and Figure 4-7 on page 60 show the complete flow for sequence items and sequences when used with the `ovm_do` macros. The entire flow includes the allocation of an object based on factory settings for the registered type, which is referred to as "creation" in this section. After creation, comes the initialization of class properties. Although the balance of the object processing depends on whether the object is a sequence item or a sequence, the `pre_do(), mid_do()` and `post_do()` callbacks of the parent sequence and randomization of the objects are also called, but at different points of processing for each object type as shown in the figures.

Note: You can use any of the macros with the SystemVerilog looping constructs.

Figure 4-6 Sequence Item Flow in Pull Mode



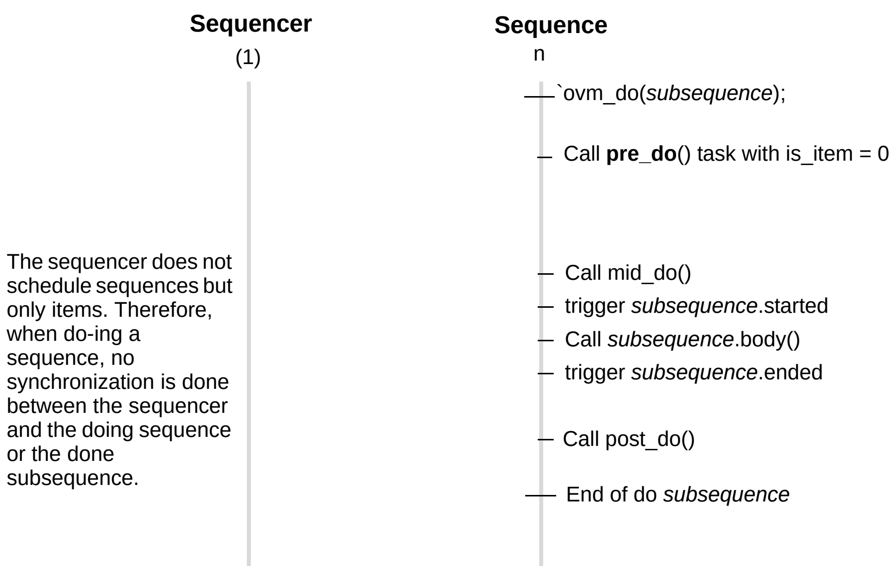
Note This flow occurs when the sequencer is set to 'pull_mode == 1'.

The `'ovm_do` macro and all related macros provide a convenient set of calls to create, randomize, and send transaction items in a sequence. The `ovm_do` macro delays randomization of the item until the driver has signaled that it is ready to receive it and the `pre_do` method has been executed. Other macro variations allow constraints to be applied to the randomization (`ovm_do_with`), or bypass the randomization altogether. The individual methods wrapped by `'ovm_do` in Figure 4-6 on page 59 may be called individually with no loss of functionality:

1. Create the item using the factory.
2. Call `wait_for_grant()`.
3. Call `pre_do()`, or some other functionality.
4. Optionally randomize *`item`*.
5. Call `mid_do()` or some other functionality, if desired.

6. Call `send_request()`.
7. Call `wait_for_item_done()`.
8. Optionally call `post_do()` or other functionality.
9. Optionally call `get_response()`.

Figure 4-7 Subsequence Flow



Note This flow does not depend on the driver interaction mode.

Sequence and Sequence Item Macros

This section describes the sequence and sequence item macros, `` `ovm_do `` and `` `ovm_do_with ``.

`ovm_do

This macro takes as an argument either a variable of type `ovm_sequence` or of type `ovm_sequence_item`. An object is created using the factory settings and assigned to the specified variable. Based on the processing in Figure 4-6 on page 59, when the driver requests an item from the sequencer, the item is randomized and provided to the driver.

The simple_seq_do sequence declaration in the example in "Declaring User-Defined Sequences" on page 57 is repeated here. The body of the sequence invokes an item of type simple_item, using the `ovm_do macro.

```
class simple_seq_do extends ovm_sequence #(simple_item);
  ... // Constructor and OVM automation macros
  // See "Creating and Adding a New Sequence" on page 84
  virtual task body();
    `ovm_do(req)
  endtask : body
endclass : simple_seq_do
```

Similarly, a sequence variable can be provided and will be processed as shown in Figure 4-7 on page 60. The following example declares another sequence (simple_seq_sub_seqs), which uses `ovm_do to execute a sequence of type simple_seq_do, which was defined earlier.

```
class simple_seq_sub_seqs extends ovm_sequence #(simple_item);
  ... // Constructor and OVM automation macros
  // See "Creating and Adding a New Sequence" on page 84.
  simple_seq_do seq_do;
  virtual task body();
    `ovm_do(seq_do)
  endtask : body
endclass : simple_seq_sub_seqs
```

`ovm_do_with

This macro is similar to "`ovm_do" on page 60. The first argument is a variable of a type derived from ovm_sequence_item, which includes items and sequences. The second argument can be any valid inline constraints that would be legal if used in arg1.randomize() with inline constraints. This enables adding different inline constraints, while still using the same item or sequence variable.

Example

This sequence produces two data items with specific constraints on the values of addr and data.

```
class simple_seq_do_with extends ovm_sequence #(simple_item);
  ... // Constructor and OVM automation macros
  // See "Creating and Adding a New Sequence" on page 84.
  virtual task body();
    `ovm_do_with(req, { addr == 16'h0120; data == 16'h0444; } )
    `ovm_do_with(req, { addr == 16'h0124; data == 16'h0666; } )
  endtask : body
endclass : simple_seq_do_with
```

Predefined Sequences

There are three built-in sequences: ovm_random_sequence, ovm_exhaustive_sequence, and ovm_simple_sequence. User-defined sequences are loaded into the sequencer's sequence queue prior to the run simulation phase. Upon entering the run phase, the sequencer starts the sequence named by its `default_sequence` configurable property and the transactions begin to flow. The default value for `default_sequence` is `ovm_random_sequence`.

ovm_random_sequence

This sequence is a built-in sequence pre-loaded into the sequencer. This sequence randomly selects and executes sequences from the sequencer's library (excluding `ovm_random_sequence` and `ovm_exhaustive_sequence`). The number of sequences executed depends on the *`count`* field of the sequencer. If *`count`* is set to -1, the random sequence will randomize a number between 0 and `ovm_sequencer::max_random_count`. If *`count`* is not -1, then *`count`* sequences will be executed by `ovm_random_sequence`.

The following task is the default sequence which all sequencers execute, unless you configure their `default_sequence` attribute to a different value.

```
task ovm_random_sequence::body();
  if (m_sequencer.count == -1) begin
    assert(randomize(l_count) with { l_count > 0 && l_count <
       m_sequencer.max_random_count; });
    m_sequencer.count = l_count;
  end
  else
    l_count = m_sequencer.count;
  ...
  repeat (l_count) begin
    assert(randomize(l_kind) with { l_kind > l_exhaustive_seq_kind &&l_kind
       < max_kind; });
    do_sequence_kind(l_kind);
  end
endtask
```

ovm_exhaustive_sequence

This sequence is a built-in sequence which is pre-loaded into the sequencer. This sequence exhaustively executes all the user-defined sequences for the current sequencer. The pre-defined `ovm_simple_sequence` will also be executed, but the other two pre-defined sequence types (`ovm_random_sequence` and `ovm_exhaustive_sequence`) will not. The sequences are executed exactly once and in a random order. The `l_kind` variable is declared as randc in order to randomize without replacement.

```
task ovm_exhaustive_sequence::body();
  l_count = m_sequencer.sequences.size() - 2;
  max_kind = m_sequencer.sequences.size();
  l_exhaustive_seq_kind =
     m_sequencer.get_seq_kind("ovm_exhaustive_sequence");
  repeat (l_count) begin
    assert(randomize(l_kind) with {
      l_kind > l_exhaustive_seq_kind &&l_kind < max_kind; });
    // l_kind is randc.
    do_sequence_kind(l_kind);
```

```
  end
endtask
```

ovm_simple_sequence

This sequence is a built-in sequence which is pre-loaded into the sequencer. This sequence calls `` `ovm_do(*item*) ``. *`item`* is a property in `ovm_sequence`. This sequence is provided to allow default execution of the OVC without any user-defined sequences.

```
task ovm_simple_sequence::body();
  `ovm_do(item)
endtask
```

Configuring the Sequencer's Default Sequence

Sequencers execute an `ovm_random_sequence` object by default. The sequencer has a string property named “default_sequence” which can be set to a user-defined sequence-type name. This sequence will be used as the default sequence for the instance of the sequencer.

To override the default sequence:

1. Declare a user-defined sequence class which derives from an appropriate base sequence class.

   The example in “Declaring User-Defined Sequences” on page 57 provides a declaration example of a sequence named `simple_seq_do`.

2. Configure the default_sequence property for a specific sequencer or a group of sequencers. Typically, this is done inside the test class before creating the component that includes the relevant sequencer(s). For example,

   ```
   set_config_string("*.master0.sequencer","default_sequence",
      "simple_seq_do");
   ```

   The first argument utilizes a wildcard mechanism. Here, any instance name containing “.master0.sequencer” will have its default_sequence property (if it exists) set to the value `simple_seq_do`.

Overriding Sequence Items and Sequences

In a user-defined `ovm_test`, for example `base_test_xbus_demo` (discussed in “Creating the Base Test” on page 78), you can configure the simulation environment to use a modified version of an existing sequence or a sequence item by using the common factory to create instances of sequence and sequence-item classes. See “The Built-In Factory and Overrides” on page 108 for more information.

To override any reference to a specific sequence or sequence-item type:

1. Declare a user-defined sequence or sequence item class which derives from an appropriate base class. The following example shows the declaration of a basic sequence item of type `simple_item`, and a derived item of type `word_aligned_item`.

2. Invoke the appropriate `ovm_factory` override method, depending on whether you are doing a global or instance-specific override. For example, assume the `simple_seq_do` sequence is executed by a sequencer of type `simple_sequencer` (both defined in "Declaring User-Defined Sequences" on page 57). You can choose to replace all processing of `simple_item` types with `word_aligned_item` types. This can be selected for all requests for `simple_item` types from the factory, or for specific instances of `simple_item`. From within an OVM component, the user can execute the following:

```
// Affect all factory requests for type simple_item.
set_type_override_by_type(simple_item::get_type(),
   word_aligned_item::get_type());
// Affect requests for type simple_item only on a given sequencer.
set_inst_override_by_type("env0.agent0.sequencer.*",
   simple_item::get_type(), world_aligned_item::get_type());
// Alternatively, affect requests for type simple_item for all
// sequencers of a specific env.
set_inst_override_by_type("env0.*.sequencer.*", simple_item::get_type(),
   word_aligned_item::get_type());
```

3. Use any of the sequence macros that allocate an object (as defined in "Sequence and Sequence Item Macros" on page 60), for example, the `` `ovm_do `` macro.

Since the sequence macros call the common factory to create the data item object, existing override requests will take effect and a `word_aligned_item` will be created instead of a `simple_item`.

Building a Reusable Sequence Library

A reusable sequence library is a set of user-defined sequences. Creating an OVC reusable sequence library is an efficient way to facilitate reuse. The environment developer can create a meaningful set of sequences to be leveraged by the test writer. Such sequence libraries avoid code duplication in tests, making them more maintainable, readable, and concise.

Tips

- Try to think of interesting protocol scenarios that many test writers can use.
- Since some users may not want to use the reusable sequence library (because the sequences may not match the design requirements of the user), do not `include your reusable sequence library within the OVC files. Leave it to the user to decide whether to use them.

Implementing Checks and Coverage

Checks and coverage are crucial to a coverage-driven verification flow. SystemVerilog allows the usage shown in Table 4-1 on page 65 for cover, covergroup, and assert constructs.

Note: This overview is for concurrent assertions. Immediate assertions can be used in any procedural statement. Refer to the SystemVerilog LRM for more information.

Table 4-1 SystemVerilog Checks and Coverage Construct Usage Overview

	class	interface	package	module	initial	always	generate	program
assert	no	yes	no	yes	yes	yes	yes	yes
cover	no	yes	yes	yes	yes	yes	yes	yes
covergroup	yes	yes	yes	yes	no	no	yes	yes

In an OVC, checks and coverage are defined in multiple locations depending on the category of functionality being analyzed. In Figure 5-2 on page 73, checks and coverage are depicted in the `ovm_monitor` and interface. The following sections describe how the cover, covergroup, and assert constructs are used in the OVM XBus OVC example (described in "XBus OVC Example" on page 133).

Implementing Checks and Coverage in Classes

Class checks and coverage should be implemented in the classes derived from `ovm_monitor`. The derived class of `ovm_monitor` is always present in the agent and thus will always contain the necessary checks and coverage. The bus monitor is created by default in an env, and if the checks and coverage collection is enabled the bus monitor will perform these functions. The remainder of this section uses the master monitor as an example of how to implement class checks and coverage, but they apply to the bus monitor as well.

You can write class checks as procedural code or SystemVerilog immediate assertions.

Tip: Use immediate assertions for simple checks that can be written in a few lines of code and use functions for complex checks that require many lines of code. The reason is that as the check becomes more complicated, so does the debug of that check.

Note: Concurrent assertions are not allowed in SystemVerilog classes per the IEEE1800 LRM.

Following is a simple example of an assertion check. This assertion verifies that the size field of the transfer is either 1, 2, 4, or 8. Otherwise the assertion fails.

```
function void xbus_master_monitor::check_transfer_size();
  check_transfer_size : assert(trans_collected.size == 1 ||
     trans_collected.size == 2 || trans_collected.size == 4 ||
     trans_collected.size == 8) else begin
       // Call DUT error: Invalid transfer size!
     end
endfunction : check_transfer_size
```

Following is a simple example of a function check. This function verifies that the size field value matches the size of the data dynamic array. While this example is not complex, it illustrates a procedural-code example of a check.

```
function void xbus_master_monitor::check_transfer_data_size();
  if (trans_collected.size != trans_collected.data.size())
    // Call DUT error: Transfer size field / data size mismatch.
endfunction : check_transfer_data_size
```

The proper time to execute these checks depends on the implementation. You should determine when to make the call to the check functions shown above. For the above example, both checks should be executed after the transfer is collected by the monitor. Since these checks happen at the same instance in time, a wrapper function can be created so that only one call has to be made. This wrapper function follows.

```
function void xbus_master_monitor::perform_transfer_checks();
  check_transfer_size();
  check_transfer_data_size();
endfunction : perform_transfer_checks
```

The `perform_transfer_checks()` function is called procedurally after the item has been collected by the monitor.

Functional coverage is implemented using SystemVerilog covergroups. The details of the covergroup (that is, what to make coverpoints, when to sample coverage, and what bins to create) should be planned and decided before implementation begins.

Following is a simple example of a covergroup.

```
// Transfer collected beat covergroup.
covergroup cov_trans_beat @cov_transaction_beat;
  option.per_instance = 1;
  beat_addr : coverpoint addr {
    option.auto_bin_max = 16; }
  beat_dir : coverpoint trans_collected.read_write;
  beat_data : coverpoint data {
    option.auto_bin_max = 8; }
  beat_wait : coverpoint wait_state {
    bins waits[] = { [0:9] };
    bins others = { [10:$] }; }
  beat_addrXdir : cross beat_addr, beat_dir;
  beat_addrXdata : cross beat_addr, beat_data;
endgroup : cov_trans_beat
```

This embedded covergroup is defined inside a class derived from `ovm_monitor` and uses the event `cov_transaction_beat` as its sampling trigger. For the above covergroup, you should assign the local variables that serve as coverpoints in a function, then emit the sampling trigger event. This is done so that each transaction data beat of the transfer can be covered. This function is shown in the following example.

```
// perform_transfer_coverage
virtual protected function void perform_transfer_coverage();
  -> cov_transaction;
  for (int unsigned i = 0; i < trans_collected.size; i++) begin
    addr = trans_collected.addr + i;
    data = trans_collected.data[i];
    wait_state = trans_collected.wait_state[i];
    -> cov_transaction_beat;
  end
endfunction : perform_transfer_coverage
```

This function covers several properties of the transfer and each element of the dynamic array data. SystemVerilog does not provide the ability to cover dynamic arrays. You should access each element individually and cover that value if necessary. The `perform_transfer_coverage()` function would, like the `perform_transfer_checks()` function, be called procedurally after the item has been collected by the monitor.

Implementing Checks and Coverage in Interfaces

Interface checks are implemented as assertions. Assertions are added to check the signal activity for a protocol. The assertions related to the physical interface are placed in the env's interface. For example, an assertion might check that an address is never X or Y during a valid transfer. Use assert as well as assume properties to express these interface checks.

An assert directive is used when the property expresses the behavior of the device under test. An assume directive is used when the property expresses the behavior of the environment that generates the stimulus to the DUT.

The mechanism to enable or disable the physical checks performed using assertions is discussed in "Controlling Checks and Coverage" on page 67.

Controlling Checks and Coverage

You should provide a means to control whether the checks are enforced and the coverage is collected. You can use an OVM bit field for this purpose. The field can be controlled using the `ovm_component` `set_config`* interface. Refer to ovm_threaded_component in the *SystemVerilog OVM Class Reference* for more information. Following is an example of using the `checks_enable` bit to control checks.

```
if (checks_enable)
  perform_transfer_checks();
```

If checks_enable is set to 0, the function that performs the checks is not called, thus disabling the checks. The following example shows how to turn off the checks for the master0.monitor.

```
set_config_int("masters[0].monitor", "checks_enable", 0);
```

The same facilities exist for the coverage_enable field in the XBus agent monitors and bus monitor.

Using OVCs

This chapter covers the steps needed to build a testbench from a set of reusable Open Verification Components (OVCs). OVM accelerates the development process and facilitates reuse. OVM users will have fewer hook-up and configuration steps and can exploit a library of reusable sequences to efficiently accomplish their verification goals.

In this chapter, a distinction is made between the environment integrator and the test writer who might have less knowledge about verification and wants to use OVM for creating tests. The test writer may skip the configuration sections and move directly into the test-creation sections.

The steps you need to perform to create a testbench from OVCs are:

1. Review the reusable OVC configuration parameters.
2. Instantiate and configure reusable OVCs.
3. Create reusable sequences for interface OVCs (optional).
4. Add a virtual sequencer (optional).
5. Add checking and functional coverage extensions.
6. Create tests to achieve coverage goals.

Before reading this chapter make sure you read the OVM Overview chapter of this manual. It is also recommended (but not required) that you read Developing Reusable Open Verification Components (OVCs) to get a deeper understanding of OVCs.

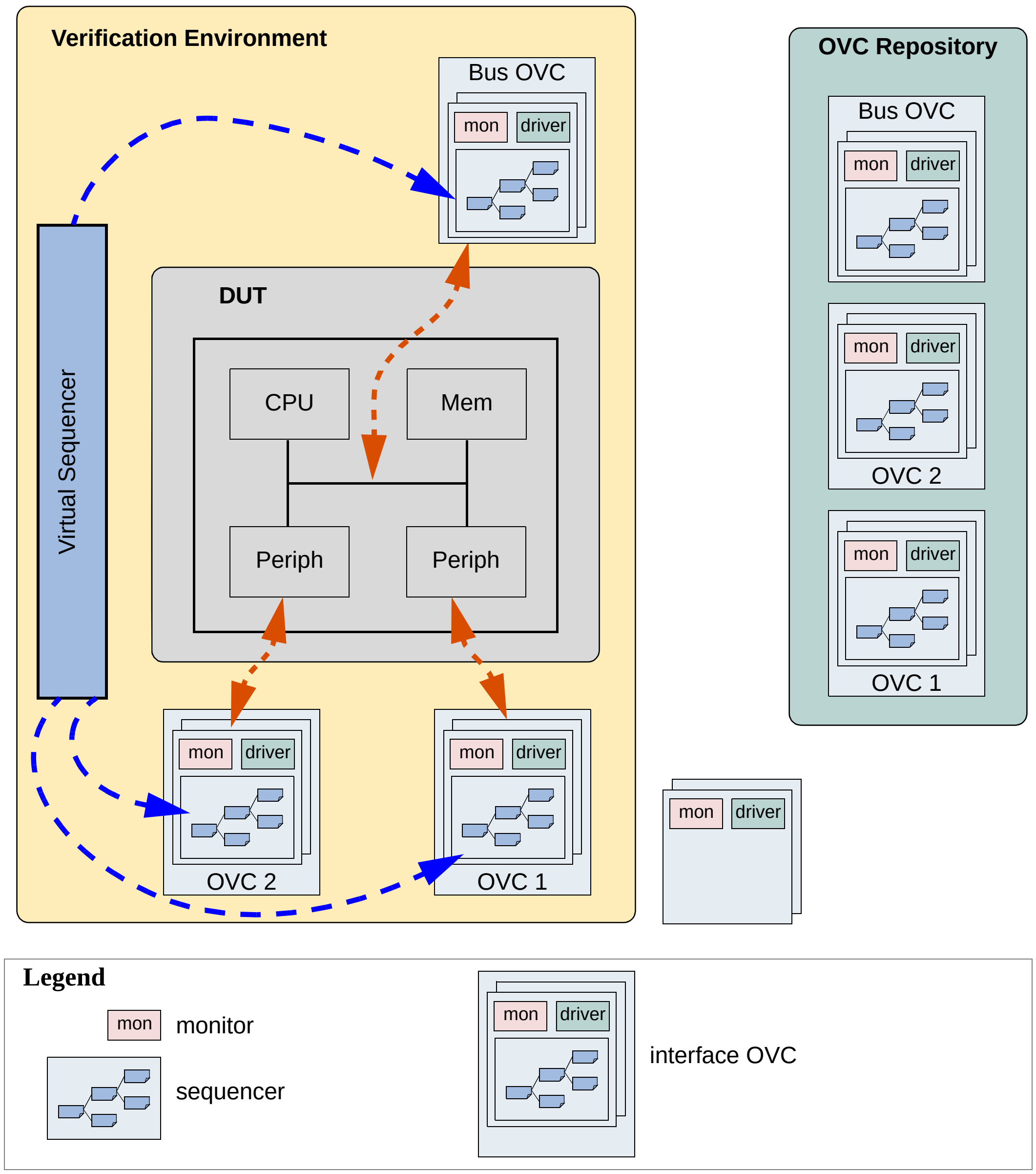
This chapter contains the following sections:

- "Implementing a Coverage Model" on page 100

Using an OVC

As illustrated in Figure 5-1 on page 71, the environment integrator instantiates and configures reusable components to build a desired testbench. The integrator also writes multiple tests to follow the verification plan in an organized way.

Figure 5-1 Verification Environment Example



Test Class

The `ovm_test` class defines the test scenario for the testbench specified in the test. The test class enables configuration of the testbench and environment classes as well as utilities for command-line test selection. Although IP developers provide default values for topological and run-time configuration properties, if you require configuration customization, use the configuration override mechanism provided by the SystemVerilog OVM Class Library. You can provide user-defined sequences in a file or package, which is included or imported by the test class. A test provides data and sequence generation and inline constraints. Test files are typically associated with a single configuration.

For usage examples of test classes refer to "Creating and Selecting a User-Defined Test" on page 78.

Tests in OVM are classes that are derived from an `ovm_test` class. Using classes allows inheritance and reuse of tests.

Testbench Class

The testbench is the container object that defines the testbench topology. The testbench instantiates the reusable verification IP and defines the configuration of that IP as required by the application.

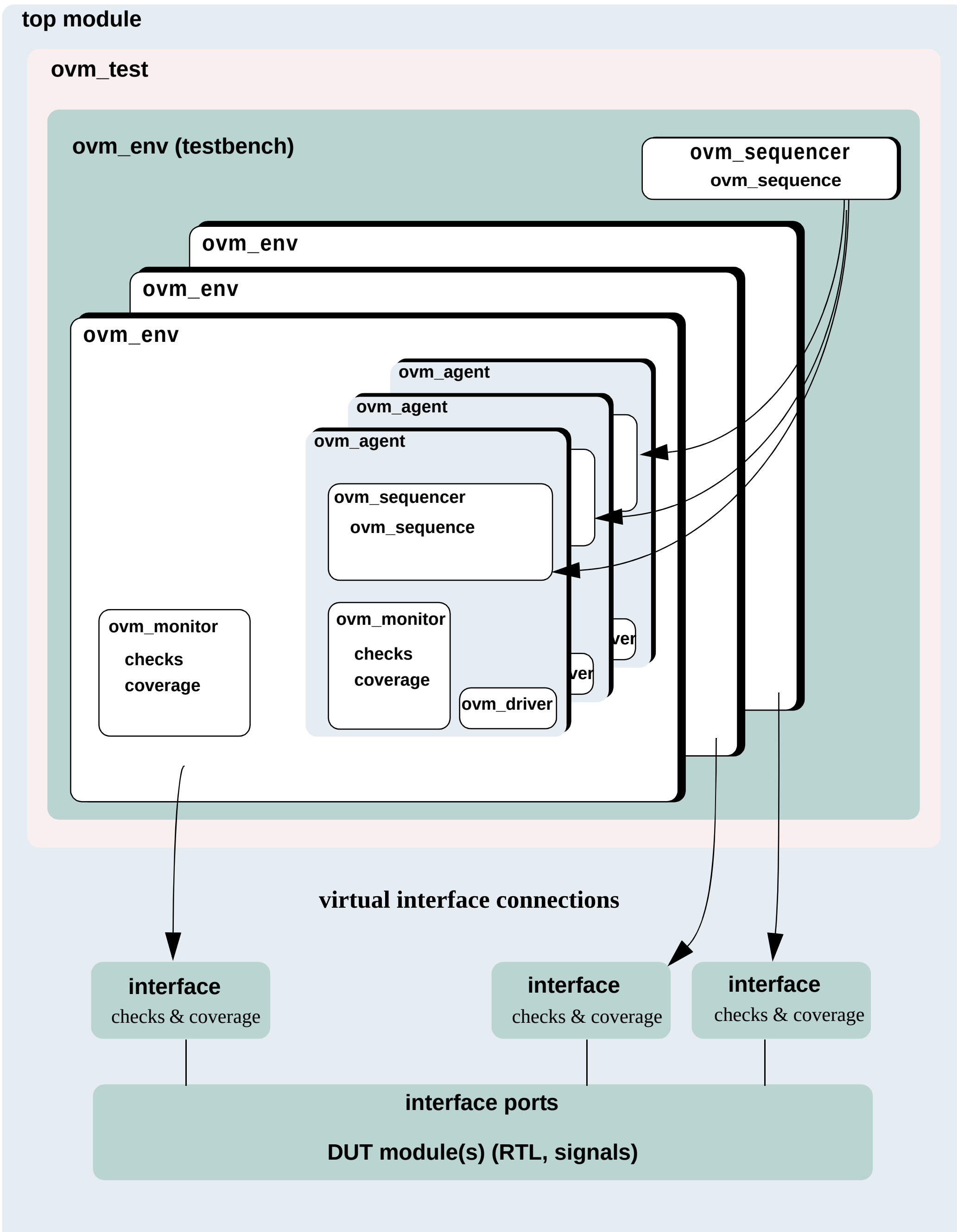
Instantiating the reusable environment directly inside the tests has several drawbacks:

- The test writer must know how to configure the environment.
- Changes to the topology require updating multiple test files, which can turn into a big task.
- The tests are not reusable because they rely on a specific environment structure.

For these reasons, OVM recommends using a testbench class. The testbench class is derived from the `ovm_env` class. The testbench instantiates and configures the reusable components for the desired verification task. Multiple tests can instantiate the testbench class and determine the nature of traffic to generate and send for the selected configuration.

Figure 5-2 on page 73 shows a typical verification environment that includes the test class containing the testbench class. Other environments (OVCs) are contained inside the testbench class.

Figure 5-2 OVC Verification Environment Class Diagram



Arrows represent virtual-interface connections.

Instantiating OVCs

This section describes how you can use OVCs to create a testbench that can be reused for multiple tests. The following example uses the verification IP in "XBus OVC Example" on page 133. This interface OVC can be used in many environments due to its configurability, but in this scenario it will be used in a simple configuration consisting of one master and one slave. The testbench sets the applicable topology overrides.

Note:

- Examples for the `set_config` calls can be found within the `build()` function.
- `set_config` must be called before the `build()` if it affects the testbench topology.

```
class xbus_demo_tb extends ovm_env;
// Provide implementations of virtual methods such as get_type_name().
  `ovm_component_utils(xbus_demo_tb)

  // XBus reusable environment
  xbus_env xbus0;

  // Scoreboard to check the memory operation of the slave
  xbus_demo_scoreboard scoreboard0;

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

  // build()
  virtual function void build();
    super.build(); // Configure before creating the subcomponents.
    set_config_int("xbus0", "num_masters", 1);
    set_config_int("xbus0", "num_slaves", 1);
    xbus0 = xbus_env::type_id::create("xbus0", this);
    scoreboard0 = xbus_demo_scoreboard::type_id::create("scoreboard0",
       this);;
  endfunction : build

  virtual function connect();
    // Connect slave0 monitor to scoreboard.
    xbus0.slaves[0].monitor.item_collected_port.connect(
    scoreboard0.item_collected_export);
    // Assign interface for xbus0.
    xbus0.assign_vi(xbus_tb_top.xi0);
  endfunction : connect

  virtual function void end_of_elaboration();
    // Set up slave address map for xbus0 (basic default).
    xbus0.set_slave_address_map("slaves[0]", 0, 16'hffff);
  endfunction : end_of_elaboration

endclass : xbus_demo_tb
```

Other configuration examples include:

- Set the `masters[0]` agent to be active:

```
set_config_int("xbus0.masters[0]", “is_active", OVM_ACTIVE);
```

- Do not collect coverage for `masters[0]` agent:

  ```
  set_config_int("xbus0.masters[0].monitor", "coverage_enable", 0);
  ```

- Set all slaves (using a wildcard) to be passive:

  ```
  set_config_int("xbus0.slaves*", "is_active", OVM_PASSIVE);
  ```

Many test classes may instantiate the testbench class above, therefore test writers do not need to understand all the details of how it is created and configured.

The `xbus_demo_tb`'s `new()` constructor is not used for creating the testbench subcomponents because there are limitations on overriding `new()` in object-oriented languages such as SystemVerilog. Instead, use a virtual `build()` function, which is a built-in OVM phase.

The `set_config_int` calls specify that the number of masters and slaves should both be 1. These configuration settings are used by the xbus0 environment during the `xbus0` `build()`. This defines the topology of the xbus0 environment, which is a child of the `xbus_demo_tb`.

In a specific test a user might want to extend the `xbus_env` and derive a new class from it. `create`() is used to instantiate the subcomponents (instead of the `new()` constructor) so that the `xbus_env` or the scoreboard classes can be replaced with derivative classes without changing the testbench file. See "Component Overrides" on page 109 for more information.

As required, `super.build()` is called as the first line of the `xbus_demo_tb`'s `build()` function. This updates the configuration fields of the `xbus_demo_tb`.

`connect()` is used to make the connection between the slave monitor and the scoreboard. The slave monitor contains a TLM analysis port which is connected to the TLM analysis export on the scoreboard. The virtual interface variable for the XBus environment is also assigned so that the environment topology can communicate with the top-level Verilog module. `connect()` is a built-in OVM phase.

After the `build()` and `connect()` functions are complete, the user can make adjustments to run-time properties since the environment is completely elaborated (that is, created and connected). The `end_of_elaboration()` function makes the environment aware of the address range to which the slave agent should respond.

The `xbus_demo_tb` defines the topology needed for the xbus demo tests. This object can be used as is or can be overridden from the test level, if necessary.

OVC Configuration

OVC Configurable Parameters

Based on the protocols used in a device, the integrator instantiates the needed environment classes and configures them for a desired operation mode. Some standard configuration parameters are recommended to address common verification needs. Other parameters are protocol- and implementation-specific.

Examples of standard configuration parameters:

- An agent can be configured for active or passive mode. In active mode, the agent drives traffic to the DUT. In passive mode, the agent passively checks and collects coverage for a device. A rule of thumb to follow is to use an active agent per device that needs to be emulated, and a passive agent for every RTL device that needs to be verified.
- The monitor collects coverage and checks a DUT interface by default. The user may disable these activities by the standard `checks_enable` and `coverage_enable` parameters.

Examples of user-defined parameters:

- The number of master agents and slave agents in an AHB OVC.
- The operation modes or speeds of a bus.

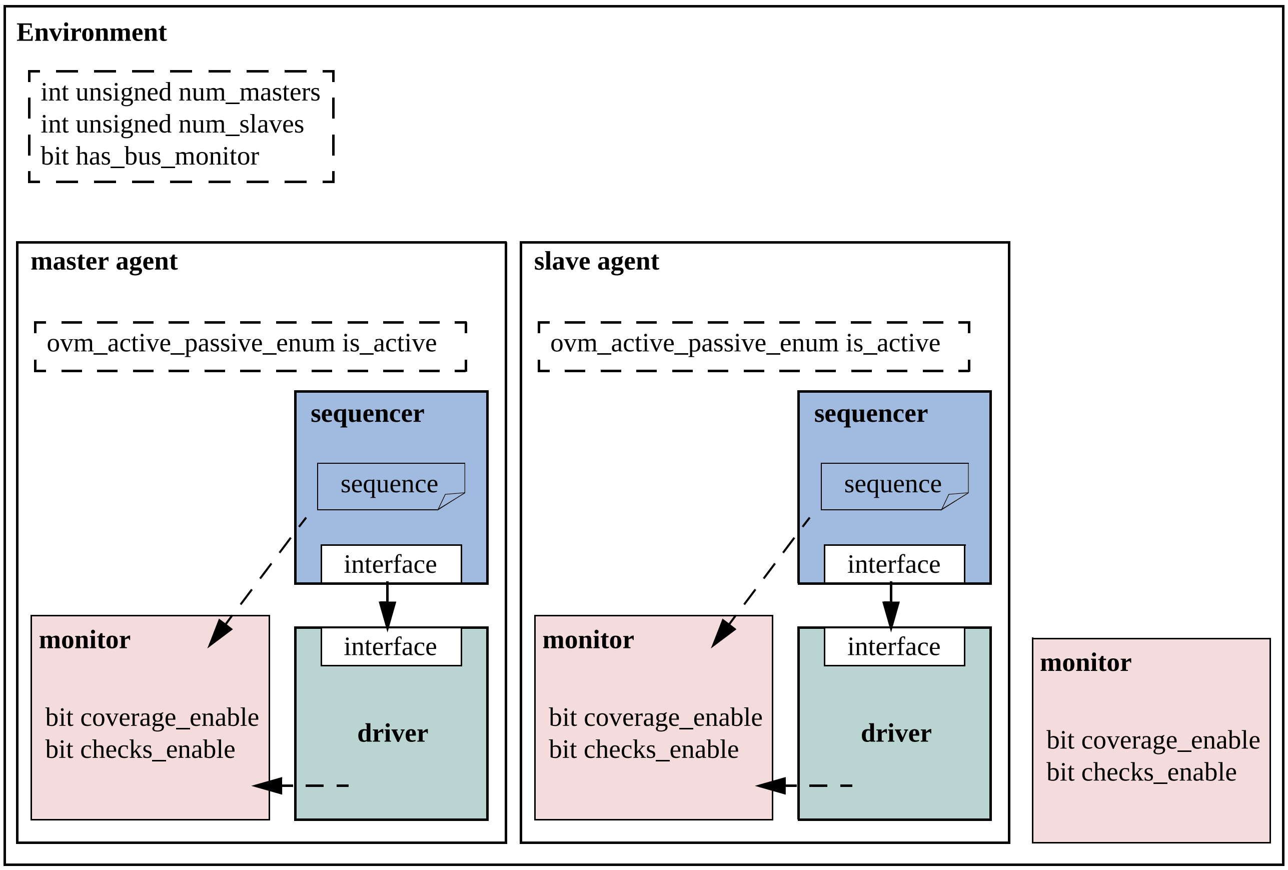
An OVM OVC should support the standard configuration parameters and provide user-defined configuration parameters as needed. Refer to the OVC's documentation for information about its user-defined parameters.

OVC Configuration Mechanism

OVM provides a configuration mechanism (see Figure 5-3 on page 77 below) to allow integrators to configure an environment without needing to know the OVC's implementation and hook-up scheme. Following are some examples.

```
set_config_int("xbus0", "num_masters", 1);
set_config_int("xbus0", "num_slaves", 1);
set_config_int("xbus0.masters[0]", “is_active", 1);
set_config_int("xbus0.slaves*", “is_active", 0);
set_config_int("xbus0.masters[0].monitor", “coverage_enable", 0);
```

Figure 5-3 Standard Configuration Fields and Locations



Using a Configuration Class

Some OVCs randomize configuration attributes inside a configuration class. Dependencies between these attributes are captured using constraints within the configuration object. In such cases, users can extend the configuration class to add new constraints, or layer additional constraints on the class using inline constraints. Once configuration is randomized, the test writer can use `set_config_object()` to assign the configuration object to one or more environments within the testbench. Similarly to `set_config_int()`, `set_config_object()` allows you to set the configuration to multiple environments in the testbench regardless of their location, and impact the build process of the testbench.

Creating and Selecting a User-Defined Test

In OVM, a test is a class that encapsulates test-specific instructions written by the test writer. This section describes how to create and select a test. It also describes how to create a test family base class to verify a topology configuration. This section contains the following:

- "Creating the Base Test" on page 78.
- "Creating Tests from a Test-Family Base Class" on page 79.
- "Test Selection" on page 79.

Creating the Base Test

The following example shows a base test that uses the `xbus_demo_tb` defined in "Instantiating OVCs" on page 74. This base test is a starting point for all derivative tests that will use the `xbus_demo_tb`. The complete test class is shown here:

```
class xbus_demo_base_test extends ovm_test;
  `ovm_component_utils(xbus_demo_base_test)
  xbus_demo_tb xbus_demo_tb0;
  // The test’s constructor
  function new (string name = "xbus_demo_base_test",
    ovm_component parent = null);
    super.new(name, parent);
  endfunction

  // Update this component's properties and create the xbus_demo_tb component.
  virtual function build(); // Create the testbench.
    super.build();
    xbus_demo_tb0 = xbus_demo_tb::type_id::create("xbus_demo_tb0", this);
  endfunction

  // Define a default run-time behavior.
  virtual task run();
    #2000
    // User-activated end of simulation
    global_stop_request(); // Terminate the simulation.
  endtask

endclass
```

The `build()` function of the base test creates the `xbus_demo_tb`. The SystemVerilog OVM Class Library will execute the `build()` function of the `xbus_demo_base_test` for the user when cycling through the simulation phases of the components. This creates the testbench environment because each sub-component will create components that will create more components in their `build()` functions.

The `run()` task of the base test prints the topology and then waits 2,000 time units, at which time the test halts the simulation using the `global_stop_request()` interface.

All of the definitions in the base test will be inherited by any test that derives from `xbus_demo_base_test`. This means that any derivative test will not have to build the testbench if the test calls `super.build()`. Likewise, the `run()` task behavior can be inherited. If the current implementation does not meet your needs, you can redefine both the `build()` and `run()` methods because they are both virtual.

Creating Tests from a Test-Family Base Class

You can derive from the base test defined in "Creating the Base Test" on page 78 in order to create tests that reuse the same topology. Since the testbench is created by the base test's `build()` function and the `run()` task defines the run phase, the derivative tests can make minor adjustments. (for example, changing the default sequence executed by the agents in the environment). Below is an example of a simple test that inherits from `xbus_demo_base_test`.

```
class test_read_modify_write extends xbus_demo_base_test;
  `ovm_component_utils(test_read_modify_write)
  // The test's constructor
  function new (string name = "test_read_modify_write",
     ovm_component parent = null);
    super.new(name, parent);
  endfunction
  // Register configurations to control which
  // sequence is executed by the sequencers.
  virtual function void build();
    // Substitute the default sequence.
    set_config_string("xbus_demo_tb0.xbus0.masters[0].sequencer",
       "default_sequence", "read_modify_write_seq");
    set_config_string("xbus_demo_tb0.xbus0.slaves[0].sequencer",
       "default_sequence", "slave_memory_seq");
    super.build();
  endfunction
endclass
```

This test changes the default sequence executed by the `masters[0]` agent and the `slaves[0]` agent. It is important that the settings for the default_sequence be set before calling `super.build()`, which creates the testbench. When `super.build()` is called, the `xbus_demo_tb0` and all its subcomponents are created.

This test relies on the `xbus_demo_base_test` implementation of the `run()` phase.

Test Selection

After you have declared a user-defined test (described in "Creating Tests from a Test-Family Base Class" on page 79), invoke the global OVM `run_test()` task in the top-level module to select a test to be simulated. Its prototype is:

```
task run_test(string test_name="");
```

When a test name is provided to the `run_test()` task, the factory is called to create an instance of the test with that type name. Simulation then starts and cycles through the simulation phases.

The following example shows how the test type name `test_read_modify_write` (defined in "Creating Tests from a Test-Family Base Class" on page 79) can be provided to the `run_test()` task.

A test name is provided to `run_test()` via a simulator command-line argument. If the top module calls `run_test()` without an argument, the `+OVM_TESTNAME=`*`test_name`* simulator command-line argument is checked. If present, `run_test()` will use *`test_name`*. Using the simulator command-line argument avoids having to hard code the test name in the `run_test()` task. For example, in the top-level module, call the `run_test()` as follows:

```
module tb_top;
  // DUT, interfaces, and all non-testbench code
  initial
    run_test();
endmodule
```

To select a test of type `test_read_modify_write` (described in "Creating Tests from a Test-Family Base Class" on page 79) using simulator command-line option, use the following command:

```
% simulator-command other-options +OVM_TESTNAME=test_read_modify_write
```

If the test name provided to `run_test()` does not exist, the simulation will exit immediately via a call to `$fatal`. If this occurs, it is likely the name was typed incorrectly or the `` `ovm_component_utils `` macro was not used.

By using this method and only changing the `+OVM_TESTNAME` argument, you can run multiple tests without having to recompile or re-elaborate the design or testbench.

Creating Meaningful Tests

The previous sections show how test classes are put together. At this point, random traffic is created and sent to the DUT. The user can change the randomization seed to achieve new test patterns. In order to achieve verification goals in a systematic way, the user will need to control test generation to cover specific areas.

The user can control the test creation using these methods:

- Add constraints to control individual data items. This method provides basic functionality. It is described in "Constraining Data Items" on page 81.
- Use OVM sequences to control the order of multiple data items. This method provides more flexibility and control. It is described in the "Using Sequences" on page 83.

Constraining Data Items

By default, sequencers repeatedly generate random data items. At this level, the test writer can control the number of generated data items and add constraints to data items to control their generated values.

To constrain data items:

1. Identify the data item classes and their generated fields in the OVC.
2. Create a derivation of the data item class that adds or overrides default constraints.
3. In a test, adjust the environment (or a subset of it) to use the newly-defined data items.
4. Run the simulation using a command-line option to specify the test name.

Data Item Example

```
typedef enum bit {BAD_PARITY, GOOD_PARITY} parity_e;

class uart_frame extends ovm_sequence_item;
  rand int unsigned transmit_delay;
  rand bit start_bit;
  rand bit [7:0] payload;
  rand bit [1:0] stop_bits;
  rand bit [3:0] error_bits;
  bit parity;
  // Control fields
  rand parity_e parity_type;

  function new(input string name);
    super.new(name);
  endfunction

  // Optional field declarations and automation flags
  `ovm_object_utils_begin(uart_frame)
     `ovm_field_int(start_bit, OVM_ALL_ON)
     `ovm_field_int(payload, OVM_ALL_ON)
     `ovm_field_int(parity, OVM_ALL_ON)
     `ovm_field_enum(parity_e, parity_type, OVM_ALL_ON + OVM_NOCOMPARE)
     `ovm_field_int(xmit_delay,  OVM_ALL_ON + OVM_DEC + OVM_NOCOMPARE)
  `ovm_object_utils_end

  // Specification section 1.2: the error bits value should be
  // different than zero.
  constraint error_bits_c {error_bits != 4'h0;}

  // Default distribution constraints
  constraint default_parity_type {parity_type dist {
    GOOD_PARITY:=90, BAD_PARITY:=10};}

  // Utility functions
  extern function bit calc_parity ( );
  ...
  endfunction

endclass: uart_frame
```

The `uart_frame` is created by the uart environment developer.

A few fields in the derived class come from the device specification. For example, a frame should have a payload that is sent to the DUT. Other fields are there to assist the test writer in controlling the generation. For example, the field `parity_type` is not being sent to the DUT, but it allows you to easily specify and control the parity distribution. Such control fields are called “knobs”. The OVC’s documentation should list the data item's knobs, their roles, and legal range.

Data items have specification constraints. These constraints can come from the DUT specification to create legal data items. For example, a legal frame must have `error_bits_c` not equal to 0. A different type of constraint in the data items constrains the traffic generation. For example, in the constraint block `default_parity_type` (in the example above), the parity bit is constrained to be 90-percent legal (good parity) and 10-percent illegal (bad parity).

Creating a Test-Specific Frame

In tests, the user may wish to change the way data items are generated. For example, the test writer may wish to have short delays. This can be achieved by deriving a new data item class and adding constraints or other class members as needed.

```
// A derived data item example
// Test code
class short_delay_frame extends uart_frame;
  // This constraint further limits the delay values.
  constraint test1_txmit_delay {transmit_delay  < 10;}

  `ovm_object_utils(short_delay_frame)

  function new(input string name="short_delay_frame");
    super.new(name);
  endfunction

endclass: short_delay_frame
```

Deriving the new class is not enough to get the desired effect. You also need to have the environment use the new class (`short_delay_frame`) rather than the OVC frame. The SystemVerilog OVM Class Library provides a mechanism that allows you to introduce the derived class to the environment.

```
class short_delay_test extends ovm_test;
  `ovm_component_utils(short_delay_test)
  uart_tb uart_tb0;
  function new (string name = "short_delay_test",ovm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function build();
    super.build();
    // Use short_delay_frame throughout the environment.
    factory.set_type_override_by_type(uart_frame::get_type(),
       short_delay_frame::get_type());
    uart_tb0 = uart_tb::type_id::create("uart_tb0", this);
  endfunction

  task run();
    ovm_top.print_topology();
    #2000;
    // User-activated end of simulation
```

```
        global_stop_request();
      endtask
    endclass
```

Calling the factory function `set_type_override_by_type()` (in bold above) instructs the environment to use short-delay frames.

At times, a user may want to send special traffic to one interface but keep sending the regular traffic to other interfaces. This can be achieved by using `set_inst_override_by_type()` inside an OVM component.

```
set_inst_override_by_type("uart_env0.master.sequencer.*",
   uart_frame::get_type(), short_delay_frame::get_type());
```

You can also use wildcards to override the instantiation of a few components.

```
set_isnt_override_by_type("uart_env*.msater.sequencer.*",
   uart_frame::get_type(), short_delay_frame::get_type());
```

Using Sequences

Constraint layering is an efficient way of uncovering bugs in your DUT. Having the constraint solver randomly select values ensures a non-biased sampling of the legal input space. However, constraint layering does not allow a user to control the order between consecutive data items. Many high-level scenarios can only be captured using a stream of ordered transactions. For example, simply randomizing bus transactions is unlikely to produce a legal scenario for your device. OVM sequences are library base classes that allow you to create meaningful ordered scenarios. This section describes OVM sequencers and sequences.

Important Randomization Concepts and Sequence Requirements

The previous section described the sequencer as a generator that can generate data items in a loop. While this is the default behavior, the sequencer actually generates sequences. User-defined sequences can be added to the sequencer's sequence library and randomly executed. If no user-defined sequences are added, then the only executed sequence is the built-in sequence called "simple_sequence" that execute a single data item.

"Controlling the Number of Sequences Created by ovm_random_sequence" on page 84 shows how you can use the configuration mechanism to modify the count to adjust the sequence generated pattern. This section introduces other advanced ways to control the sequencer, including:

- Creating and adding a new sequence to be executed.
- Changing the distribution of executed sequences.
- Adjust the sequencer to start from a sequence other than the pre-defined random sequence.

Controlling the Number of Sequences Created by ovm_random_sequence

The default number of generated sequences is a random number between 0 and `ovm_sequencer::max_random_count`. The user can modify the number of generated sequences (*count*). Use the configuration mechanism to change the value of *count*. For example, to generate and send 10 sequences, use:

```
set_config_int("*.cpu_seqr", "count", 10);
```

You can disable a sequencer from generating any sequences by setting the *count* to 0.

```
set_config_int("*.cpu_seqr", "count", 0);
```

Note: Having more data items than *count* is not necessarily a bug. The sequencer does not generate data items directly. By default, it generates *count* number of simple sequences that translate into *count* number of items. The sequencer has more built-in capabilities, which are described in the next section.

Creating and Adding a New Sequence

To create a user-defined sequence:

1. Derive a sequence from the `ovm_sequence` base class.
2. Use the `` `ovm_sequence_utils `` macro to associate the sequence with the relevant sequencer type and to declare the various automation utilities. This macro is similar to the `` `ovm_object_utils `` macro (and its variations) except that it takes another argument, which is the sequencer type name this sequence is associated with. This macro also provides a `p_sequencer` variable that is of the type specified by the second argument of the macro. This allows access to derived type-specific sequencer properties.
3. Implement the sequence's body task with the specific scenario you want the sequence to execute. In the body, you can execute data items and other sequences using "`ovm_do" on page 60 and "`ovm_do_with" on page 61.

Example

The class `retry_seq` in the example below defines a new sequence. It is derived from `ovm_sequence` and uses the `` `ovm_sequence_utils `` macro to associate this sequence with `uart_tx_sequencer` and to declare the various utilities `` `ovm_object_utils `` provides.

```
// Send one BAD_PARITY frame followed by a GOOD_PARITY
// frame with the same payload.
class retry_seq extends ovm_sequence #(uart_frame);
  rand bit [7:0] pload; // Randomizable sequence parameter
  ...
```

```
    // OVM automation for sequences
    ‘ovm_sequence_utils_begin(retry_seq, uart_tx_sequencer)
      ‘ovm_field_object(frame, OVM_ALL_ON)
      ‘ovm_field_int(pload, OVM_ALL_ON)
    ‘ovm_sequence_utils_end

    // Constructor
    function new(string name="retry_seq");
      super.new(name);
    endfunction

    task body ( ); // Sequence behavior
      ‘ovm_do_with(req, {payload == pload; parity == BAD_PARITY;} )
      ‘ovm_do_with(req, {payload == pload; parity == GOOD_PARITY;} )
    endtask : body

  endclass: retry_seq
```

Sequences can have parameters which can be randomized (for example, `pload` in this example). Use constraints to control the randomization of these parameters. Then use the randomized parameters within the `body()` task to guide the sequencer's behavior.

The `body` task defines the main behavior of a sequence. Since it is a task, you can use any procedural code, loops, fork and join, wait for events, and so on.

The `` `ovm_do_with `` macro randomizes and executes an item with inline constraints. The `` `ovm_do_with `` also sends the data item to the driver, which sends it to the DUT. The execution of the `body` task is blocked until the driver has sent the item to the DUT. Use the `` `ovm_do `` macro to randomize the item without inline constraints.

In the example above, when the retry sequence is executed, it will randomize the payload, send a frame with the generated payload having illegal parity, and follow it with a frame with a similar payload but with legal parity.

A sequencer type is provided as the second parameter to the `` `ovm_sequence_utils `` macro, which means that this sequence is added to the sequencer pool and could be randomly executed by the default random sequence. Since the sequencer type is provided, the `p_sequencer` variable can be declared the appropriate type and initialized.

Describing Nested Sequences

You can define more abstract sequences using existing sequences. Doing so provides additional reuse and makes it easier to maintain the test suite. For example, after defining the configuration sequence per device in a block-level testbench, the user may define a system-level configuration sequence which is a combination of the already-defined sequences.

Executing (doing) a sequence is similar to doing a data item. For example:

```
  // Call retry sequence wrapped with random frames.

  class rand_retry_seq extends ovm_sequence #(uart_frame);
    // Constructor, and so on
```

```
  ...
  `ovm_sequence_utils(rand_retry_rand_seq, uart_tx_sequencer)
  retry_seq retry_sequence; // Variable of a previously declared sequence
  task body (); // Sequence behavior
    `ovm_do (req)
    `ovm_do_with(retry_sequence, {pload inside {[0:31]};})
    `ovm_do(req)
  endtask
endclass
```

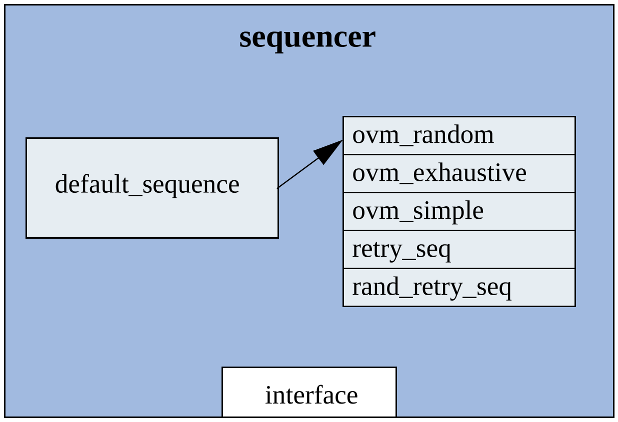
The `rand_retry_seq` has a field called `retry_sequence`. `retry_seq` is a user-predefined sequence.

The `body()` task is `do`-ing this sequence and layering inline constraints from above. This layering from above is one of many advantages that OVM sequences have.

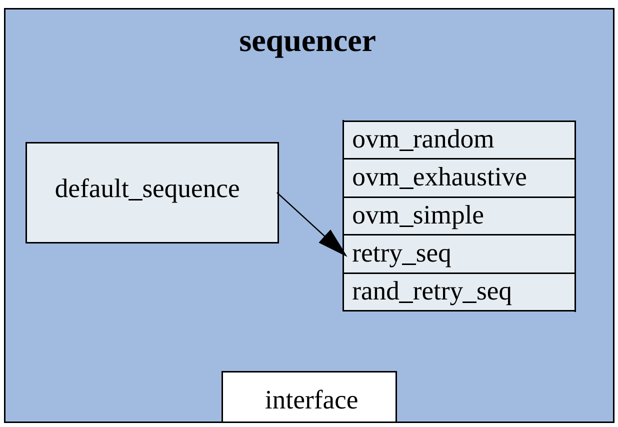
Adjusting the Sequencer

The sequencer has a string property named “default_sequence” which can be set to a user-defined sequence type. This sequence type is used as the default sequence for the current instance of the sequencer (Figure 5-4 on page 87).

Figure 5-4 Sequencer with a Sequence Library



In default mode, the sequencer executes the random sequence, which randomly selects sequences and executes them.



Setting default_sequence to “retry_seq” using
set_config_string("*.sequencer", "default_sequence", "retry_seq");
causes the sequencer to execute the “retry_seq” sequence.

To override the default sequence:

1. Declare a user-defined sequence class which derives from an appropriate base sequence class.
2. Configure the default_sequence property for a specific sequencer or a group of sequencers. This is typically done inside the test class, before creating the component that includes the relevant sequencer(s). For example,

```
set_config_string("*.master0.sequencer", "default_sequence","retry_seq");
```

The first argument uses a wildcard to match any instance name containing ".master0.sequencer" to set the default_sequence property (if it exists) to the value "retry_seq".

Sequence Libraries and Reuse

Use of sequences is an important part of OVC reuse. The environment developer who knows and understands the OVC protocol specifications can create interesting parameterized reusable sequences. This library of sequences enables the environment user to leverage interesting scenarios to achieve coverage goals more quickly. Check to see if your OVC's sequencer comes with a library of sequences. The example below shows a printout of a `sequencer.print()` command.

```
----------------------------------------------------------------
Name              Type                Size      Value
----------------------------------------------------------------
sequencer         uart_tx_sequencer-            @1011
default_sequence  string                19      ovm_random_sequence
sequences         da(string)             4      -
   [0]            string                19      ovm_random_sequence
   [1]            string                23      ovm_exhaustive_sequence
   [2]            string                19      ovm_simple_sequence
   [3]            string                 9      retry_seq
   [4]            string                14      rand_retry_seq
count             integral              32      -1
max_random_count  integral              32      'd10
max_random_depth  integral              32      'd4
```

The default sequence of this sequencer is `ovm_random_sequence`, which means that sequences will be randomly generated in a loop by default.

This sequencer has five sequences associated with it. Three sequencers are built-in sequences (`ovm_random_sequence`, `ovm_exhaustive_sequence`, and `ovm_simple_sequence`), and two are user-defined (`retry_seq` and `rand_retry_seq`).

The built-in exhaustive sequence is similar to random sequence. It randomly picks a sequence for execution, but it will not repeat a sequence before generating all other sequences. It also does not use the sequencer's `count` property but rather executes each sequence in the sequence library barring `ovm_random_sequence` and `ovm_exhaustive_sequence` once and only once.

The `count` value in this sequencer is -1, which means that the number of generated sequences will be between 0 and `max_random_count` (10, the default value, in this example).

For more information about sequences refer to "Advanced Sequence Control" on page 113.

Directed-Test Style Interface

The sequence style discussed in "Using Sequences" on page 83 is the recommended way to create tests. Focus is placed on creating reusable sequences that you can use across many tests, instead of placing stimulus scenarios directly inside the test. Each sequencer is preloaded with the default traffic that will be generated at run time and sent to the DUT. Inside the tests, the test writer needs to touch only the sequencers that need to be modified.

Some test writers, however, are accustomed to writing directed tests. In directed tests, you write procedural code in which you explicitly request each interface to generate and send items. While directed tests are not the recommended test-creation style, OVM support this method using the sequencer's `execute_item()` task. Before using directed tests, consider their disadvantages compared to the OVM-recommended test-creation method:

- Directed tests require more code to write and maintain. This becomes critical in system-level environments.
- In directed tests, the high-level intention of the code is not as clear or as easy to read and understand. In the recommended method, the code is focused on test-specific needs, and other system-related aspects are present by default. For example, the arbitration logic for slaves that service requests does not need to be coded in every test.
- Directed tests are less reusable because they contain specific and unreusable information.
- In the recommended method, tests are random by default. All declared sequences are candidates for execution by default. You must explicitly exclude a sequence from being executed. This prevents the problem of missing sequences and creates a more random pattern that can expose unanticipated bugs.
- In the recommended method for many protocols, you should never have to touch the high-level sequence, which serves as a template for other sub-sequences to be executed in a certain order.

The following code is an example of a directed test.

Note:

- The `execute_item()` task can execute a data item or a sequence. It blocks until the item or the sequence is executed by the sequencer. You can use regular SystemVerilog constructs such as fork/join to model concurrency.
- The default activity in the sequencers is disabled by setting the count parameters of all sequencers to 0. The `execute_item()` task is used to send traffic in a deterministic way.
- Using default random activity is a good practice. It is straightforward and a good investment. The use of `execute_item()` should be minimized and limited to specific scenarios.

```
class directed_test extends xbus_demo_base_test;
  `ovm_component_utils(directed_test)
```

```
    xbus_demo_tb xbus_demo_tb0;
    function new (string name = "directed_test",
      ovm_component parent = null);
      super.new(name, parent);
    endfunction
    virtual function void build();
      super.build();
      set_config_int("*.sequencer", "count", 0);
      // Create the testbench.
      xbus_demo_tb0 = xbus_demo_tb::type_id::create("xbus_demo_tb0", this);
    endfunction
    virtual task run();
      bit success; simple_item item;
      #10;
      item = new();
      success = item.randomize();
      tb.ahb.masters[1].sequencer.execute_item(item);
      success = item.randomize() with { addr < 32'h0123; } ;
      tb.ahb.masters[1].sequencer.execute_item(item);
    endtask
  endclass
```

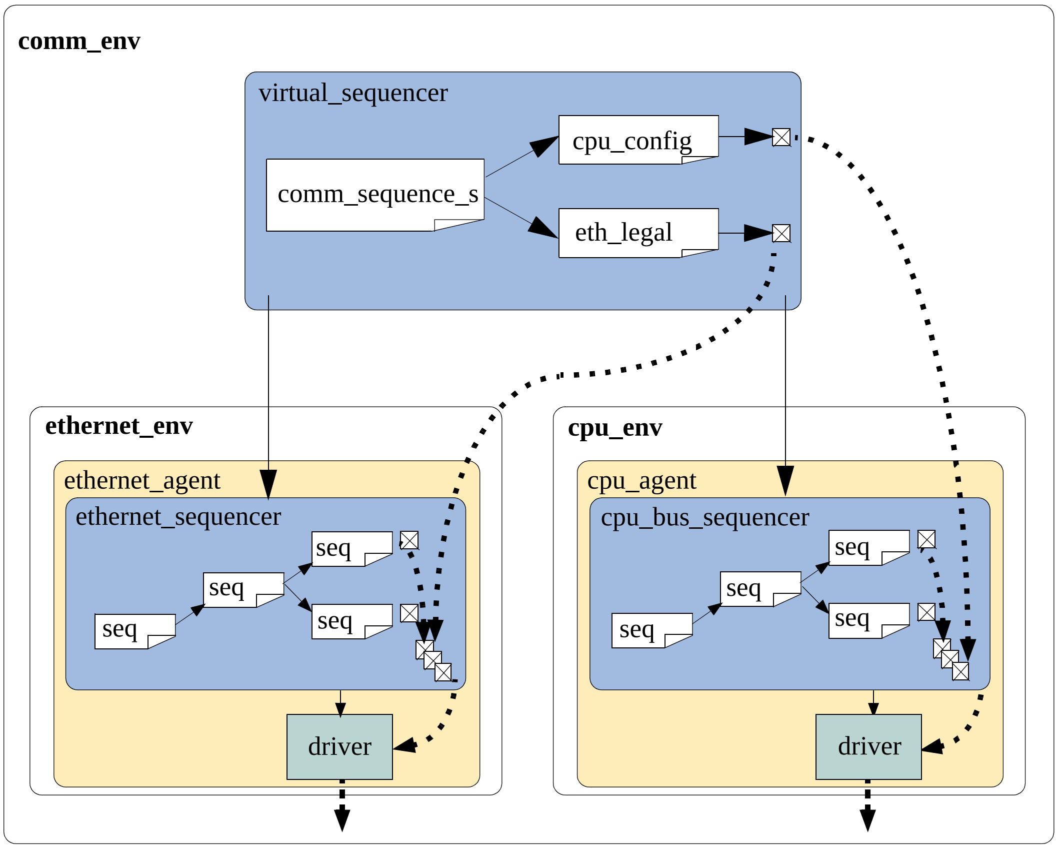
Virtual Sequences

"Creating Meaningful Tests" on page 80 describes how to efficiently control a single-interface generation pattern. However, in a system-level environment multiple components are generating stimuli in parallel. The user might want to coordinate timing and data between the multiple channels. Also, a user may want to define a reusable system-level scenario. Virtual sequences are associated with a virtual sequencer and are used to coordinate stimulus generation in a testbench hierarchy. In general, a virtual sequencer contains references to its subsequencers, that is, driver sequencers or other virtual sequencers in which it will invoke sequences. Virtual sequences can invoke other virtual sequences associated with its sequencer, as well as sequences in each of the subsequencers. However, virtual sequencers do not have their own data item and therefore do not execute data items on themselves. Virtual sequences can execute items on other sequencers that can execute items.

Virtual sequences enable centralized control over the activity of multiple verification components which are connected to the various interfaces of the DUT. By creating virtual sequences, you can easily reuse existing sequence libraries of the underlying interface components and block-level environments to create coordinated system-level scenarios.

In Figure 5-5 on page 91 below, the virtual sequencer invokes configuration sequences on the ethernet and cpu OVCs. The configuration sequences are developed during block-level testing.

Figure 5-5 Virtual Sequence



There are three ways in which the virtual sequencer can interact with its subsequencers:

- "Business as usual"—Virtual subsequencers and subsequencers send transactions simultaneously.
- Disable subsequencers—Virtual sequencer is the only one driving.
- Using `grab()` and `ungrab()`—Virtual sequencer takes control of the underlying driver(s) for a limited time.

When using virtual sequences, most users disable the subsequencers and invoke sequences only from the virtual sequence. For more information, see "Controlling Other Sequencers" on page 93.

To invoke sequences, you can do one of the following:

- Use the appropriate do macro
- Use the sequence `start()` method.

Creating a Virtual Sequencer

For high-level control of multiple sequencers from a single sequencer, use a sequencer that is not attached to a driver and does not process items itself. A sequencer acting in this role is referred to as a virtual sequencer.

To create a virtual sequencer that controls several subsequencers:

1. Derive a virtual sequencer class from the `ovm_sequencer` class.
2. Add references to the sequencers on which the virtual sequences will coordinate the activity. These references will be assigned by a higher-level component (typically the testbench).

The following example declares a virtual sequencer with two subsequencers. Two interfaces called "eth" and "cpu" are created in the `build` function, which will be hooked up to the actual subsequencers.

```
class simple_virtual_sequencer extends ovm_sequencer;
  eth_sequencer eth_seqr;
  cpu_sequencer cpu_seqr;
  // Constructor
  function new(input string name="simple_virtual_sequencer",
    input ovm_component parent=null);
    super.new(name, parent);
    // Automation macro for virtual sequencer (no data item)
    `ovm_update_sequence_lib
  endfunction
  // OVM automation macros for sequencers
  'ovm_sequencer_utils(simple_virtual_sequencer)
endclass: simple_virtual_sequencer
```

Note: The `` `ovm_update_sequence_lib `` macro is used in the constructor when defining a virtual sequencer. This is different than (non-virtual) driver sequencers, which have an associated data item type. When this macro is used, the `ovm_simple_sequence` is not added to the sequencer's sequence library. This is important because the simple sequence only does items, and a virtual sequencer is not connected to a driver that can process the items. For driver sequencers, use the `` `ovm_update_sequence_lib_and_item `` macro. See "Creating the Sequencer" on page 43 for more information.

Subsequencers can be driver sequencers or other virtual sequencers. The connection of the actual subsequencer instances via reference is done later, as shown in "Connecting a Virtual Sequencer to Subsequencers" on page 94.

Creating a Virtual Sequence

Creating a virtual sequence is similar to creating a driver sequence, with the following differences:

- A virtual sequence uses `` `ovm_do_on `` or `` `ovm_do_on_with `` to execute sequences on any of the subsequencers connected to the current virtual sequencer.
- A virtual sequence uses `` `ovm_do `` or `` `ovm_do_with `` to execute other virtual sequences of this sequencer. A virtual sequence cannot use `` `ovm_do `` or `` `ovm_do_with `` to execute items. Virtual sequencers do not have items associated with them, only sequences.

To create a virtual sequence:

1. Declare a sequence class by deriving it from `ovm_sequence`, just like a driver sequence.
2. Define a `body()` method that implements the desired logic of the sequence.
3. Use the `` `ovm_do_on `` (or `` `ovm_do_on_with ``) macro to invoke sequences in the underlying subsequencers.
4. Use the `` `ovm_do `` (or `` `ovm_do_with ``) macro to invoke other virtual sequences in the current virtual sequencer.

The following example shows a simple virtual sequence controlling two subsequencers: a cpu sequencer and an ethernet sequencer. Assume that the cpu sequencer has a `cpu_config_seq` sequence in its library and the ethernet sequencer provides an `eth_large_payload_seq` sequence in its library. The following sequence example invokes these two sequencers, one after the other.

```
class simple_virt_seq extends ovm_sequence;
  ... // Constructor and OVM automation macros
  // See "Creating and Adding a New Sequence" on page 84.
  // A sequence from the cpu sequencer library
  cpu_config_seq conf_seq;
  // A sequence from the ethernet subsequencer library
  eth_large_payload_seq frame_seq;
  // A virtual sequence from this sequencer's library
  random_traffic_virt_seq rand_virt_seq;
  virtual task body();
    // Invoke a sequence in the cpu subsequencer.
    'ovm_do_on(conf_seq, p_sequencer.cpu_seqr)
    // Invoke a sequence in the ethernet subsequencer.
    'ovm_do_on(frame_seq, p_sequencer.eth_seqr)
    // Invoke another virtual sequence in this sequencer.
    'ovm_do(rand_virt_seq)
  endtask : body
endclass : simple_virt_seq
```

Controlling Other Sequencers

When using a virtual sequencer, you will need to consider how you want the subsequencers to behave in relation to the virtual sequence behavior being defined. There are three basic possibilities:

- Business as usual—You want the virtual sequencer and the subsequencers to generate traffic at the same time, using the built-in capability of the original subsequencers. The data items resulting from the subsequencers' default behavior—along with those injected by sequences invoked by the virtual sequencer—will be intermixed and executed in an arbitrary order by the driver. This is the default behavior, so there is no need to do anything to achieve this.

- Disable the subsequencers—Using the `set_config` routines, you can set the count property of the subsequencers to 0, thus disabling their default behavior. Recall that, by default, sequencers start their `ovm_random_sequence`, which uses the count property of the sequencer to determine how many sequences to execute.

  The following code snippet disables the subsequencers in the example in "Connecting a Virtual Sequencer to Subsequencers" on page 94 below.

  ```
  // Configuration: Disable subsequencer sequences.
  set_config_int("*.cpu_seqr", "count", 0);
  set_config_int("*.eth_seqr", "count", 0);
  ```

- Use `grab()` and `ungrab()`—Using `grab()` and `ungrab()`, a virtual sequence can achieve full control over its subsequencers for a limited time and then let the original sequences continue working.

  Note: Only (non-virtual) driver sequencers can be grabbed. Therefore, you should make sure that a given subsequencer is not a virtual sequencer before you attempt to grab it. The following example illustrates this using the functions `is_virtual_sequencer()`, `grab()`, and `ungrab()` in the sequence consumer interface.

  ```
  virtual task body();
    // Grab the cpu sequencer if not virtual.
    if (p_sequencer.cpu_seqr != null)
      p_sequencer.cpu_seqr.grab(this);
    // Execute a sequence.
    'ovm_do_on(conf_seq, p_sequencer.cpu_seqr)
    // Ungrab.
    if (p_sequencer.cpu_seqr != null)
      p_sequencer.cpu_seqr.ungrab(this);
  endtask
  ```

  Note: When grabbing several sequencers, make sure to use some convention to avoid deadlocks. For example, always grab in a standard order.

Connecting a Virtual Sequencer to Subsequencers

To connect a virtual sequencer to its subsequencers:

1. Assign the sequencer references specified in the virtual sequencer to instances of the sequencers. This is a simple reference assignment and should be done only after all components are created.

   ```
   v_sequencer.cpu_seqr = cpu_seqr;
   v_sequencer.eth_seqr = eth_seqr;
   ```

2. Perform the assignment in the `connect()` phase of the verification environment at the appropriate location in the verification environment hierarchy.

The following more-complete example shows a top-level testbench, which instantiates the ethernet and cpu components and the virtual sequencer that controls the two. At the testbench level, the path to the sequencers inside the various components is known and that path is used to get a handle to them and connect them to the virtual sequencer.

```
class simple_tb extends ovm_env;
  cpu_env_c cpu0; // Reuse a cpu verification component.
  eth_env_c eth0; // Reuse an ethernet verification component.
  simple_virtual_sequencer v_sequencer;
  ... // Constructor and OVM automation macros
  virtual function void build();
    super.build();
    // Configuration: Disable subsequencer sequences.
    set_config_int("*.cpu_seqr", "count", 0);
    set_config_int("*.eth_seqr", "count", 0);
    // Configuration: Set the default sequence for the virtual sequencer.
    set_config_string("v_sequencer", "default_sequence",
       simple_virt_seq");
    // Build envs with subsequencers.
    cpu0 = cpu_env_c::type_id::create("cpu0", this);
    eth0 = eth_env_c::type_id::create("eth0", this);
    // Build the virtual sequencer.
    v_sequencer = simple_virtual_sequencer::type_id::create("v_sequencer",
       this);
  endfunction : build

  // Connect virtual sequencer to subsequencers.
  function void connect();
    v_sequencer.cpu_seqr = cpu0.master[0].sequencer;
    v_sequencer.eth_seqr = eth0.tx_rx_agent.sequencer;
  endfunction : connect

endclass: simple_tb
```

Checking for DUT Correctness

Getting the device into desired states is a significant part of verification. The environment should verify valid responses from the DUT before a feature is declared verified. Two types of auto-checking mechanisms can be used:

- Assertions—Derived from the specification or from the implementation and ensure correct timing behavior. Assertions typically focus on signal-level activity.
- Data checkers—Ensure overall device correctness.

As was mentioned in “Monitor” on page 17, checking and coverage should be done in the monitor regardless of the driving logic. Reusable assertions are part of reusable components. See “Developing Reusable Open Verification Components (OVCs)” on page 35 for more information. Designers can also place assertions in the DUT RTL. Refer to your ABV documentation for more information.

This section focuses on data checkers.

Scoreboards

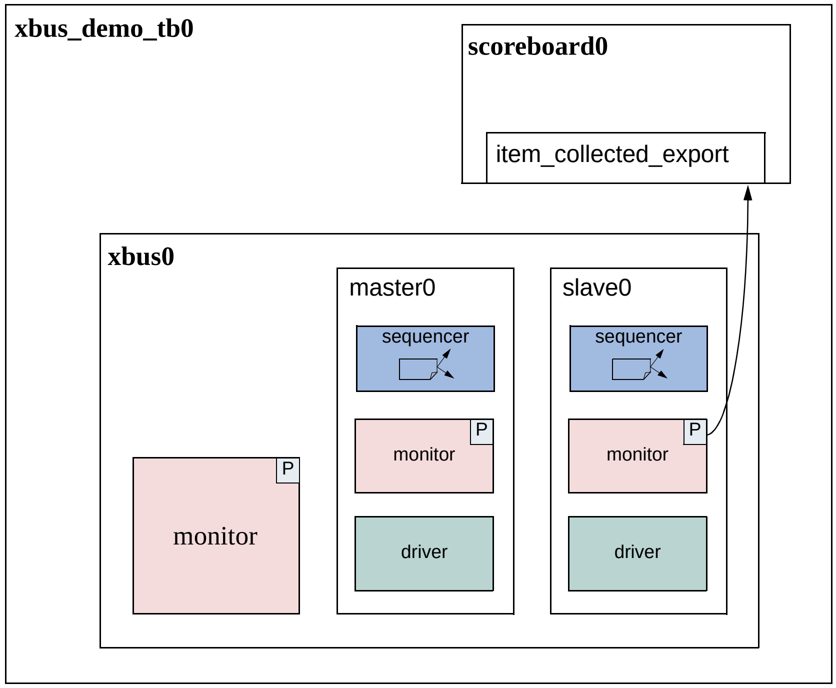
A crucial element of a self-checking environment is the scoreboard. Typically, a scoreboard verifies the proper operation of your design at a functional level. The responsibility of a scoreboard varies greatly depending on the implementation. This section will show an example of a scoreboard that verifies that a given XBus slave interface operates as a simple memory. While the memory operation is critical to the XBus demonstration environment, you should focus on the steps necessary to create and use a scoreboard in an environment so those steps can be repeated for any scoreboard application.

XBus Scoreboard Example

For the XBus demo environment, a scoreboard is necessary to verify that the slave agent is operating as a simple memory. The data written to an address should be returned when that address is read. The desired topology is shown in Figure 5-6 on page 97.

In this example, the user has created a testbench containing one XBus environment that contains the bus monitor, one active master agent, and one active slave agent. Every component in the XBus environment is created using the `build()` methods defined by the IP developer.

Figure 5-6 XBus Demo Environment



Creating the Scoreboard

Before the scoreboard can be added to the `xbus_demo_tb`, the scoreboard component must be defined.

To define the scoreboard:

1. Add the TLM export necessary to communicate with the environment monitor(s).
2. Implement the necessary functions and tasks required by the TLM export.
3. Define the action taken when the export is called.

Adding Exports to ovm_scoreboard

In the example shown in Figure 5-6 on page 97, the scoreboard requires only one port to communicate with the environment. Since the monitors in the environment have provided an analysis port `write()` interface via the TLM `ovm_analysis_port`(s), the scoreboard will provide the TLM `ovm_analysis_imp`.

The `xbus_demo_scoreboard` component derives from the `ovm_scoreboard` and declares and instantiates an `analysis_imp`. For more information on TLM interfaces, see "TLM Interfaces" in the *SystemVerilog OVM Class Reference*. The declaration and creation is done inside the constructor.

```
1 class xbus_demo_scoreboard extends ovm_scoreboard;
2   ovm_analysis_imp #(xbus_transfer, xbus_demo_scoreboard)
3     item_collected_export;
4   ...
5   function new (string name, ovm_component parent);
6     super.new(name, parent);
7     item_collected_export = new("item_collected_export", this);
8   endfunction : new
9   ...
```

Line 2 declares the `ovm_analysis_export`. The first parameter, `xbus_transfer`, defines the `ovm_object` communicated via this TLM interface. The second parameter defines the type of this implementation's parent. This is required so that the parent's `write()` method can be called by the export.

Line 7 creates the implementation instance. The constructor arguments define the name of this implementation instance and its parent.

Requirements of the TLM Implementation

Since the scoreboard provides an `ovm_analysis_imp`, the scoreboard must implement all interfaces required by that export. This means you must define the implementation for the write virtual function. For the `xbus_demo_scoreboard`, `write()` has been defined as:

```
virtual function void write(xbus_transfer trans);
  if (!disable_scoreboard)
    memory_verify(trans);
endfunction : write
```

The `write()` implementation defines what happens when data is provided on this interface. In this case, if `disable_scoreboard` is 0, the `memory_verify()` function is called with the transaction as the argument.

Defining the Action Taken

When the write port is called via `write()`, the implementation of `write()` in the parent of the implementation is called. For more information, see "TLM Interfaces" in the *SystemVerilog OVM Class Reference*. As seen in the previous section, the `write()` function is defined to called the `memory_verify()` function if `disable_scoreboard` is set to 0.

The `memory_verify()` function makes the appropriate calls and comparisons needed to verify a memory operation. This function is not crucial to the communication of the scoreboard with the rest of the environment and will not be discussed. The xbus_demo_scoreboard.sv file shows the implementation.

Adding the Scoreboard to the Environment

Once the scoreboard is defined, the scoreboard can be added to the XBus demo testbench. First, declare the `xbus_demo_scoreboard` inside the `xbus_demo_tb` class.

```
xbus_demo_scoreboard scoreboard0;
```

After the scoreboard is declared, you can construct the scoreboard inside the `build()` phase:

```
function xbus_demo_tb::build();
  ...
  scoreboard0 = xbus_demo_scoreboard::type_id::create("scoreboard0", this);
  ...
endfunction
```

Here, the `scoreboard0` of type `xbus_demo_scoreboard` is created using the `create()` function and given the name “scoreboard0”. It is then assigned the `xbus_demo_tb` as its parent.

After the scoreboard is created, the `xbus_demo_tb` can connect the port on the XBus environment `slaves[0]` monitor to the export on the scoreboard.

```
function xbus_demo_tb::connect();
  ...
  xbus0.slaves[0].monitor.item_collected_port.connect(
     scoreboard0.item_collected_export);
  ...
endfunction
```

This `xbus_demo_tb`'s `connect()` function code makes the connection, using the TLM ports `connect()` interface, between the port in the monitor of the `slaves[0]` agent inside the `xbus0` environment and the implementation in the `xbus_demo_scoreboard` called `scoreboard0`. For more information on the use of binding of TLM ports, see “TLM Interfaces” in the *SystemVerilog OVM Class Reference*.

Summary

The process for adding a scoreboard in this section can be applied to other scoreboard applications in terms of environment communication. To summarize:

1. Create the scoreboard component.
    - Add the necessary exports.
    - Implement the required functions and tasks.
    - Create the functions necessary to perform the implementation-specific functionality.

2. Add the scoreboard to the environment.

    - Declare and instantiate the scoreboard component.
    - Connect the scoreboard implementation(s) to the environment ports of interest.

The XBus demo has a complete scoreboard example. See "XBus OVC Example" on page 133 more information.

Implementing a Coverage Model

In order to ensure thorough verification you need observers to represent your verification goals. SystemVerilog provides a rich set of functional-coverage features.

Selecting a Coverage Method

No single coverage metric ensures completeness. There are two coverage methods:

- Explicit coverage—is user-defined coverage. The user specifies the coverage goals, the needed values, and collection time. As such, analyzing these goals is straightforward. Completing all your coverage goals means that you have achieved 100% of your verification goals and verification has been completed. An example of such a metric is SystemVerilog functional coverage. The disadvantage of such metrics is that missing goals are not taken into account.
- Implicit coverage—is done with automatic metrics that are driven from the RTL or other metrics already existing in the code. Typically, creating an implicit coverage report is straightforward and does not require a lot of effort. For example, code coverage, expression coverage, and FSM (finite-state machine) coverage are types of implicit coverage. The disadvantage of implicit coverage is that it is difficult to map the coverage requirements to the verification goals. It also is difficult to map coverage holes into unexecuted high-level features. In addition, implicit coverage is not complete since it does not take into account high-level abstract events and does not create associations between parallel threads (that is, two or more events occurring simultaneously).

Starting with explicit coverage is recommended. You should build a coverage model that represents your high-level verification goals. Later, you can use implicit coverage as a "safety net" to check and balance the explicit coverage.

Note: Reaching 100% functional coverage with very low code-coverage typically means that the functional coverage needs to be refined and enhanced.

Implementing a Functional Coverage Model

An OVC should come with a protocol-specific functional-coverage model. As a user you may want to disable some coverage aspects that are not important or do not need to be verified. For example, you might not need to test all types of bus transactions in your system, or you might want to remove that goal from the coverage logic that specifies all types of transactions as goals. You might also want to extend the functional-coverage model and create associations between the OVC coverage and other attributes in the system or other interface OVCs. For example, you might want to ensure proper behavior when all types of transactions are sent and the FIFO in the system is full. This would translate into crossing the transaction type with the FIFO-status variable. This section describes how to implement this type of functional coverage model.

Enabling and Disabling Coverage

The verification IP developer should provide configuration properties that allow you to control the interesting aspects of the coverage (see "Controlling Checks and Coverage" on page 67). The VIP documentation will tell you what properties can be set to affect coverage. The most basic of controls would determine whether coverage is collected at all. The XBus monitors demonstrate this level of control. If the you want to disable coverage before the environment is created, use the `set_config_int()` interface.

```
set_config_int("xbus0.masters[0].monitor", "coverage_enable", 0);
```

Once the environment is created, you can set this property directly.

```
xbus0.masters[0].monitor.coverage_enable = 0;
```

This is a simple Verilog assignment to a class property (or variable).

Advanced Topics

This chapter discusses OVM topics and capabilities of the SystemVerilog OVM Class Library that are beyond the essential material covered in the previous chapters. Consult this chapter as needed if you require more detailed information. This chapter discusses:

The ovm_component Base Class

All the infrastructure components in an OVM verification environment, including environments and tests, are derived either directly or indirectly from the `ovm_component` class. User-defined classes derived from this class inherit built-in automation. Typically, you will derive your classes from the methodology classes, which are themselves extensions of `ovm_component`. However, understanding the `ovm_component` is important because many of the facilities that the methodology classes offer are derived from this class.

Note: The `ovm_threaded_component` class has been deprecated in OVM 2.0 and is now simply a typedef for `ovm_component`.

The following sections describe some of the capabilities that are provided by the `ovm_component` base class and how to use them.The key pieces of functionality provided by the `ovm_component` base class include:

- Phasing and execution control.
- Configuration methods.
- Factory convenience methods.
- Hierarchical reporting control.

Simulation Phase Methods

The SystemVerilog OVM Class Library provides built-in simulation phase methods. These phases are hooks for you to include logic to be executed at critical points in time. For example, if you need checking logic to be executed at the end of the simulation, you can extend the `check()` phase and embed procedural code in it. Your code then will be executed at the desired time during simulation. See ovm_phase in the *SystemVerilog OVM Class Reference* for more information on using built-in phases.

From a high-level view, the existing simulation phases (in simulation order) are:

- "build()" on page 104.
- "connect()" on page 105.
- "end_of_elaboration()" on page 105.
- "start_of_simulation()" on page 105.
- "run()" on page 105.
- "extract()" on page 106.
- "check()" on page 107.
- "report()" on page 107.

build()

The first phase of the OVM phasing mechanism is the `build()` phase, which is called automatically for all components in a top-down fashion. The `build()` method creates its component's child components and optionally configures them. Since `build()` is called top-down, the parent's configuration calls will be completed before the child's `build()` method is called. Although not recommended, a parent component may explicitly call `build()` on its children as part of the `parent.build()`.

The top-down execution order allows each parent's `build()` method to configure or otherwise control child parameters before the child components' `build()` method is executed. To ensure that `build()` does not get called twice in this case, every `build()` implementation should call `super.build()` as the first statement of `build()`.

This phase is a function and executes in zero time.

```
class my_comp extends ovm_component;
  ...
  virtual void function build();
    super.build();
```

```
    // Get configuration information.
    // Create child components.
    // configure child components
  endfunction

  ...
endclass
```

connect()

The `connect()` phase is executed after `build()`. Because the environment is created during the component's `build()` in a top-down fashion, the user may rely on the fact that the hierarchical test/environment/component topology has been fully created when `connect()` is called.

This phase is a function and executes in zero time.

```
class my_comp extends ovm_component;
  ...
  virtual void function connect();
    if(is_active == OVM_ACTIVE)
      driver.seq_item_port.connect(sequencer.seq_item_export);
    for(int i = 0; i<num_subscribers; i++)
      monitor.analysis_port.connect(subscr[i].analysis_export);
    ...
  endfunction

  ...
endclass
```

end_of_elaboration()

The `end_of_elaboration()` phase allows you to make final adjustments to the environment after it has been built and connected. The user can assume that the entire environment is created and connected before this method is called. This phase is a function and executes in zero time.

start_of_simulation()

The `start_of_simulation()` phase provides a convenient place to perform any pre-`run()` activity such as displaying banners, printing final testbench topology and configuration information.This phase is a function and executes in zero time.

run()

The `run()` phase is the only predefined time-consuming phase, which defines the implementation of a component’s primary run-time functionality. Implemented as a task, it can fork other processes.

When a component returns from its `run` task, it does not signify completion of its `run` phase. Any processes that it may have forked *continue to run*. The run phase terminates in one of three ways:

- **stop**—When a component's `enable_stop_interrupt` bit is set and `global_stop_request` is called, the component's `stop` task is called. Components can implement `stop` to allow completion of in-progress transactions, flush queues, and so on. Upon return from `stop` by all enabled components, a `kill` is issued.
- **kill**—When called, all component's `run` processes are killed immediately. While `kill` can be called directly, it is recommended that components use the stopping mechanism. This affords a more ordered and safe shutdown.
- **timeout—**If a timeout was set, the phase ends if it expires before either `stop` or `kill` occur.

The following describe the `run()` phase task of sequencer and driver components.

- **Sequencer—**The sequencer generates stimulus data, passes it to the driver for execution, and starts the default sequence. The sequencer generates a data item with the specified constraints and randomization and passes it to the driver. This activity is handled by the SystemVerilog OVM Class Library automatically.
- **Driver—**When reset is deasserted, the driver gets the next item to be performed from the sequencer and drives the HDL signals as per the protocol. Once the current item is completed, the driver gives the "item done" indication. A driver in a proactive agent (master) initiates transfers on the bus according to test directives. A driver in a reactive agent (slave) responds to transfers on the bus rather than initiating actions. This activity is specified by the user.

extract()

This phase can be used to extract simulation results prior to checking in the next phase. Typically, it is used for user-defined activities such as processing the simulation results. Following are some examples of what you can do in this phase.

- Collect assertion-error count.
- Extract coverage information.
- Extract the internal signals and register values of the DUT.
- Extract internal variable values from components.
- Extract statistics or other information from components.

This phase is a function and executes in zero time. It is called in bottom-up order.

check()

Having extracted vital simulation results in the previous phase, the `check` phase can be used to validate such data and determine the overall simulation outcome. This phase is a function and executes in zero time. It is called in bottom-up order.

report()

This phase executes last and is used to output results to files and/or the screen. This phase is a function and executes in zero time. It is called in bottom-up order.

Adding User-Defined Phases

In addition to the predefined phases listed above, OVM provides the `ovm_phase` base class that allows you to add your own phases anywhere in the list.

To define a new phase:

1. Derive a subclass of `ovm_phase` that implements either the `call_task()` or `call_func` method, depending on whether the new phase is to be time-consuming (a task) or not (a function).

```
1  class my_comp extends ovm_component;
2    ...
3    virtual my_task();  return; endtask // make virtual
4    ...
5  endclass
6
7  class my_task_phase extends ovm_phase;
8    function new();
9      super.new("my_task",1,1);
10   endfunction
11   task call_task(ovm_component parent);
12     my_comp_type my_comp;
13     if ($cast(my_comp,parent))
14       my_comp.my_task_phase()
15   endtask
16   virtual function string get_type_name ();
17     return "my_task";
18   endfunction
19 endclass
```

Line 9 When calling `super.new()` the new subclass must provide three arguments:

- The name of the phase, which is typically the name of the callback method.
- A bit to indicate whether the method is to be called top-down (1) or bottom-up (0).
- A bit to indicate whether the method is a task (1) or a function (0).

Note: OVM includes several macros to simplify the definition of new phases:

```
‘define ovm_phase_task_decl(NAME,TOP_DOWN)
‘define ovm_phase_func_topdown_decl(NAME) ‘ovm_phase_func_decl(NAME,1)
‘define ovm_phase_func_bottomup_decl(NAME) ‘ovm_phase_func_decl(NAME,0)
‘define ovm_phase_task_topdown_decl(NAME) ‘ovm_phase_task_decl(NAME,1)
‘define ovm_phase_task_bottomup_decl(NAME) ‘ovm_phase_task_decl(NAME,0)
```

2. Declare an instance of the new phase object

```
my_task_phase my_task_ph = new();
```

3. Register the phase with the OVM phase controller, `ovm_top`.

```
ovm_top.insert_phase(my_task_ph, run_ph);
```

The second argument, `run_ph`, is the phase after which the new phase will be inserted.To insert a phase at the beginning of the list, this argument should be NULL.

The Built-In Factory and Overrides

About the Factory

OVM provides a built-in factory to allow components to create objects without specifying the exact class of the object being creating. The factory provides this capability with a static allocation function that you can use instead of the built-in `new` function. The function provided by the factory is:

`type_name``::type_id::create(string `*`name`*`, ovm_component `*`parent`*`)`

Since the `create()` method is automatically type-specific, it may be used to create components or objects. When creating objects, the second argument, *`parent`*, is optional.

A component using the factory to create data objects would execute code like the following:

```
task mycomponent::run();
  mytype data;  // Data must be mytype or derivative.
  data = mytype::type_id::create("data");
  $display("type of object is: %0s", data.get_type_name());
  ...
endtask
```

In the code above, the component requests an object from the factory that is of type “mytype” with an instance name of “data”.

When the factory creates this object, it will first search for an instance override that matches the full instance name of the object. If no instance-specific override is found, the factory will search for a type-wide override for the type “mytype”. If no type override is found then the type created will be of type “mytype”.

Factory Registration

You must tell the factory how to generate objects of specific types. In OVM, there are a number of ways to do this allocation.

- Use the `` `ovm_object_utils(T) `` or `` `ovm_component_utils(T) `` macro in a derivative `ovm_object` or `ovm_component` class declaration, respectively. These macros expand code which will register the given type with the factory. The argument `T` may be a parameterized type

```
`ovm_object_utils(packet)
‘ovm_component_utils(my_driver)
```

- Use the registration macros `` `ovm_object_registry(T,S) `` or `` `ovm_component_registry(T,S) ``. These macros can appear anywhere in the declaration space of the class declaration of `T` and will associate the string `S` to the object type `T`. These macros are called by the corresponding `ovm_*_utils` macros, so you might use them only if you do not use the `ovm_*_utils` macros.

Component Overrides

A global factory is provided that allows you to substitute a predefined-component type with some other type that is specialized for your needs, without having to derive the container type. The factory can replace a component type within the component hierarchy without changing any other component in the hierarchy.

A global factory is available for this purpose. You need to know how to use the factory, but not how the factory works.

Note: All type-override code should be executed in a parent prior to building the child(ren). This means that environment overrides should be specified in the test.

Two interfaces, `set_type_override_by_type` and `set_inst_override_`by_type, exist to replace default components. These interfaces will be examined one at a time.

To override a default component:

1. Define a class that derives from the appropriate OVM base class.
2. Execute the override (described in the following sections).
3. Build the environment.

Type Overrides

The first component override replaces all components of the specified type with the new specified type. The prototype is.

```
set_type_override_by_type(orig_type, override_type, bit replace = 1);
```

The first argument (`orig_type`) is the type, obtained by calling the static `get_type()` method of the type (`orig_type::get_type()`). That type will be overridden by the second argument (`override_type::get_type()`). The third argument, `replace`, determines whether to replace an existing override (`replace` = 1). If this bit is 0 and an override of the given type does not exist, the override is registered with the factory. If this bit is 0 and an override of the given type does exist, the override is ignored.

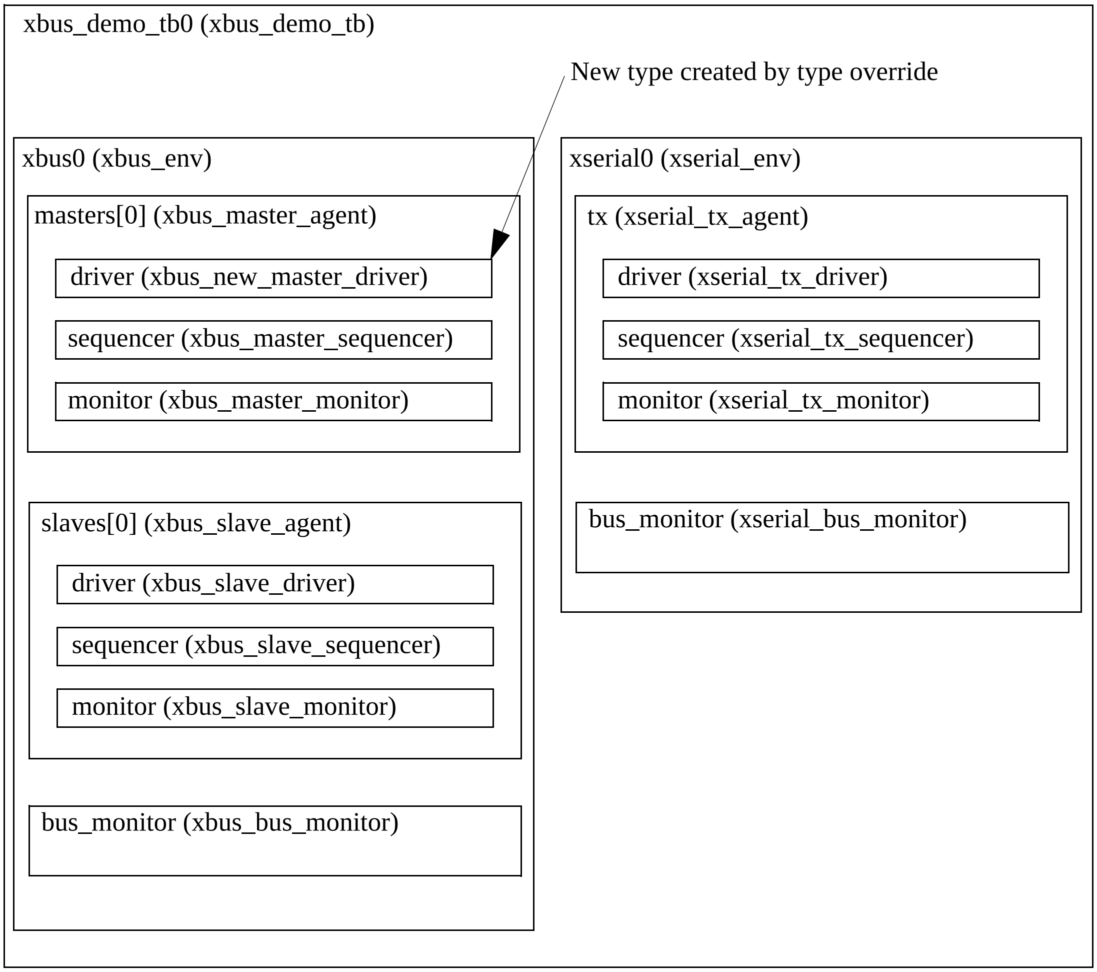
If no overrides are specified, the environment will be constructed using default types. For example, the environment would be created using an `xbus_master_driver` type component inside `xbus_master_agent.build()`. The `set_type_override_by_type` interface allows you to override this behavior in order to have an `xbus_new_master_driver` for all instances of `xbus_master_driver`.

```
set_type_override_by_type(xbus_master_driver::get_type(),
   xbus_new_master_driver::get_type);
```

This overrides the default type (`xbus_master_driver`) to be the new type (`xbus_new_master_driver`). In this case, we have overridden the type that is created when the environment should create an `xbus_master_driver`. The complete hierarchy would now be built as shown in Figure 6-1 on page 111.

Note: While only one `xbus_master_driver` instance is replaced in this example, any and all `xbus_master_driver` instances would be replaced in an environment containing multiple `xbus_master_drivers`.

Figure 6-1 Hierarchy Created with set_type_override() Applied



Instance Overrides

The second component override replaces targeted components of the matching instance path with the new specified type. The prototype for `ovm_component` is

```
set_inst_override_by_type(string inst_path, orig_type, override_type);
```

The first argument, `inst_path`, is the relative component name of the instance override. It can be considered the “target” of the override. The second argument, `orig_type`, is the type to be overridden (specified by `orig_type::get_type()`) and replaced by the type specified by the last argument, `override_type` (also using `override_type::get_type()`).

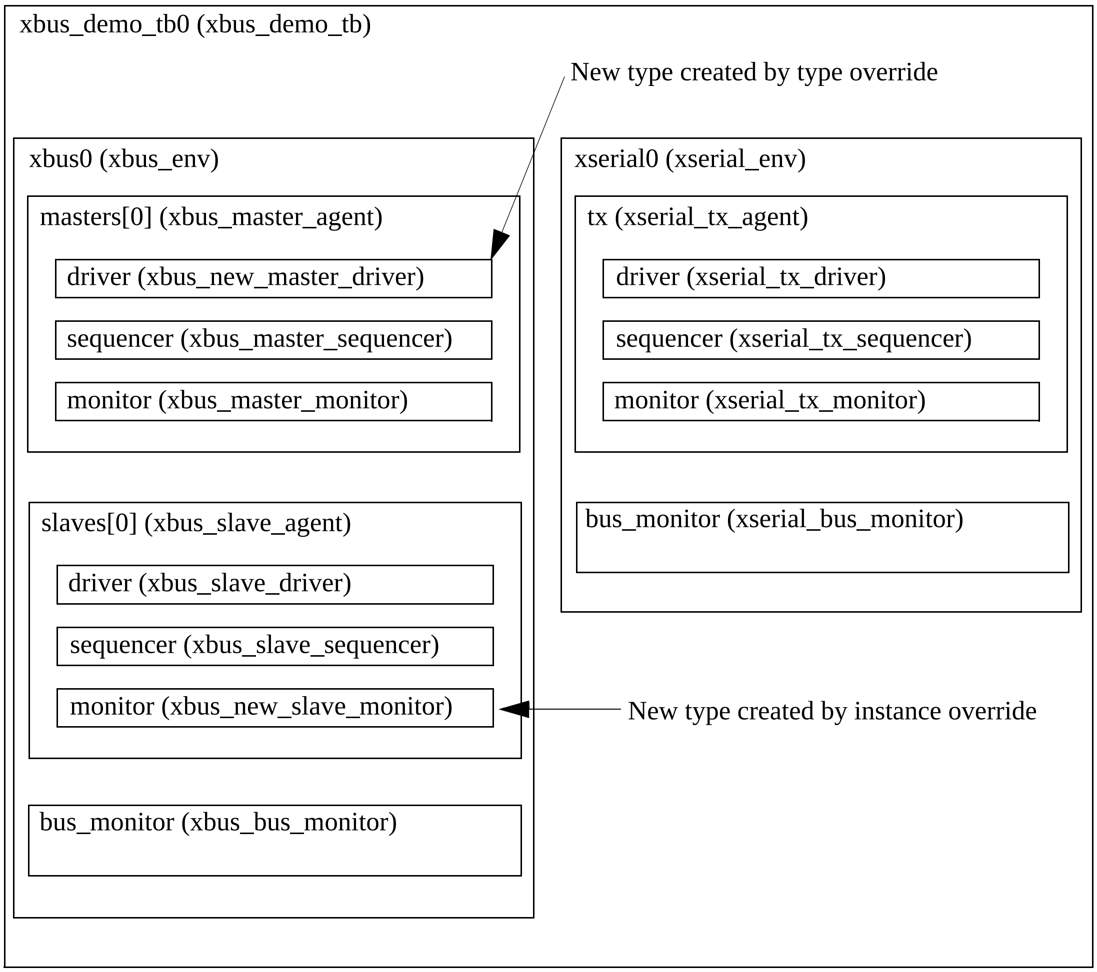
Assume the `xbus_new_slave_monitor` has already been defined. Once the following code is executed, the environment will now create the new type, `xbus_new_slave_monitor`, for all instances that match the instance path.

```
set_inst_override_by_type("slaves[0].monitor",
   xbus_slave_monitor::get_type(), xbus_new_slave_monitor::get_type());
```

In this case, the type is overridden that is created when the environment should create an `xbus_slave_monitor` for only the `slaves[0].monitor` instance that matches the instance path in the override. The complete hierarchy would now be built as shown in Figure 6-2 on page 112.

For illustration purposes, this hierarchy assumes both overrides have been executed.

Figure 6-2 Hierarchy Created with Both Overrides Applied



Note: Instance overrides are used in a first-match order. For each component, the first applicable instance override is used when the environment is constructed. If no instance overrides are found, then the type overrides are searched for any applicable type overrides. The ordering of the instance overrides in your code affects the application of the instance overrides. You should execute more-specific instance overrides first. For example,

```
set_inst_override_by_type("a.b.*", mytype::get_type(),
                                        newtype::get_type());
set_inst_override_by_Type("a.b.c", mytype::get_type(),
                                        different_type::get_type());
```

will create `a.b.c` with `different_type`. All other objects under `a.b` of `mytype` are created using `newtype`. If you switch the order of the instance override calls then all of the objects under `a.b` will get "`newtype`" and the instance override `a.b.c` is ignored.

```
set_inst_override_by_type("a.b.c", mytype::get_type(),
                                        different_type::get_type());
set_inst_override_by_type("a.b.*", mytype::get_type(),
                                        newtype::get_type());
```

Advanced Sequence Control

This section discusses advanced techniques for sequence control. It contains the following subsections:

Implementing Complex Scenarios

This section contains the following subsections:

Executing Multiple Sequences Concurrently

There are two ways you can create concurrently-executing sequences:

- Using the ovm_do Macros with fork/join.
- Starting Several Sequences in Parallel using the start() method.

The following sections show an example of each method.

Using the ovm_do Macros with fork/join

In this example, the sequences are executed with fork/join. The simulator schedules which sequence requests interaction with the sequencer. The sequencer schedules which items are provided to the driver, arbitrating between the sequences that are willing to provide an item for execution and selects them one at a time. The `a` and `b` sequences are subsequences of the `fork_join_sequence`.

```
class fork_join_sequence extends ovm_sequence #(simple_item);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  a_seq a;
  b_seq b;
  virtual task body();
    fork
      `ovm_do(a)
      `ovm_do(b)
    join
  endtask : body
endclass : fork_join_sequence
```

Starting Several Sequences in Parallel

In this example, the `concurrent_seq` sequence activates two sequences in parallel. It does not wait for the sequences to complete. Instead, it immediately finishes after activating the sequences. Also, the `a` and `b` sequences are started as root sequences.

```
class concurrent_seq extends ovm_sequence #(simple_item);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  a_seq a;
  b_seq b;
  virtual task body();
    // Initialize the sequence variables with the factory.
    `ovm_create(a)
    `ovm_create(b)
    // Start each subsequence as a new thread.
    fork
      a.start(p_sequencer);
      b.start(p_sequencer);
    join
  endtask : body
endclass : concurrent_seq
```

Note: The `sequence.start()` method allows the sequence to be started on any sequencer.

See ovm_create in the *SystemVerilog OVM Class Reference* for additional information.

Using the pre_body() and post_body() Callbacks

The SystemVerilog OVM Class Library provides two additional callback tasks, `pre_body()` and `post_body()`, which are invoked before and after the sequence's `body()` task, respectively. These callbacks are invoked only when a sequence is started by its sequencer's `start_sequence()` task or the sequence's `start()` task.

Examples for using the `pre_body()` and `post_body()` callbacks include:

- Synchronization to some event before the `body()` task starts.
- Calling a cleanup task when the `body()` task ends.

The following example declares a new sequence type and implements its callback tasks.

```
class revised_seq extends fork_join_sequence;
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  task pre_body();
    super.pre_body();
    // Wait until initialization is done.
    @p_sequencer.initialization_done;
  endtask : pre_body
  task post_body();
    super.post_body();
    do_cleanup();
  endtask : post_body
endclass : revised_seq
```

The `pre_body()` and `post_body()` callbacks are not invoked in a sequence that is executed by one of the `` `ovm_do `` macros.

Note: The `initialization_done` event declared in the sequencer can be accessed directly via the `p_sequencer` variable. The `p_sequencer` variable is available since the `` `ovm_sequence_utils `` macro was used. This prevents the user from having to declare a variable of the appropriate type and initialize it using `$cast`.

Interrupt Sequences

A DUT might include an interrupt option. Typically, an interrupt should be coupled with some response by the agent. Once the interrupt is serviced, activity prior to the interrupt should be resumed from the point where it was interrupted. Your verification environment can support interrupts using sequences.

To handle interrupts using sequences:

1. Define an interrupt handler sequence that will do the following:
   a. Wait for the interrupt event to occur.
   b. Grab the sequencer for exclusive access.
   c. Execute the interrupt service operations using the proper items or sequences.

d. Ungrab the sequencer.

2. Start the interrupt-handler sequence in the sequencer or in the default sequence. (You can configure the sequencer to run the default sequence when the simulation begins.)

Example

1. Define an interrupt handler sequence.

```
// Upon an interrupt, grab the sequencer, and execute a
// read_status_seq sequence.
class interrupt_handler_seq extends ovm_sequence #(bus_transfer);
  ... // Constructor and OVM automation macros here
    // See "Creating and Adding a New Sequence" on page 84.
  read_status_seq stat_seq;
  virtual task body();
    forever begin
      // Initialize the sequence variables with the factory.
      @p_sequencer.interrupt;
      grab(p_sequencer);
      `ovm_do(stat_seq)
      ungrab(p_sequencer);
    end
  endtask : body
endclass : interrupt_handler_seq
```

2. Start the interrupt handler sequence in the sequencer. The example below does this in the sequencer itself at the run phase:

```
class my_sequncer extends ovm_sequencer;
  ... // Constructor and OVM automation macros here
      // See "Creating and Adding a New Sequence" on page 84.
  interrupt_handler_seq interrupt_seq;
  virtual task run();
    interrupt_seq =
       interrupt_handler_seq::type_id::create("interrupt_seq");
    interrupt_seq.start(this);
    super.run();
  endtask : run
endclass : my_sequncer
```

Note: In this step, we cannot use any of the `` `ovm_do `` macros since they can be used only in sequences. Instead, we use utility functions in the sequencer itself to create an instance of the interrupt handler sequence through the common factory.

Controlling the Scheduling of Items

There might be several sequences doing items concurrently. However, the driver can handle only one item at a time. Therefore, the sequencer maintains a queue of do actions. When the driver requests an item, the sequencer chooses a single do action to perform from the do actions waiting in its queue. Therefore, when a sequence is doing an item, the do action is blocked until the sequencer is ready to choose it.

The scheduling algorithm works on a first-come-first-served basis. You can affect the algorithm using `grab()`, `ungrab()`, and `is_relevant()`.

If a sequence is grabbing the sequencer, then the sequencer will choose the first do action that satisfies the following conditions:

- It is done by the grabbing sequence or its descendants.
- The `is_relevant()` method of the sequence doing it returns 1.

If no sequence is grabbing the sequencer, then the sequencer will choose the first do action that satisfies the following condition:

The `is_relevant()` method of the sequence doing it returns 1.

If there is no do action to choose, then `get_next_item()` is blocked. The sequencer will try to choose again (that is, reactivate the scheduling algorithm) when one of the following happens:

- Another do action is added to the queue.
- A new sequence grabs the sequencer, or the current grabber ungrabs the sequencer.
- Any one of the blocked sequence's `wait_for_relevant()` task returns. See "Run-Time Control of Sequence Relevance" on page 117 for more information.

When calling `try_next_item()`, if the sequencer does not succeed in choosing a do action before the time specified by `ovm_driver::wait_for_sequences()` elapses, then `ovm_driver::try_next_item()` returns with null.

Run-Time Control of Sequence Relevance

In some applications, it is useful to invoke sequences concurrently with other sequences and have them execute items under certain conditions. Such a sequence can therefore become relevant or irrelevant, based on the current conditions, which may include the state of the DUT, the state of other components in the verification environment, or both. To implement this, you can use the sequence `is_relevant()` function. Its effect on scheduling is discussed in "Controlling the Scheduling of Items" on page 116.

If you are using `is_relevant()`, you must also implement the `wait_for_relevant()` task to prevent the sequencer from hanging under certain circumstances. The following example illustrates the use of both.

```
class flow_control_seq extends ovm_sequence #(bus_transfer);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  bit relevant_flag;
  function bit is_relevant();
```

```
    return(relevant_flag);
  endfunction
  // This task is started by the sequencer if none of the running
  // sequences is relevant. The task must return when the sequence
  // becomes relevant again.
  task wait_for_relevant();
    while(!is_relevant())
      @(relevant_flag); // Use the appropriate sensitivity list.
  endtask
  task monitor_credits();
    ...
    // Logic goes here to monitor available credits, setting
    // relevant_flag to 1 if enough credits exist to send
    // count frames, 0 otherwise.
  endtask : monitor_credits
  task send_frames();
    my_frame frame;
    repeat (count) `ovm_do(frame)
  endtask : send_frames
  virtual task body();
    fork
      monitor_credits();
      send_frames();
    join_any
  endtask : body
endclass : flow_control_seq
```

Protocol Layering

This section discusses the layering of protocols and how to implement it using sequences.

This section includes:

Introduction to Layering

Some verification environments require layering of data items of different protocols. Examples include TCP over IP and ATM over Sonet. Sequence layering and virtual sequences are two ways in which sequencers can be composed to create a layered protocol implementation.

Layering of Protocols

The classic example of protocol layering can be described by generic higher- and lower-levels (or layers) of a protocol. An array of bytes may be meaningless to the lower-level protocol, while in the higher-level protocol context, the array provides control and data messages to be processed appropriately.

For example, assume that there are two sequencers. The low-layer sequencer drives `lower_layer_items`, that are defined as:

```
class lower_layer_item extends ovm_sequence_item;
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  bit[`MAX_PL:0][`DATA_SIZE-1:0] payload;
endclass : lower_layer_item
```

The low-level sequences base class is defined as:

```
class lower_layer_seq_base extends ovm_sequence #(lower_layer_item);
  ... // Constructor and OVM automation macros go here.
      // See "Using Sequences" on page 83.
  lower_layer_item item;
  virtual task body();
    ...
  endtask : body
endclass : lower_layer_seq_base
```

In one case, you want to send `lower_layer_items` with random data. In another case, you want the data to come from a higher-layer data protocol. The higher-layer protocol in this example drives `higher_layer_items` which will be mapped to one or more `lower_layer_items`. Therefore, the high-level sequence base class is defined as:

```
class higher_layer_seq_base extends ovm_sequence #(higher_layer_item);
  ... // Constructor and OVM automation macros
      // See "Using Sequences" on page 83.
  higher_layer_item item;
  virtual task body();
    ...
  endtask : body
endclass : higher_layer_seq_base
```

Layering and Sequences

Layering is best implemented with sequences. There are two ways to do layering using sequences:

- "Layering Inside One Sequencer" on page 119 applies for simple cases only.
- "Using Layered Sequencers" on page 125 applies for all layering.

Layering Inside One Sequencer

For simple cases, you can layer inside one sequencer by generating a data item of the higher layer within a lower-layer sequence. Do this by creating another sequence kind for the lower-layer sequencer. For example:

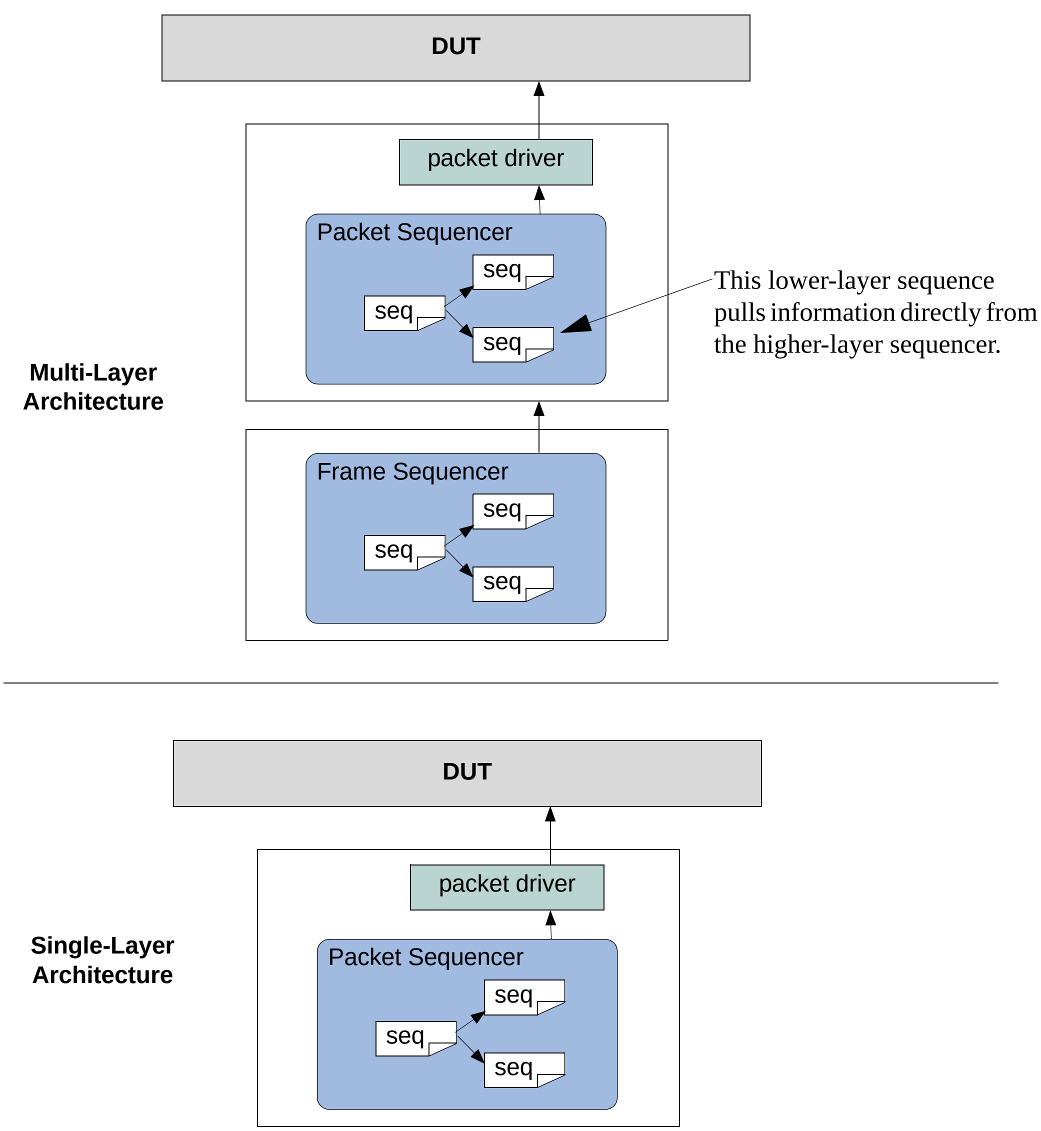
```
class use_higher_level_item_seq extends lower_layer_base_seq;
  ... // Constructor and OVM automation macros go here.
      // See "Using Sequences" on page 83.
  higher_layer_item hli;
  lower_layer_item lli;
  task body();
    // Create a higher-level item.
    `ovm_create(hli)
    ... // Randomize it here.
    send_higher_level_item(hli);
  endtask : body
  task send_higher_level_item(higher_layer_item hli);
    for(int i = 0 ; i< hli.length; i++) begin
      // Convert the higher-level item to lower-level items and send.
      `ovm_create(lli);
      ... // Slice and dice hli to form property values of lli.
      `ovm_send(lli)
    end
  endtask : send_higher_level_item
endclass: use_higher_level_item_seq
```

The `use_higher_level_item_seq` sequence generates a single `higher_layer_item` and sends it in chunks, in one or more `lower_layer_items`, until the data of the `higher_layer_item` is exhausted. See ovm_create in the *SystemVerilog OVM Class Reference* for more information.

Layering of Several Sequencers

This general approach to layering several sequencers uses multiple sequencers as shown in Figure 6-3 on page 121 below.

Figure 6-3 Layering Architecture



Taking the `higher_layer_item` and `lower_layer_item` example, there is a lower-layer sequence and a higher-layer sequence (complete with their sequencers). The lower-layer sequence pulls data from the higher-layer sequencer (or from the higher-layer driver).

Each sequencer can be encapsulated in an OVC so that layering can be done by connecting the OVCs.

Styles of Layering

This section includes the following sections:

Basic Layering

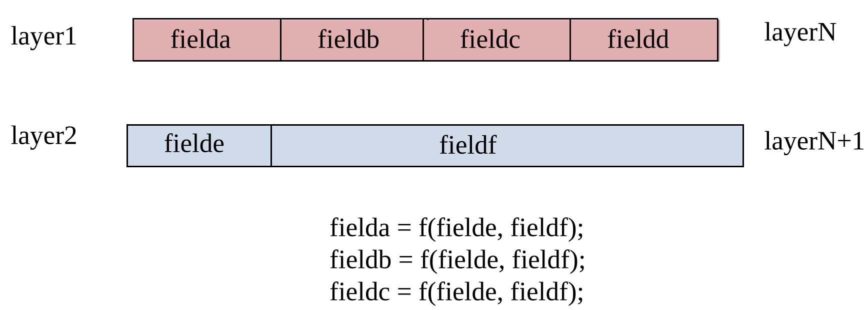
The simplest general scenario of basic layering consists of:

- The driver accepts layer1 items.
- The layer1 items are constructed from layer2 items in some way. The layer2 items are, in turn, constructed from layer3 items, and so on.
- For every layerN and layerN+1, there is a mechanism that takes layerN+1 items and converts them into layerN items.

You can also have multiple kinds of layer1 and layer2 items. In different configurations, you might want to layer any kind of layer2 item over any kind of layer1 item.

The remainder of this section describes possible variations and complications, depending on the particular protocol or on the desired test-writing flexibility.

Figure 6-4 Layering of Protocols

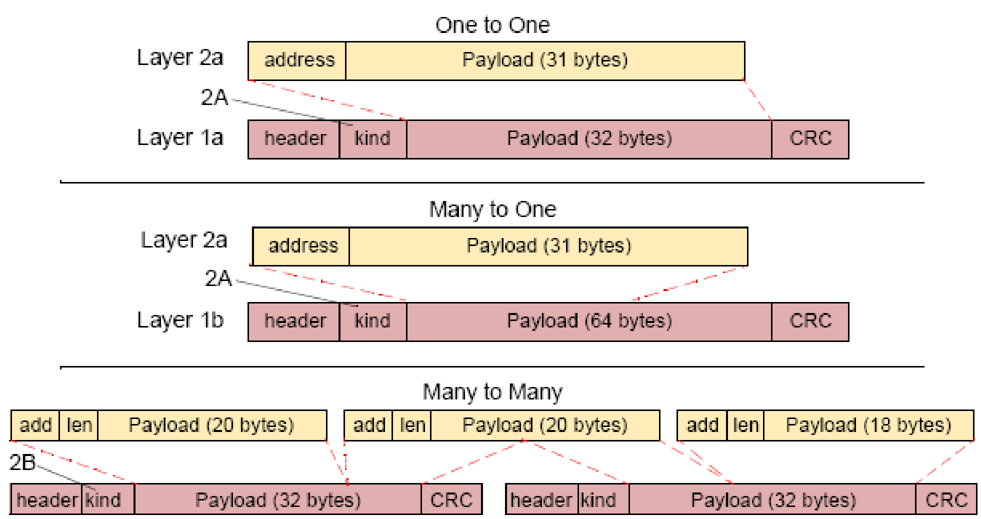


One-to-One, One-to-Many, Many-to-One, Many-to-Many

A conversion mechanism might need to cope with the following situations (see Figure 6-5 on page 123):

- One-to-one—One high-layer item must be converted into one low-layer item.
- One-to-many—One large high-layer item must be broken into many low-layer items.
- Many-to-one—Many high-layer items must be combined into one large low-layer item (as in Sonet, for example).
- Many-to-many—Multiple higher-layer items must be taken in and converted into multiple lower-layer items. For example, high-layer packets are ten-bytes long, and low-layer packets are three to 35 bytes long. In this case, there could be remainders.

Figure 6-5 Layer Mapping



Different Configurations at Pre-Run Generation and Run Time

A system might need to support different modes of operation defined by topology, data type, or other application-specific requirements. For example, in one environment, you might have only layer1 items. In another environment, layer1 items would be dictated by layer2 items. You might also want to decouple the layers further, for example, so that layer2 items could drive either layer1 items or layer1 cells (on another interface) or both.

At times you might have a mix of inputs from multiple sources at run time. For example, you might want to have one low-layer sequencer send items that come from several high-layer sequencers.

Timing Control

In some configurations, the high-layer items drive the timing completely. When high-layer items are created, they are immediately converted into low-layer items.

In other configurations, the low-layer sequences pace the operation. When a low-layer do macro is executed, the corresponding high-layer item should appear in zero time.

Finally, there is a case where items are driven to the DUT according to the timing of the low-layer sequences, but the high-layer sequences are not reacting in zero time. Rather, if there is no data available from the high-layer sequences, then some default value (for example, a zero filler) is used instead. `ovm_driver::try_next_item()` would be used by the lower-level driver in this case.

Data Control

In some configurations, the high-layer items completely dictate which low-layer items reach the DUT. The low layer simply acts as a slave.

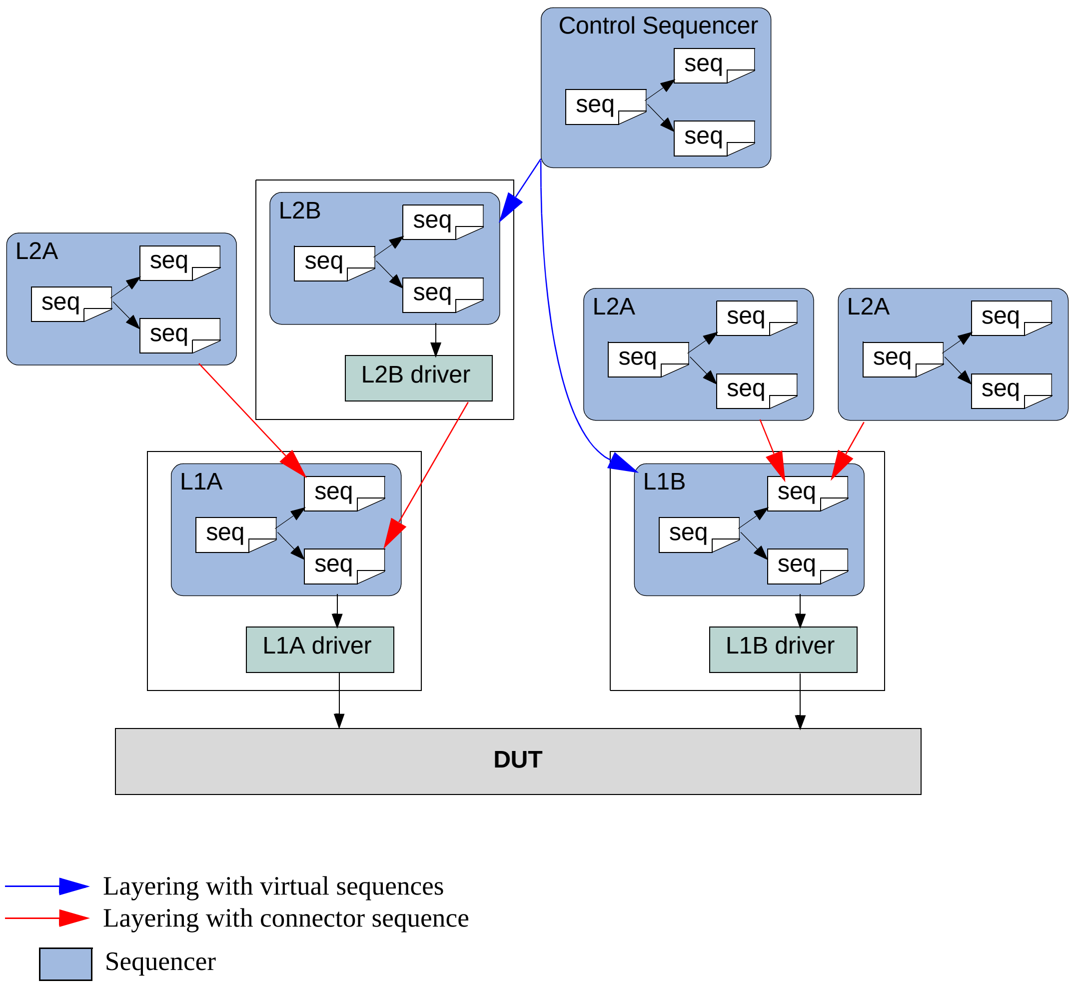
Often, however, both layers influence what reaches the DUT. For example, the high layer might influence the data in the payload while the low layer influences other attributes of the items reaching the DUT. In these cases, the choice of sequences for both layers is meaningful.

Controlling Sequences on Multiple Sequencers

In the most general case, you have a graph consisting of several sequencers, some of which may control sequence execution on other sequencers and some of which may generate items directly. Some low-layer "driver sequencers" are connected to the DUT, some higher-layer driver sequencers are layered above them, and some sequencers on top feed into all of the driver sequencers below.

In the example configuration shown in Figure 6-6 on page 125, a low-layer sequencer (L1B) gets input from multiple high-layer sequencers (two instances of L2A) as well as from a controlling sequencer.

Figure 6-6 Most-General Case of Using Virtual Sequencers



Using Layered Sequencers

Layered sequencers work as follows:

- Higher-layer sequencers operate as usual, generating upper-layer data items and sending them through the `seq_item_pull_export`. In most cases, you will not need to change the upper-layer sequencer or sequences that will be used in a layered application.
- The lower-layer sequencers connect to the higher-layer sequencer(s) from which information must be pulled. The pulled information (a higher-layer item) is put in a property of the sequence and is then used to constrain various properties in the lower-layer item(s). The actual connectivity between the layers is done in the same manner as the connection between a sequencer and a

driver. To connect to the higher-layer sequencer, you must declare a corresponding `ovm_seq_item_pull_port` in the lower-layer sequencer (see Example 6-1 on page 126). The connection itself is performed at the time the containing object's `connect()` method is invoked.

- The lower-layer sequencers send information to a lower-layer driver that interacts with a DUT's physical interface.

Assuming you already have created (or are reusing) upper-layer and lower-layer sequencers, follow these steps below to create the layering.

To layer sequencers:

1. Create a lower-layer sequence which does the following:
   - Repeatedly pulls upper-layer items from the upper-layer sequencer.
   - Translates them to lower-layer items.
   - Sends them to the lower-layer driver.

   To preserve late generation of the upper-layer items, pull the upper-layer items from within the lower-sequence's `pre_do()` task. This ensures that the upper-layer item will be randomized only when the lower-layer driver is ready to start processing the matching lower-layer items.

2. Connect the lower-layer sequencer to the upper-layer sequencer using the same technique as when connecting a driver to a sequencer.

3. Configure the lower-layer sequencer's default sequence to be the sequence you created in step 1 above.

Example 6-1 Layer Sequencers Example

Assume you are reusing the upper- and lower-layer classes from components created earlier. The lower-layer components are likely to be encapsulated inside an agent modeling the interface protocol. This example shows how to achieve layering without introducing the recommended reuse structure to keep the code compact.

```
// Upper-layer classes
class upper_item extends ovm_sequence_item;
  ...
endclass : upper_item
class upper_sequencer extends ovm_sequencer #(upper_item);
  ...
endclass : upper_sequencer


// Lower-layer classes
class lower_item extends ovm_sequence_item;
```

```
  ...
endclass : lower_item

class lower_sequencer extends ovm_sequencer #(lower_item);
  ovm_seq_item_pull_port #(upper_item) upper_seq_item_port;
  ...
  function new (string name, ovm_component parent);
    super.new(name, parent);
    upper_seq_item_port = new(“upper_seq_item_port”,this);
    `ovm_update_sequence_lib_and_item(...)
  endfunction : new
  ...
endclass : lower_sequencer

class lower_driver extends ovm_driver #(lower_item);
  ...
endclass : lower_driver
```

Now create a lower-layer sequence that pulls upper-layer items and translates them to lower-layer items:

```
class higher_to_lower_seq extends ovm_sequence #(lower_item);
  ... // Constructor and OVM automation macros go here.
      // See “Using Sequences” on page 83.
  upper_item u_item;
  lower_item l_item;

  virtual task body();
    forever begin
      `ovm_do_with(l_item,
         { ... }) // Constraints based on u_item
    end
  endtask : body

  // In the pre_do task, pull an upper item from upper sequencer.
  virtual task pre_do(bit is_item);
    p_sequencer.upper_seq_item_port.get_next_item(u_item);
  endtask : pre_do

  // In the post_do task, signal the upper sequencer we are done.
  // And, if desired, update the upper-item properties for the
  // upper-sequencer to use.
  virtual function void post_do(ovm_sequence_item this_item);
    p_sequencer.upper_seq_item_port.item_done(this_item);
  endfunction : post_do
endclass : higher_to_lower_seq
```

The following example illustrates connecting a lower-layer sequencer with an upper-layer sequencer.

Note: The lower-layer sequencer is likely to be encapsulated inside an interface OVC, therefore it will be encapsulated in an env and an agent. This does not change the layering scheme but changes the path to connect the sequencers to each other in the tb file. The connection to the upper sequencer to the lower sequencer will typically happen in the tb env. Where as the connection from lower sequencer to its driver will happen in the `connect()` phase of the agent.

```
// This code resides in an env class.

lower_driver    l_driver0;
lower_sequencer l_sequencer0;
upper_sequencer u_sequencer0;
```

```
function void build();
  // Make lower sequencer execute upper-to-lower translation sequence.
  set_config_string("l_sequencer0", "default_sequence",
     "higher_to_lower_seq");
  // Build the components.
  l_driver0 = lower_driver::type_id::create("l_driver0", this);
  l_sequencer0 = lower_sequencer::type_id::create(("l_sequencer0", this);
  u_sequencer0 = upper_sequencer::type_id::create(("u_sequencer0", this);
endfunction : build

// Connect the components.
function void connect();
  // Connect the upper and lower sequencers.
  l_sequencer0.upper_seq_item_port.connect(u_sequencer0.seq_item_export);
  // Connect the lower sequencer and driver.
  l_driver0.seq_item_port.connect(l_sequencer0.seq_item_export);
endfunction : connect
```

Advanced Generation-Related Aspects of Sequences

This section contains the following subsection:

Randomizing the Kind of Generated Sequences

It is useful in some cases to be able to create a sequence that can randomly select another sequence type and then execute it. The following examples show several ways of achieving this.

The use of `` `ovm_sequence_utils `` registers a sequence type with a particular sequencer's sequence library. The `seq_kind` property is used to identify a specific type in the sequence library based on the sequence type. For example, `get_seq_kind("simple_seq_do")` returns an integer that can be used to identify the sequence type `simple_seq_do`.

Note: The integer value of `seq_kind` for a given sequence type can change from simulation to simulation, therefore you should use the `get_seq_kind()` function to guarantee the correct mapping between the type and the `seq_kind` value.

Example 6-2 Distributed Sequence Generation

The following example executes a sequence ten times. Each time the sequence's type (`seq_kind`) is randomized using a distribution constraint.

```
class distribution_sequence extends ovm_sequence #(bus_transfer);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
```

```
  virtual task body();
    repeat(10) begin
      assert( this.randomize(seq_kind)
        with { seq_kind dist { get_seq_kind("a_seq") := 1,
                           get_seq_kind("b_seq") := 2,
                           get_seq_kind("c_seq") := 5} ; } );
      // Invoke a sequence of the selected seq_kind.
      do_sequence_kind(seq_kind);
    end
  endtask : body
endclass : distribution_sequence
```

Random Selection

The following example shows a sequence that randomly selects from any of the sequence types registered to this sequencer, except the ones you want to avoid. This is a useful approach as it can select from any user-defined sequences you might add in the future. In the code example below, only the sequence type `a_seq` is prevented from being selected.

```
class infinity_minus_sequence extends ovm_sequence #(bus_transfer);
  ...// Constructor and OVM automation macros go here.
     // See "Creating and Adding a New Sequence" on page 84.
  function new(string name="infinity_minus_sequence");
    super.new(name);
  endfunction
  `ovm_sequence_utils(infinity_minus_sequence, xbus_master_sequencer)
  virtual task body();
    // Run any sequence in the sequence library except a_seq.
    for (int i=0; i<p_sequencer.count; i++)
    begin
      assert( this.randomize(seq_kind) with {
      seq_kind != get_seq_kind("ovm_simple_sequence"); } );
      // Invoke a sequence of the selected kind.
      do_sequence_kind(seq_kind);
    end
  endtask : body
endclass
```

Generating the Item or Sequence in Advance

The various `` `ovm_do* `` macros perform several steps sequentially, including the allocation of an object (sequence or sequence item), synchronization with the driver (if needed), randomization, sending to the driver, and so on. The SystemVerilog OVM Class Library provides additional macros that enable finer control of these various steps. This section describes these macros.

`ovm_create

This macro allocates an object using the common factory and initializes its properties. Its argument is a variable of type `ovm_sequence_item` or `ovm_sequence`. You can use the macro with

SystemVerilog's `constraint_mode()` and `rand_mode()` functions to control subsequent randomization of the sequence or sequence item.

In the following example, `my_seq` is similar to previous sequences that have been discussed. The main differences involve the use of the `` `ovm_create(item0) `` call. After the macro call, there are the use of `rand_mode()` and `constraint_mode()` functions and some direct assignments to properties of `item0`. The manipulation of the `item0` object is possible since memory has been allocated for it, but randomization has not yet taken place. Subsequent sections will review the possible options for sending this pre-generated item to the driver.

```
class my_seq extends ovm_sequence #(my_item);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  virtual task body();
    `ovm_create(req)
    req.addr.rand_mode(0); // Disables randomization of addr
    req.dc1.constraint_mode(0); // Disables constraint dc1
    req.addr = 27;
    ...
  endtask : body
endclass: my_seq
```

You can also use a sequence variable as an argument to `` `ovm_create ``.

Note: You might need to disable a constraint to avoid a conflict.

`ovm_send

This macro processes the `ovm_sequence_item` or `ovm_sequence` class handle argument as shown in Figure 4-6 on page 59 and Figure 4-7 on page 60, without any allocation or randomization. Sequence items are placed in the sequencer's queue to await processing while subsequences are processed immediately. The parent `pre_do()`, `mid_do()`, and `post_do()` callbacks still occur as shown.

In the following example, we show the use of `ovm_create()` to pre-allocate a sequence item along with `` `ovm_send ``, which processes it as shown in Figure 4-6 on page 59, without allocation or randomization.

```
class my_seq2 extends ovm_sequence #(my_item);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  virtual task body();
    `ovm_create(req)
    req.addr = 27;
    req.data = 4;
    // No randomization. Use a purely pre-generated item.
    `ovm_send(req)
  endtask : body
endclass: my_seq2
```

Similarly, a sequence variable could be provided to the `` `ovm_create `` and `` `ovm_send `` calls above, in which case the sequence would be processed in the manner shown in Figure 4-7 on page 60, without allocation or randomization.

`ovm_rand_send, `ovm_rand_send_with

These macros are identical to "'ovm_send" on page 130, with the single difference of randomizing the given class handle before processing it. This enables you to adjust an object as required while still using class constraints with late randomization, that is, randomization on the cycle that the driver is requesting the item. `` `ovm_rand_send() `` takes just the object handle. `` `ovm_rand_send_with() `` takes an extra argument, which can be any valid inline constraints to be used for the randomization.

The following example shows the use of `` `ovm_create `` to pre-allocate a sequence item along with the `` `ovm_rand_send* `` macros, which process it as shown in Figure 4-6 on page 59, without allocation. The `rand_mode()` and `constraint_mode()` constructs are used to show fine-grain control on the randomization of an object.

```
class my_seq3 extends ovm_sequence #(my_item);
  ... // Constructor and OVM automation macros go here.
      // See "Creating and Adding a New Sequence" on page 84.
  virtual task body();
    `ovm_create(req)
    req.addr.rand_mode(0);
    req.dc1.constraint_mode(0);
    req.addr = 27;
    // Randomize and process the item.
    `ovm_rand_send(req)

    // Randomize and process again, this time with inline constraints.
    `ovm_rand_send_with(req, {data < 1000;})
  endtask : body
endclass: my_seq3
```

Executing Sequences and Items on other Sequencers

In the preceding sections, all `ovm_do` macros (and their variants) execute the specified item or sequence on the current `p_sequencer`. To allow sequences to execute items or other sequences on specific sequencers, additional macro variants are included that allow specification of the desired sequencer.

'ovm_do_on, 'ovm_do_on_with, 'ovm_do_on_pri, 'ovm_do_on_pri_with

All of these macros are exactly the same as their root versions, except that they all take an additional argument (always the second argument) that is a reference to a specific sequencer.

```
'ovm_do_on(s_seq, that_sequencer);
'ovm_do_on_with(s_seq, that_sequencer, {s_seq.foo == 32'h3;})
```

XBus OVC Example

This chapter introduces the basic architecture of the XBus OVC. It also discusses an executable demo you can run to get hands-on experience in simulation. The XBus source code is provided as a further aid to understanding the OVC architecture. When developing your own simulation environment, you should follow the XBus structure and not its protocol-specific functionality.

All XBus OVC subcomponents inherit from some base class in the SystemVerilog OVM Class Library, so make sure you have the *SystemVerilog OVM Class Reference* available while reading this chapter. It will be important to know, understand, and use the features of these base classes to fully appreciate the rich features you get—with very little added code—right out of the box. If you have not done so, made sure the SystemVerilog OVM Class Library in installed as described in "Installing OVM" on page 9.

You should also familiarize yourself with the XBus specification in the XBus Specification chapter. While not a prerequisite, understanding the XBus protocol will help you distinguish XBus protocol-specific features from OVC protocol-independent architecture.

This chapter contains the following sections:

XBus Demo

The XBus demo constructs an verification environment consisting of a master and a slave. In the default test, the XBus slave communicates using the `slave_memory` sequence. The XBus master `read_modify_write` sequence validates the behavior of the XBus slave memory device.

Instructions for running the XBus example can be found in the readme.txt file in the examples/xbus/examples directory of the OVM kit.

The output from the simulation below shows the XBus testbench topology containing an environment.The environment contains one active master and one active slave agent.

The test runs the `read_modify_write` sequence, which activates the read byte sequence followed by the write byte sequence, followed by another read byte sequence. An assertion verifies that the data read in the second read byte sequence is identical to the data written in the write byte sequence. The following output is generated when the test is simulated with `OVM_VERBOSITY = OVM_LOW`.

```
[0] hier=__global__: Running test test_read_modify_write...
[0] hier=ovm_test_top: Printing the test topology:
----------------------------------------------------------------------
Name                     Type                Size                Value
----------------------------------------------------------------------
ovm_test_top             test_read_modify_w+ -                    @719
  xbus_demo_tb0          xbus_demo_tb        -                    @716
    scoreboard0          xbus_demo_scoreboa+ -                    @710
      item_collected_ex+ ovm_connector_base  -                   @1068
      disable_scoreboard integral            1                     'h0
      num_writes         integral            32                    'd0
      num_init_reads     integral            32                    'd0
      num_uninit_reads   integral            32                    'd0
      recording_detail   ovm_verbosity       32               OVM_FULL
    xbus0                xbus_env            -                    @713
      bus_monitor        xbus_bus_monitor    -                    @874
      masters[0]         xbus_master_agent   -                    @998
      slaves[0]          xbus_slave_agent    -                    @939
      has_bus_monitor    integral            1                     'h1
      num_masters        integral            32                    'h1
      num_slaves         integral            32                    'h1
      intf_checks_enable integral            1                     'h1
      intf_coverage_ena+ integral            1                     'h1
      recording_detail   ovm_verbosity       32               OVM_FULL
```

```
   recording_detail     ovm_verbosity       32                OVM_FULL
----------------------------------------------------------------------
[190] hier=ovm_test_top.xbus_demo_tb0.scoreboard0: READ to empty
address...Updating address : 7f4e with data : 15
[190] hier=ovm_test_top.xbus_demo_tb0.xbus0.bus_monitor: Transfer collected :
----------------------------------------------------------------------
Name                      Type                Size                Value
----------------------------------------------------------------------
xbus_transfer_inst        xbus_transfer       -                   @1394
  addr                    integral            16                'h7f4e
  read_write              xbus_read_write_en+ 32                  READ
  size                    integral            32                   'h1
  data                    da(integral)        1                      -
    [0]                   integral            8                   'h15
  wait_state              da(integral)        0                      -
  error_pos               integral            32                   'h0
  transmit_delay          integral            32                   'h0
  master                  string              10            masters[0]
  slave                   string              9              slaves[0]
  begin_time              time                64                   150
  end_time                time                64                   190
----------------------------------------------------------------------

[320] hier=ovm_test_top.xbus_demo_tb0.scoreboard0: WRITE to existing
address...Updating address : 7f4e with data : 16
[320] hier=ovm_test_top.xbus_demo_tb0.xbus0.bus_monitor: Transfer collected :
----------------------------------------------------------------------
Name                      Type                Size                Value
----------------------------------------------------------------------
xbus_transfer_inst        xbus_transfer       -                   @1394
  addr                    integral            16                'h7f4e
  read_write              xbus_read_write_en+ 32                 WRITE
  size                    integral            32                   'h1
  data                    da(integral)        1                      -
    [0]                   integral            8                   'h16
  wait_state              da(integral)        0                      -
  error_pos               integral            32                   'h0
  transmit_delay          integral            32                   'h0
  master                  string              10            masters[0]
  slave                   string              9              slaves[0]
  begin_time              time                64                   300
```

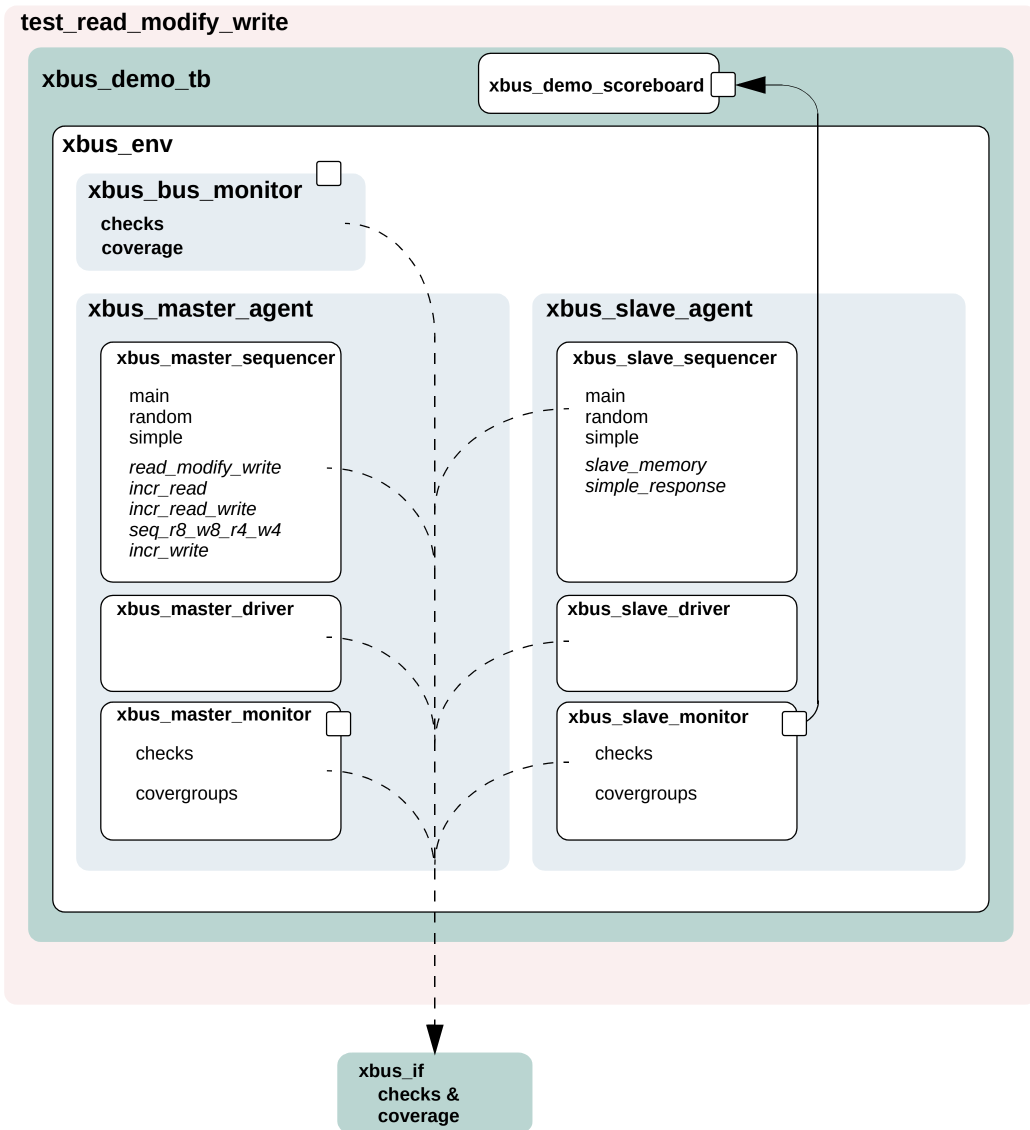
```
  end_time              time                 64                    320
----------------------------------------------------------------------

[410] hier=ovm_test_top.xbus_demo_tb0.scoreboard0: READ to existing
address...Checking address : 7f4e with data : 16
[410] hier=ovm_test_top.xbus_demo_tb0.xbus0.bus_monitor: Transfer collected :
----------------------------------------------------------------------
Name                    Type                 Size                Value
----------------------------------------------------------------------
xbus_transfer_inst      xbus_transfer        -                   @1394
  addr                  integral             16                'h7f4e
  read_write            xbus_read_write_en+  32                   READ
  size                  integral             32                    'h1
  data                  da(integral)         1                       -
    [0]                 integral             8                    'h16
  wait_state            da(integral)         0                       -
  error_pos             integral             32                    'h0
  transmit_delay        integral             32                    'h0
  master                string               10             masters[0]
  slave                 string               9               slaves[0]
  begin_time            time                 64                    370
  end_time              time                 64                    410
----------------------------------------------------------------------

[2000] hier=ovm_test_top: Calling global_stop_request() to end the run phase
[2000] hier=ovm_test_top.xbus_demo_tb0.scoreboard0: Reporting scoreboard
information...
----------------------------------------------------------------------
Name                    Type                 Size                Value
----------------------------------------------------------------------
scoreboard0             xbus_demo_scoreboa+  -                    @710
  item_collected_export ovm_connector_base   -                   @1068
    recording_detail    ovm_verbosity        32               OVM_FULL
  disable_scoreboard    integral             1                     'h0
  num_writes            integral             32                    'd1
  num_init_reads        integral             32                    'd1
  num_uninit_reads      integral             32                    'd1
  recording_detail      ovm_verbosity        32               OVM_FULL
----------------------------------------------------------------------

--- OVM Report Summary ---
```

```
** Report counts by severity
OVM_INFO :   10
OVM_WARNING :    0
OVM_ERROR :    0
OVM_FATAL :    0
** Report counts by id
[DEBUG]     9
[RNTST]     1
Simulation complete via $finish(1) at time 2 US + 7
```

XBus Demo Architecture

Figure 7-1 on page 138 shows the testbench topology of the XBus simulation environment in the XBus demo example delivered with this release.

Figure 7-1 XBus Demo Architecture



XBus Top Module

The XBus testbench is instantiated in a top-level module to create a class-based simulation environment. The example below uses an example DUT with XBus-specific content. The example is trivial intentionally so that the focus is on the XBus OVC environment.

The top module contains the typical HDL constructs and a SystemVerilog interface. This interface is used to connect the class-based testbench to the DUT. The XBus environment inside the testbench uses a virtual interface variable to refer to the SystemVerilog interface. The following example shows the XBus interface (`xi0`) and the example DUT connected together. The `run_test()` command used to simulate the DUT and the testbench is covered in the next section.

Example 7-1 xbus_tb_top.sv

```
module xbus_tb_top;

  `include "xbus.svh"
  `include "test_lib.sv"

  xbus_if xi0(); // SystemVerilog interface to the DUT

  dut_dummy dut(
    xi0.sig_request[0],
    ...
    xi0.sig_error
  );

  initial begin
    run_test();
  end

  initial begin
    xi0.sig_reset <= 1'b1;
    xi0.sig_clock <= 1'b1;
    #51 xi0.sig_reset = 1'b0;
  end

  //Generate clock.
  always
    #5 xi0.sig_clock = ~xi0.sig_clock;

endmodule
```

The XBus SystemVerilog interface is instantiated in the top-level testbench module. The interface uses generally-accepted naming conventions for its signals to allow easy mapping to any naming conventions employed by other implementations of the XBus protocol. The DUT pins connect directly to the signal inside the interface instance. Currently, the signals are simple non-directional variables that are driven either by the DUT or the class-based testbench environment via a virtual interface. The XBus interface contains concurrent assertions to perform physical checks. Refer to “Checking for DUT Correctness” on page 95 and “XBus Interface” on page 151 for more information.

The Test

In OVM, the test is defined in a separate class, `test_read_modify_write`. It derives from `xbus_demo_base_test` that, in turn, derives from `ovm_test`. The `xbus_demo_base_test` test builds the `xbus_demo_tb` object and manages the `run()` phase of the test. Subsequent derived tests, such as `test_read_modify_write`, can leverage this functionality as shown in the example below.

All classes that use the `` `ovm_component_utils `` macros are registered with a common factory, `ovm_factory`. When the top module calls `run_test(`*`test_name`*`)`, the factory is called upon to create an instance of a test with type *`test_name`* and then simulation is started. When `run_test` is called without an argument, a `+OVM_TESTNAME=`*`test_name`* command-line option is checked and, if it exists, the test with that type name is created and executed. If neither are found, all constructed components will be cycled through their simulation phases. Refer to "Creating and Selecting a User-Defined Test" on page 78 for more information.

Example 7-2 test_lib.sv

```
`include "xbus_demo_tb.sv"

class xbus_demo_base_test extends ovm_test;

  `ovm_component_utils(xbus_demo_base_test)

  xbus_demo_tb xbus_demo_tb0; // XBus verification environment
  ovm_table_printer printer;

  function new(string name = "xbus_demo_base_test",
    ovm_component parent=null);
    super.new(name, parent);
  endfunction
  // OVM build() phase
  virtual function void build();
    super.build();
    // Enable transaction recording.
    set_config_int("*", "recording_detail", OVM_FULL);
    // Create the testbench.
    xbus_demo_tb0 = xbus_demo_tb::type_id::create("xbus_demo_tb0", this);
    // Create a specific-depth printer for printing the topology.
    printer = new();
    printer.knobs.depth = 3;
  endfunction
   // Built-in OVM phase
  function void end_of_elaboration();
    // Set verbosity for the bus monitor.
    xbus_demo_tb0.xbus0.bus_monitor.set_report_verbosity_level(OVM_FULL);
    // Print the test topology.
    this.print(printer);
  endfunction : end_of_elaboration();
  // OVM run() phase
  task run();
    #2000;
    // Call global_stop_request() to end the run phase.
    global_stop_request();
  endtask
endclass
```

Line 1 Include the necessary file for the test. The testbench used in this example is the `xbus_demo_tb` that contains, by default, the bus monitor, one master, and one slave. See "Testbench Environment" on page 142.

Lines 3-5 All tests should derive from the `ovm_test` class and use the `` `ovm_component_utils `` or the `` `ovm_component_utils_begin ``/`` `ovm_component_utils_end `` macros. See the *SystemVerilog OVM Class Reference* for more information.

Line 7 Declare the testbench. It will be constructed by the `build()` function of the test.

Line 8 Declare a printer of type `ovm_table_printer`, which will be used later to print the topology. This is an optional feature. It is helpful in viewing the relationship of your topology defined in the configuration and the physical testbench created for simulation. Refer to the *SystemVerilog OVM Class Reference* for different types of printers available.

Lines 15-24 Specify the `build()` function for the base test. As required, `build` first calls the `super.build()` function in order to update any overridden fields. Then the `xbus_demo_tb` is created using the `create()` function. The `build()` function of the `xbus_demo_tb` is executed by the OVM library phasing mechanism during `build()`. The user is not required to explicitly call `xbus_demo_tb0.build()`.

Lines 26-31 Specify the `end_of_elaboration()` function for the base test. This function is called after all the component's `build()` and `connect()` phases are executed. At this point, the test can assume that the complete testbench hierarchy is created and all testbench connections are made. The test topology is printed.

Lines 33-37 Specify the `run()` task for the base test. In this case, the simulation is run for `2000` time units. Finally, the simulation is stopped using the `global_stop_request()` function.

Now that the base test is defined, a derived test will be examined. The following code is a continuation of the `test_lib.sv` file.

```
class test_read_modify_write extends xbus_demo_base_test;
  `ovm_component_utils(test_read_modify_write)
  function new(string name = "test_read_modify_write",
     ovm_component parent=null);
    super.new(name,parent);
  endfunction

  virtual function void build();
    // Set the default sequence for the master and slave.
    set_config_string("xbus_demo_tb0.xbus0.masters[0].sequencer",
    "default_sequence", "read_modify_write_seq");
    set_config_string("xbus_demo_tb0.xbus0.slaves[0].sequencer",
    "default_sequence", "slave_memory_seq");
    // Create the testbench.
    super.build();
  endfunction

endclass
```

The `build()` function of the derivative test, `test_read_modify_write`, is of interest. The `build()` function registers an override of `read_modify_write_seq` to the master agent's sequence sequencer and also an override of `slave_memory_seq` to the slave agent's sequence sequencer. Once these overrides are executed, `super.build()` is called which creates the `xbus_demo_tb0` as specified in the `xbus_demo_base_test` build function.

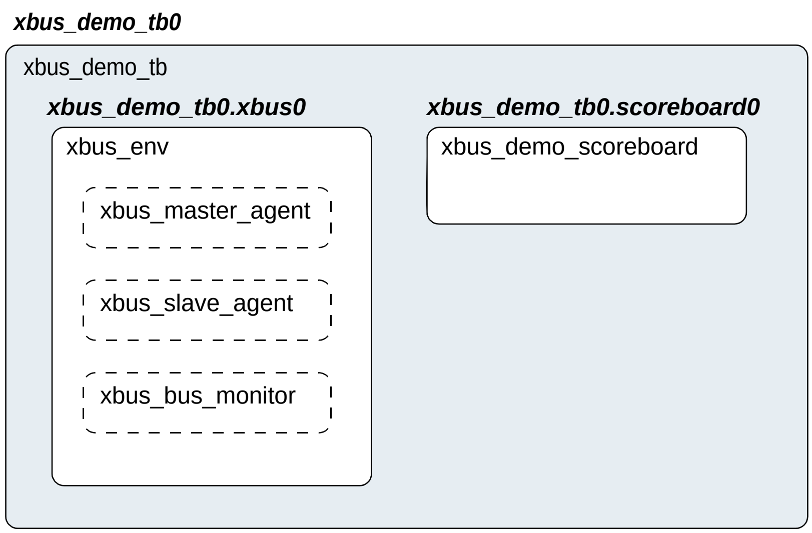
The `run()` task implementation is inherited by `test_read_modify_write` since this test derives from the `xbus_demo_base_test`. Since that implementation is sufficient for this test, no action is required by you. This greatly simplifies this test.

Testbench Environment

This section discusses the testbench created in "test_lib.sv" on page 140. The code that creates the `xbus_demo_tb` is repeated here.

```
xbus_demo_tb0 = xbus_demo_tb::type_id::create("xbus_demo_tb0", this);
```

Figure 7-2 Testbench Derived from ovm_env



In general, testbenches can contain any number of envs (OVCs) of any type: xbus, pci, ahb, ethernet, and so on. The XBus demo creates a simple testbench consisting of a single XBus environment (OVC) with one master agent, one slave agent, and one bus monitor (see Figure 7-2 on page 142). The following code defines a class that specifies this configuration. The test will create an instance of this class.

Example 7-3 xbus_demo_tb.sv

```
function void xbus_demo_tb::build();
```

```
2    super.build();
3    set_config_int("xbus0", "num_masters", 1);
4    set_config_int("xbus0", "num_slaves", 1);
5    xbus0 = xbus_env::type_id::create("xbus0", this);
6    scoreboard0 = xbus_demo_scoreboard::type_id::create("scoreboard0", this);
7  endfunction : build
8
9 function void xbus_demo_tb::connect();
10   // Connect the slave0 monitor to scoreboard.
11   xbus0.slaves[0].monitor.item_collected_port.connect(
12      scoreboard0.item_collected_export);
13   // Assign interface for xbus0.
14   xbus0.assign_vi(xbus_tb_top.xi0);
15 endfunction : connect
16
17 function void end_of_elaboration();
18   // Set up slave address map for xbus0 (basic default).
19   xbus0.set_slave_address_map("slaves[0]", 0, 16'hffff);
20 endfunction : end_of_elaborationct
```

Line 1 Declare the `build()` function.

Line 2 Call `super.build()` in order to update any overridden fields. This is important because the test, which creates the testbench, may register overrides for the testbench. Calling `super.build()` will ensure that those overrides are updated.

Lines 3-4 The `set_config_int` calls are adjusting the `num_masters` and `num_slaves` configuration fields of the `xbus_env`. In this case, the `xbus0` instance of the `xbus_env` is being manipulated. Line 3 instructs the `xbus0` instance of the `xbus_env` to contain one master agent. The `num_masters` property of the `xbus_env` specifies how many master agents should be created. The same is done for `num_slaves`.

Line 5 Create the `xbus_env` instance named `xbus0`. The `create()` call specifies that an object of type `xbus_env` should be created with the instance name `xbus0`.

Line 6 As with `xbus0`, the scoreboard is created.

Line 9 Declare the `connect()` function.

Lines 10-14 Make the connections necessary for the `xbus0` environment and the `scoreboard0`. Two connections are made:

- The TLM connection between the analysis port on the `xbus0.slaves[0].monitor` and the analysis export on the `scoreboard0` instance.
- The assignment, or "connection", to the SystemVerilog interface instantiated in the XBus top module. This assignment will allow the testbench to communicate with the DUT.

Line 17 Declare the `end_of_elaboration()` built-in OVM phase.

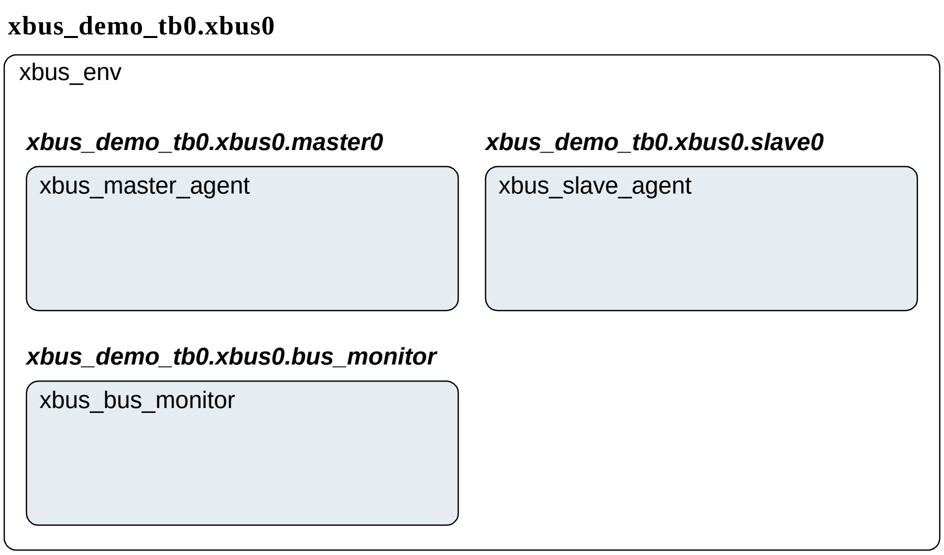
Line 19 Assign the slave address map for the `slaves[0]`. This can be done once the `build()` and `connect()` functions are complete since the `end_of_elaboration()` function expects the complete testbench to be created and connected.

XBus Environment

The `xbus_env` component contains any number of XBus master and slave agents. In this demo, the `xbus_env` (shown in Figure 7-3 on page 144) is configured to contain just one master and one slave agent.

Note: The bus monitor is created by default.

Figure 7-3 Instance of xbus_env



The `build()` function of the `xbus_env` creates the master agents, slave agents, and the bus monitor. Three properties control whether these are created. The source code is shown here.

```
function void xbus_env::build();
  string inst_name;
  super.build();
  if(has_bus_monitor == 1) begin
    bus_monitor = xbus_bus_monitor::type_id::create("bus_monitor", this);
  end
  masters = new[num_masters];
  for(int i = 0; i < num_masters; i++) begin
    $sformat(inst_name, "masters[%0d]", i);
    masters[i] = xbus_master_agent::type_id::create(inst_name, this);
    set_config_int({inst_name, "*"}, "master_id", i);
```

```
12   end
13   slaves = new[num_slaves];
14   for(int i = 0; i < num_slaves; i++) begin
15     $sformat(inst_name, "slaves[%0d]", i);
16     slaves[i] = xbus_slave_agent::type_id::create("xbus_slave_agent",
17        this);
18        inst_name));
19   end
20 endfunction: build
```

Line 1 Declare the `build()` function.

Line 3 Call `super.build()`. This guarantees that the configuration fields (`num_masters`, `num_slaves`, and `has_bus_monitor`) are updated per any overrides.

Lines 4-6 Create the bus monitor if the `has_bus_monitor` control field is set to 1. The `create` function is used for creation.

Lines 7-12 The master's dynamic array is sized per the `num_masters` control field. This allows the for loop to populate the dynamic array according to the `num_masters` control field. The instance name that is used for the master agent instance is built using `$sformat` so that the instance names match the dynamic-array identifiers exactly. The iterator of the for loop is also used to register a configuration override targeted at the `master_id` properties of the master agent and all its children (through the use of the asterisk). This defines which request-grant pair is driven by the master agent.

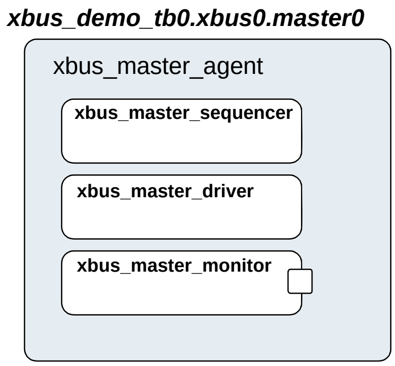
Lines 13-19 As in the master-agent creation code above, this code creates the slave agents using `num_slaves` and does not require the configuration override.

XBus Agent

The `xbus_master_agent` (shown in Figure 7-4 on page 146) and `xbus_slave_agent` are structured identically; the only difference is the protocol-specific function of its subcomponents.

The XBus master agent contains up to three subcomponents: the sequencer, driver, and monitor. By default, all three are created. However, the configuration can specify the agent as passive (`is_active=OVM_PASSIVE`), which disables the creation of the sequencer and driver. The `xbus_master_agent` is derived from `ovm_agent`.

Figure 7-4 Instance of xbus_master_agent



The `build()` function of the `xbus_master_agent` is specified to create the driver, sequencer, and the monitor. The `is_active` property controls whether the driver and sequencer are created.

```
1  function void xbus_master_agent::build();
2    super.build();
3    monitor = xbus_master_monitor::type_id::create("monitor", this);
4    if (is_active == OVM_ACTIVE) begin
5      sequencer = xbus_master_sequencer::type_id::create("sequencer", this);
6      driver = xbus_master_driver::type_id::create("driver", this);
7    end
8  endfunction : build
9
10 function void xbus_master_agent::connect();
11   if (is_active == OVM_ACTIVE) begin
12     driver.seq_item_port.connect(sequencer0.seq_item_export);
13   end
14 endfunction
```

Line 1 Declare the `build()` function.

Line 2 Call `super.build()`. This guarantees that the configuration field (`is_active`) is updated per any overrides.

Line 3 Create the monitor. The monitor is always created. Creation is not conditional on a control field.

Lines 4-7 Create the sequencer and driver if the `is_active` control field is set to `OVM_ACTIVE`. The `create_component` function is used for creation.

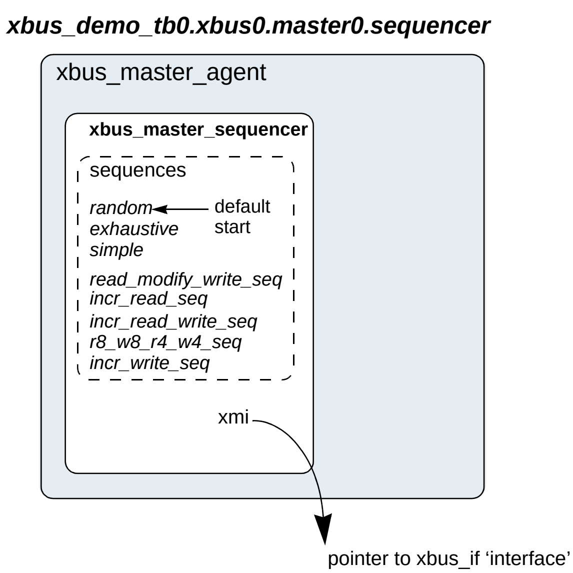
Line 10 Declare the `connect()` function.

Lines 11-13 Since the driver expects transactions from the sequencer, the interfaces in both components should be connected using the `connect()` function. The agent (which creates the monitor, sequencer, and driver) is responsible for connecting the interfaces of its children.

XBus Sequencer

This component controls the flow of sequence items to the driver (see Figure 7-5 on page 147). The sequencer controls which sequence items are provided to the driver. The `ovm_sequencer` base class includes three built-in sequences: `ovm_random_sequence`, `ovm_exhaustive_sequence`, and `ovm_simple_sequence`. Refer to "Predefined Sequences" on page 61 for more information. The `default_sequence` property selects the sequence to start. By default, a sequence of type `ovm_random_sequence` is started.

Figure 7-5 Instance of xbus_master_sequencer



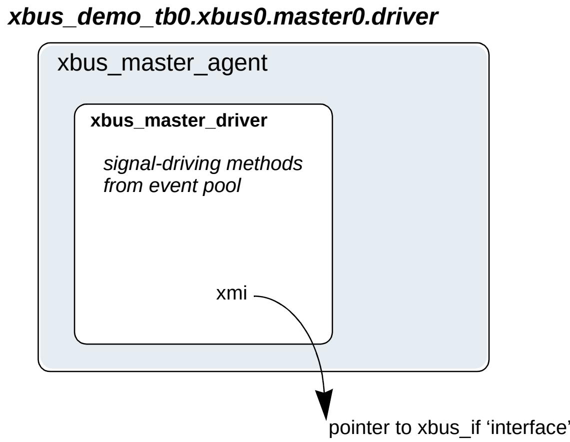
A user-defined sequencer provides an optional virtual interface to enable sequences to synchronize with the protocol's physical signals. The `xmi` variable is a simple SystemVerilog virtual interface reference which is assigned to the physical SystemVerilog interface. After the XBus environment is built, the `xmi` variable is still undefined. You must set this variable before starting simulation using direct assignment or the `assign_vi()` convenience method provided by the IP developer. The XBus example provides a function called `assign_vi()` in the environment that assigns the virtual interfaces of the agent's children. This use can be seen in the XBus demo database.

The sequencer's constructor begins with the required `super.new()` call, followed by a `` `ovm_update_seqeunce_lib_and_item `` macro. This macro expands to a function call that copies all of the statically-registered sequences into the sequencer's local sequence library, which contains all of the sequences available for execution by this sequencer. You can easily create sequences that randomly select from among the other available sequences and scenarios.

XBus Driver

This component drives the XBus bus-signals interface by way of the `xmi` virtual interface property (see Figure 7-6 on page 148 below). The `xbus_master_driver` fetches `xbus_transfer` transactions from the sequencer and processes them based on the physical-protocol definition. In the XBus example, the `seq_item_port` methods `get_next_item()` and `item_done()` are accessed to retrieve transactions from the sequencer.

Figure 7-6 Instance of xbus_master_driver



The primary role of the driver is to drive (in a master) or respond (in a slave) on the XBus bus according to the signal-level protocol. This is done in the `run()` task that is automatically invoked as part of OVM's built-in simulation phasing (discussed in “Simulation Phase Methods” on page 104). For the master driver, the core routine is summarized as follows:

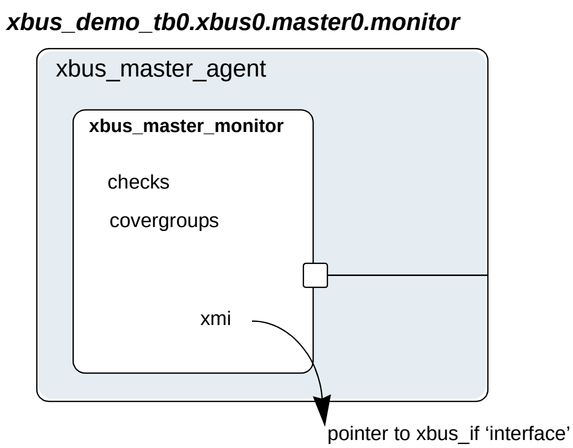
```
task xbus_master_driver::run();
  ...
  @(negedge xmi.sig_reset);
  forever begin // Repeat the following forever.
    @(posedge xmi.sig_clock);
    seq_item_port.get_next_item(item); // Pull item from sequencer.
    ...
    drive_transfer(item); // Drive item onto signal-level bus.
    ...
    seq_item_port.item_done(); // Indicate we are done.
  end
endtask
```

Once the `sig_reset` signal is deasserted, the driver's `run` task runs forever until stopped by way of the `global_stop_request()` task. You are encouraged to study the XBus driver source code to gain a deeper understanding of the implementation specific to the XBus protocol.

XBus Agent Monitor

The XBus monitor collects `xbus_transfers` seen on the XBus signal-level interface (see Figure 7-7 on page 149). If the checks and coverage are present, those corresponding functions are performed as well.

Figure 7-7 Instance of xbus_master_monitor



The primary role of the XBus master monitor is to sample the activity on the XBus master interface and collect the `xbus_transfer` transactions that pertain to its parent master agent only. The transactions that are collected are provided to the external world by way of a TLM analysis port. The monitor performs this duty in the run task that is automatically invoked as part of simulation phasing. The run task may fork other processes and call other functions or tasks in performance of its duties. The exact implementation is protocol- and programmer-dependent, but the entry point, the run task, is the same for all components. Refer to "Simulation Phase Methods" on page 104 for more information about simulation phases.

The monitor's functionality is contained in an infinite loop defined with the `run()` task. The `global_stop_request()` will cause the `run()` tasks to finish by default, allowing other simulation phases to complete and allow the simulation to end.

The checks are responsible for enforcing protocol-specific checks, and the coverage is responsible for collecting functional coverage from the collected `xbus_transfers`.

XBus Bus Monitor

The XBus bus monitor collects `xbus_transfers` seen on the XBus signal-level interface and emits status updates via a state transaction, indicating different activity on the bus. The XBus bus monitor has class checks and collects coverage if checks and coverage collection is enabled. The XBus bus monitor is instantiated within an the XBus environment.

The `xbus_env` `build()` function has a control field called `has_bus_monitor`, which determines whether the `xbus_bus_monitor` is created or not. The bus monitor will be created by default since the default value for this control field is one. You can use the `set_config_int` interface to override this value.

```
set_config_int("xbus0", "has_bus_monitor", 0);
```

Here, the `xbus0` instance of `xbus_env` has its `has_bus_monitor` control field overridden to 0. Therefore, the `xbus_bus_monitor` in `xbus0` will not be present. The `build()` function for the `xbus_env` that uses the `has_bus_monitor` control field can be found in "XBus Environment" on page 144.

Collecting Transfers from the Bus

The XBus bus monitor populates the fields of `xbus_transfer` including the master and slave, which indicate which master and slave are performing a transfer on the bus. These fields are required to ensure that a slave responds to the appropriate address range when initiated by a master.

In the XBus protocol, each master on the bus has a dedicated request signal and a dedicated grant signal defined by the master agent's ID. To determine which master is performing a transfer on the bus, the XBus bus monitor checks which grant line is asserted.

To keep the XBus bus monitor example simple, an assumption has been made that the *n*th master connects to the *n*th request and grant lines. For example, `master[0]` is connected to `grant0`, `master[1]` is connected to `grant1`, and so on. Therefore, when the XBus bus monitor sees that `grant0` is asserted, it assumes that `master[0]` is performing the transfer on the bus.

To determine which slave should respond to the transfer on the bus, the XBus bus monitor needs to know the address range supported by each slave in the environment. The environment developer has created the user interface API, `xbus_env::set_slave_address_map()`, to set the address map for the slave as well as the bus monitor. The prototype for this function is

```
set_slave_address_map(string slave_name, int min_addr, int max_addr);
```

For each slave, call `set_slave_address_map()` with the minimum and maximum address values that the slave should respond to. This function sets the address map for the slave and provides information to the bus monitor about each slave and its address map.

Using the address map information for each slave and the address that is collected from the bus, the bus monitor determines which slave has responded to the transfer.

Number of Transfers

The bus monitor has a protected field property, `num_transactions`, which holds the number of transfers that were monitored on the bus.

Notifiers Emitted by the XBus Bus Monitor

The XBus bus monitor contains two analysis ports, which provide information on the different types of activity occurring on the XBus signal-level interface. They are:

- `state_port`—This port provides an `xbus_status` object which contains an enumerated `bus_state` property. The `bus_state` property reflects bus-state changes. For example, when the bus enters reset, the `bus_state` property is set to `RST_START` and the `xbus_status` object is written to the analysis port.
- `item_collected_port`—This port provides the XBus transfer that is collected from the signal interface after a transfer is complete. This collected transfer is written to the `item_collected_port` analysis port.

Note: Any component provided by the appropriate TLM interfaces can attach to these TLM ports and listen to the information provided.

Checks and Coverage

The XBus bus monitor performs protocol-specific checks using class checks and collects functional coverage from the collected `xbus_transfers`.

The OVM field `coverage_enable` and `checks_enable` are used to control whether coverage and checks, respectively, will be performed or not. Refer to "Implementing a Coverage Model" on page 100 for more information.

XBus Interface

The XBus interface is a named bundle of nets and variables such that the master agents, slave agents, and bus monitor can drive or monitor the signals in it. Any physical checks to be performed are placed in the interface. Refer to "Implementing a Coverage Model" on page 100.

Assertions are added to perform physical checks.The `xbus_env` field `intf_checks_enable` controls whether these checks are performed. Refer to "Implementing a Coverage Model" on page 100 for more information.

The code below is an example of a physical check for the XBus interface, which checks that a valid address is driven during the normal address phase. A concurrent assertion is added to the interface to perform the check and is labeled `assertAddrUnknown`. This assertion evaluates on every positive edge of `sig_clock` if `checks_enable` is true. The `checks_enable` bit is controlled by the `intf_checks_enable` field. If any bit of the address is found to be at an unknown value during the normal address phase, an error message is issued.

```
always @(posedge sig_clock)
  begin
    assertAddrUnknown:assert property (
    disable iff(!checks_enable)
      (sig_grant |-> ! $isunknown(sig_addr)))
    else
      $error("ERR_ADDR_XZ\n Address went to X or Z during Address Phase");
  end
```

XBus Specification

Introduction

Motivation

The motivation for the XBus specification is to provide an example of a simple bus standard for demonstration purposes and to illustrate the methodology required for a bus-based OVC. As such, the XBus specification is designed to demonstrate all of the important features of a typical modern bus standard while keeping complexity to a minimum.

Bus Overview

The XBus is a simple non-multiplexed, synchronous bus with no pipelining (so as to ensure simple drivers). The address bus is 16 bits wide and the data bus is byte-wide (so as to avoid alignment issues). Simple burst transfers are allowed and slaves are able to throttle data rates by inserting wait states.

The bus can have any number of masters and slaves (the number of masters is only limited by the arbitration implementation). Masters and slaves are collectively known as “bus agents”.

The transfer of data is split into three phases: Arbitration Phase, Address Phase and Data Phase. Because no pipelining is allowed, these phases happen sequentially for each burst of data. The Arbitration and Address Phases each take exactly one clock cycle. The Data Phase may take one or more clock cycles.

Bus Description

Bus Signals

The list of bus signals (not including arbitration signals) is shown in Table 8-1 on page 154. All control signals are active high.

Table 8-1 Bus Signals

Signal Name	Width (bits)	Driven By	Purpose
clock	1	n/a	Master clock for bus
reset	1	n/a	Bus reset
start	1	arbiter	This signal is high during the Arbitration Phase and low during the Address and Data Phases.
addr	16	master	Address of first byte of a transfer
size	2	master	Indicates how many bytes will be transfers: n 00 => 1 byte n 01 => 2 bytes n 10 => 4 bytes n 11 => 8 bytes
read	1	master	This signal is high for read transfers (write must be low).
write	1	master	This signal is high for write transfers (read must be low).
bip	1	master	Burst In Progress—driven high by master during Data Phase for all bytes except the last byte of the burst. This signal, when combined with wait and error, can be used by the arbiter to determine if the bus will start a new transfer in the next clock cycle.
data	8	master/slave	Data for reads and writes
wait	1	slave	High if slave needs master to wait for completion of transfer
error	1	slave	High if slave error condition applies to this transfer

Clocking

All bus agents operate synchronous to the rising edge of the *clock* signal with the exception of *gnt* signals (see "Arbitration Phase" on page 155).

Reset

The active high *reset* signal is synchronous to the rising edge of clock. *reset* shall be asserted during power up and shall remain asserted for a minimum of five rising edges of clock* after power and clock have stabilized. Thereafter, *reset* shall be de-asserted synchronous to a rising edge of clock.

reset may be asserted at any time during operation. In such cases, *reset* must be asserted for at least three clock cycles and must be both asserted and de-asserted synchronous to the rising edge of clock. The assertion of *reset* cancels any pending transfer at the first rising edge of clock where *reset* is asserted. Any bytes that have been transferred prior to assertion of *reset* are considered to have succeeded. Any byte that would have succeeded at the rising edge of clock where *reset* is first asserted is considered to have failed.

While *reset* is asserted, all agents should ignore all bus and arbitration signals. While reset is asserted, the arbiter should drive *start* and all *gnt* signals low. At the first rising edge of clock where reset is de-asserted, the arbiter should drive *start* high. Thereafter, the normal bus operation should occur.

Arbitration Phase

Each XBus shall have a single, central arbiter to perform arbitration and certain other central control functions.

The Arbitration Phase always lasts for one clock cycle. During the Arbitration Phase, the arbiter shall drive the *start* signal high. At all other times, the arbiter should drive the *start* signal low. The *start* signal can therefore be used by slaves to synchronize themselves with the start of each transfer. The arbiter shall always drive *start* high in the cycle following the last cycle of each Data Phase or in the cycle following a NOP Address Phase. The last cycle of a Data Phase is defined as a Data Phase cycle in which either the *error* signal is high, or both the *bip* and *wait* signals are low.

Each master on the bus has a dedicated *req* signal and a dedicated *gnt* signal. The arbiter samples all *req* signals at each falling edge of clock where *start* is asserted and asserts a single *gnt* signal based on an unspecified priority system. At all falling edges of clock where *start* is not asserted, the arbiter shall drive all *gnt* signals low. Thus a master can see assertion of its *gnt* signal not only as an indication that it has been granted the bus, but also as an indication that it must start an Address Phase. It is not necessary for the master to check the *start* signal before starting its Address Phase.

Once a master is granted the bus, it must drive a transaction onto the bus immediately. No other master is allowed to drive the bus until the current master has completed its transaction.

Note: Only the arbiter is allowed to drive a NOP transfer. This means that a master must drive a real transfer if it is granted the bus. Therefore, masters should not request the bus unless they can guarantee they will be ready to do a real transfer.

Arbitration signals shall be active high and shall be named according to a convention whereby the first part of the name is the root signal name ("*req_*" for the request signal; "*gnt_*" for the grant signal) and the second part of the name is the logical name or number of the master. Although the arbitration signals form part of the XBus specification, they are not considered to be "bus" signals as they are not connected to all agents on the bus.

It is up to individual implementations to decide upon an appropriate arbitration system. Arbiters might allocate different priorities to each master or might choose randomly with each master having equal priority.

Address Phase

The Address Phase lasts for a single clock cycle and always immediately follows the Arbitration Phase.

NOP Cycle

Where no master has requested the bus and the *start* signal is asserted at the falling edge of clock, no *gnt* signal is asserted at the start of the Address Phase and the arbiter itself is responsible for driving the bus to a "no operation" (NOP) state. It does this by driving the *addr* and *size* signals to all zeroes and both the *read* and *write* signals low. A NOP address phase has no associated data phase so the arbiter shall assert the *start* signal in the following clock cycle.

Note: This means that the arbiter is connected to certain bus signals in addition to the arbitration signals and behaves as a "default master".

Normal Address Phase

If, at the rising edge of clock, a master sees its *gnt* signal asserted, then it must drive a valid Address Phase in the following cycle. The master should also de-assert its *req* signal at this clock edge unless it has a further transfer pending.

During the Address Phase, the granted master should drive the *addr* and *size* signals to valid values and should drive either *read* or *write* (but not both) high.The address driven on *addr* represents the address of the first byte of a burst transfer. It is up to the slave to generate subsequent addresses during burst transfers.

The master shall only drive the *addr, size, read* and *write* signals during the Address Phase. During the subsequent Data Phase, the master should not drive these signals.

Data Phase

The Data Phase may last for one or more clock cycles. The Data Phase follows immediately after the Address Phase (and is immediately followed by the Arbitration Phase).

Write Transfer

On clock cycle after driving a write Address Phase, the master shall drive the first byte of data onto the bus. If at the end of this clock cycle, the slave has asserted the *wait* signal, then the master shall continue to drive the same data byte for a further clock cycle. The *data* signal may only change at the end of a cycle where *wait* is not asserted. Thus, the slave can insert as many wait states as it requires. The master shall drive the *bip* signal high throughout the Data Phase until the point at which the final byte of the transfer is driven onto the bus, at which point it shall be driven low.

At the end of the transfer (the end of the cycle where both *bip* and *wait* are low) the master shall cease to drive all bus signals.

Error during Write Transfer

The slave shall drive the *error* throughout the Data Phase. If a slave encounters an error condition at any point during the Data Phase of a write transfer, it may signal this by asserting the *error* signal. To signal an error condition, the slave must drive the *error* signal high while driving the *wait* signal low. This indicates to the master that the associated byte of the transfer failed—any previous bytes in the burst are considered to have succeeded; any subsequent bytes in the burst are abandoned. The assertion of *error* always terminates the Data Phase even if *bip* is asserted simultaneously.

Read Transfer

On the clock cycle after the master drives a read Address Phase, the slave can take one of two actions. It can either drive the first byte of data onto the bus while driving the *wait* signal low or it can drive the *wait* signal high to indicate that it is not yet ready to drive data. Each byte of data is latched only by the master at the end of a cycle where *wait* is low—thus the slave can insert as many wait states as is required. The master shall drive the *bip* signal high throughout the Data Phase until the point at which the master is ready to receive the final byte of the transfer, at which point it shall be driven low.

At the end of the transfer (the end of the cycle where both *bip* and *wait* are low) the master shall cease to drive all bus signals.

Error during Read Transfer

The slave shall drive the *error* throughout the Data Phase. If a slave encounters an error condition at any point during a read transfer, it may signal this by asserting the *error* signal. To signal an error condition, the slave must drive the *error* signal high while driving the *wait* signal low. This indicates to the master that the associated byte of the transfer failed—any previous bytes in the burst are considered to have succeeded; any subsequent bytes in the burst are abandoned. The assertion of *error* always terminates the Data Phase even if *bip* is asserted simultaneously.

What Drives What When

Table 8-2 What Drives What When

Signal Name	Arbitration Phase	Address Phase	Data Phase
start	Driven to 1 by arbiter	Driven to 0 by arbiter	Driven to 0 by arbiter
addr	Not driven	Driven by master (or to 0 by arbiter for NOP)	Not driven
size	Not driven	Driven by master (or to 0 by arbiter for NOP)	Not driven
read	Not driven	Driven by master (or to 0 by arbiter for NOP)	Not driven
write	Not driven	Driven by master (or to 0 by arbiter for NOP)	Not driven
bip	Not driven	Not driven	Driven to 1 by master for all but last byte of transfer
data	Not driven	Not driven	Driven by master during writes. Driven by slave during reads in cycles where *wait* is low; otherwise, don't care (may be driven to unknown state or not driven at all).
wait	Not driven	Not driven	Driven by slave

Table 8-2 What Drives What When (continued)

Signal Name	Arbitration Phase	Address Phase	Data Phase
error	Not driven	Not driven	Driven by slave

Optional Pipelining Scheme

As previously stated, the XBus standard does not normally support pipelining. However, pipelining can optionally be implemented.

Note: All agents (including arbitration) on a bus must agree either to pipeline or not to pipeline. Mixing pipelined and non-pipelined agents on the same bus is not supported.

Because pipelining overlaps the Arbitration, Address, and Data Phases, two levels of pipelining are provided. That is, there are a total of three transfers in progress at any one time.

Note: Pipelining results in different bus agents driving the same signals in consecutive clock cycles. As such, there is no period where the signal is not driven as part of a change of sequencers. As a result, care is necessary in the physical design of the bus to ensure that bus contention does not occur. A multiplexed approach will be required (in the form of either a ring or a star).

Pipelined Arbitration Phase

In a pipelined system, the Arbitration Phase is performed in parallel with the Address and Data Phases. Arbitration is carried out in every clock cycle regardless of whether this is necessary or not. This is because the arbiter cannot predict whether the next clock cycle will mark the start of a new Address Phase.

The Arbiter asserts the *start* signal in the clock cycle after the end of each Data Phase as in the non-pipelined system. However, this *start* signal marks the start of all three Phases in parallel.

The end of a Data Phase can be recognized by either assertion of *error* or de-assertion of both *bip* and *wait*.

Pipelined Address Phase

A master that has its *gnt* signal asserted at the clock edge where a Data Phase completes is granted the Address Phase of the bus. It must immediately start driving an Address Phase. Unlike in the non-pipelined bus, where the Address Phase lasts a single clock cycle, the Address Phase in a pipelined bus lasts until the end of the next Data Phase.

Where no master requests the bus and therefore no master is granted the bus, the arbiter is responsible for driving NOP until the end of the next Data Phase.

Pipelined Data Phase

The Data Phase of a pipelined bus is similar to that of a non-pipelined bus. Where the arbiter drives a NOP for the preceding Address Phase, the master must drive *error*, *bip* and *wait* low during the Data Phase (which will last for a single clock cycle in this case).

Example Timing Diagrams

Figure 8-1 Example Write Waveform

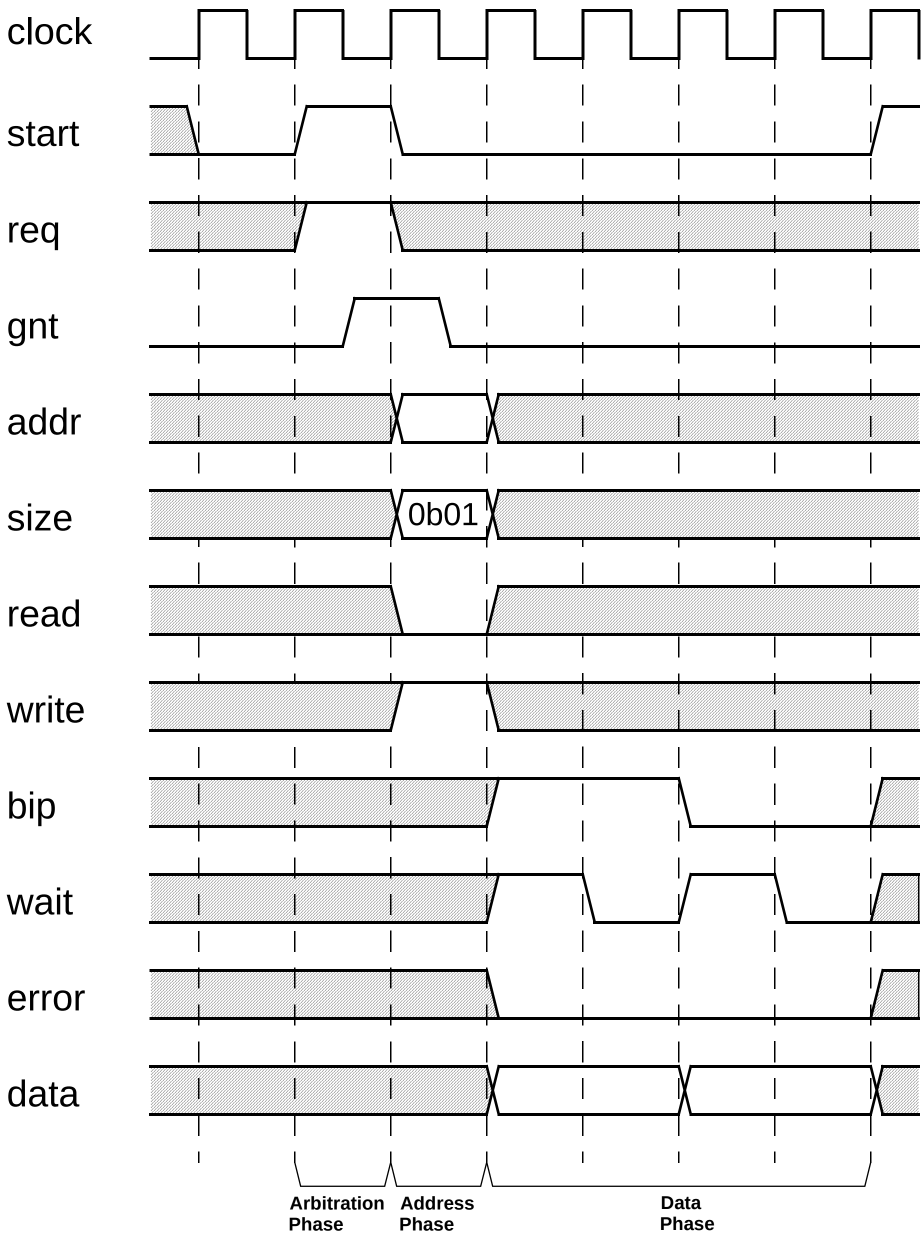


Figure 8-2 Example Read Waveform