Universal Verification Methodology (UVM) 1.0 EA User's Guide

May 14, 2010

Notices

Suggestions for improvements to the UVM 1.0 EA User's Guide are welcome. They should be sent to the VIP email reflector

vip-tc@lists.accellera.org

The current Working Group's website address is

www.accellera.org/activities/vip

Introduction

While this guide offers a set of instructions to perform one or more specific verification tasks, it should be supplemented by education, experience, and professional judgment. Not all aspects of this guide may be applicable in all circumstances. The UVM 1.0 EA User's Guide does not necessarily represent the standard of care by which the adequacy of a given professional service must be judged nor should this document be applied without consideration of a project's unique aspects. This guide has been approved through the Accellera consensus process and serves to increase the awareness of information and approaches in verification methodology. This guide may have several recommendations to accomplish the same thing and may require some judgment to determine the best course of action.

The *UVM 1.0 EA Class Reference* represents the foundation used to create the UVM 1.0 EA User's Guide. This guide is a way to apply the *UVM 1.0 EA Class Reference*, but is not the only way. Accellera believes standards are an important ingredient to foster innovation and continues to encourage industry innovation based on its standards.

Contents

1. Overview

This chapter describes:

— How to use the Universal Verification Methodology (UVM) for creating SystemVerilog testbenches.

— The recommended architecture of a verification component.

1.1 Introduction to UVM

The following subsections describe the UVM basics.

1.1.1 Coverage-Driven Verification (CDV)

UVM 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.

1.1.2 Testbenches and Environments

An UVM testbench is composed of reusable verification environments called *verification components*. A verification component is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, a design submodule, or a full system. Each verification component 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 verification component is applied to the device under test (DUT) to verify your implementation of the protocol or design architecture.

Figure 1 shows an example of a verification environment with three interface verification components. These verification components might be stored in a company repository and reused for multiple verification environments. The interface verification component 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.

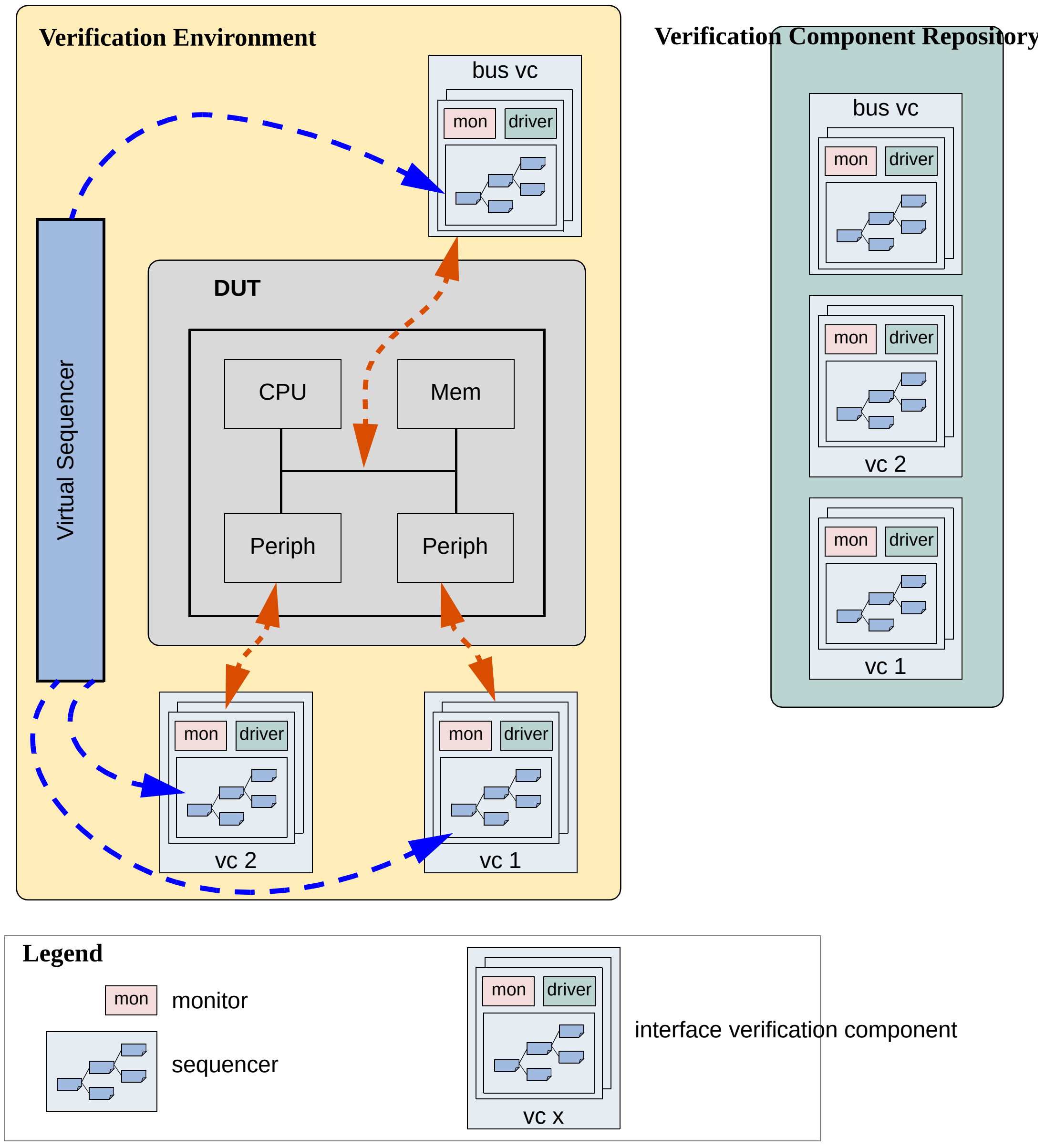


Figure 1—Verification Environment Example

1.2 Verification Component Overview

The following subsections describe the components of a verification component.

1.2.1 Data Item (Transaction)

Data items represent the input to the device under test (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.

1.2.2 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.

1.2.3 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 *UVM Class Reference* and to Section 5.9, Section 6.5.2, and Section 6.6.2.

Sequencers also can be layered on top of each other to model protocol layering. Refer to Section 7.5.2.5 for more information.

1.2.4 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.

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

1.2.5 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, UVM 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. Verification components 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.

1.2.6 Environment

The environment (env) is the top-level component of the verification component. 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 illustrates the structure of a reusable verification environment. Notice that a verification component 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.

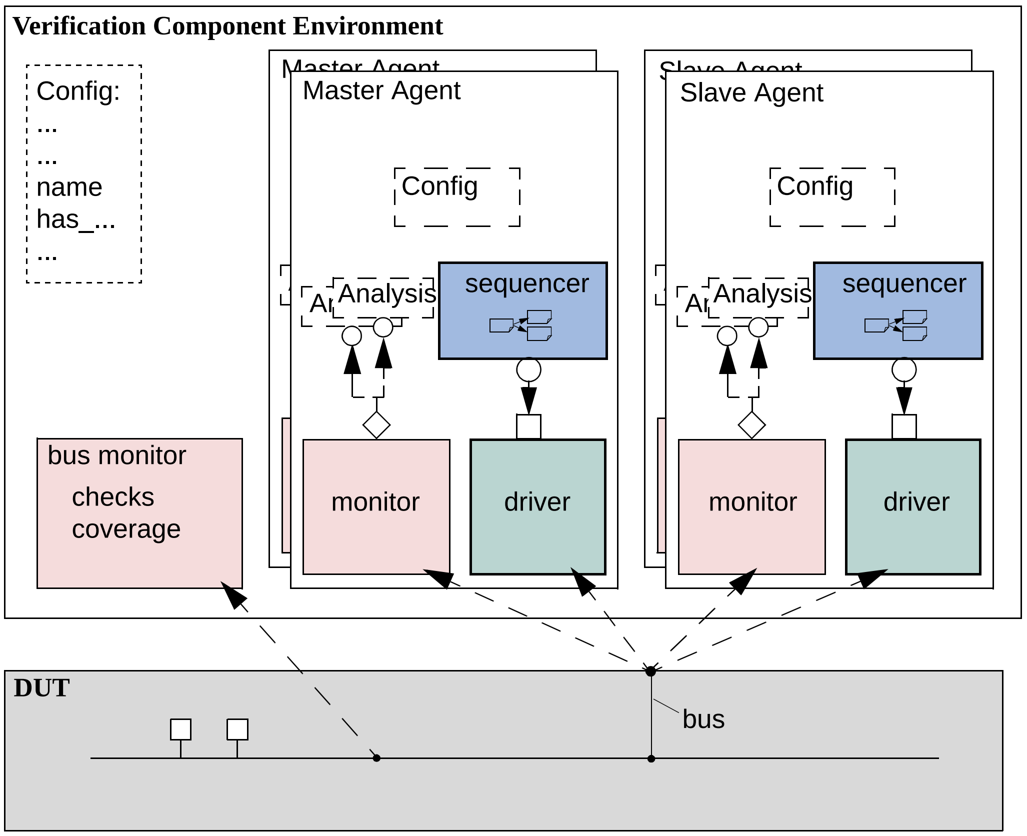


Figure 2—Typical Verification Component Environment

The environment class (`uvm_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 UVM provides to replace existing component behavior with IP-specific behavior.

1.3 The UVM Class Library

The UVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments (see Figure 3). 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 *UVM Class Reference* for more information.

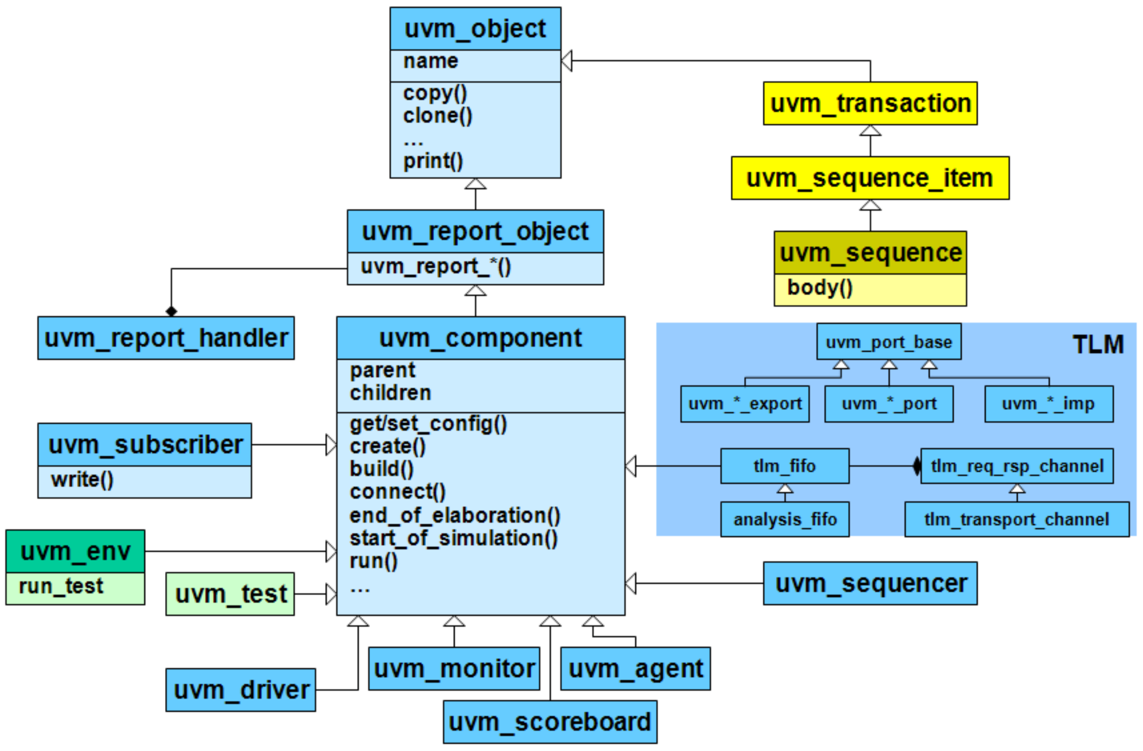


Figure 3—(Partial) UVM Class Hierarchy

The advantages of using the UVM Class Library include:

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

b) Correctly-implemented UVM concepts—Each component in the block diagram in Figure 2 is derived from a corresponding UVM Class Library component. Figure 4 shows the same diagram using the derived UVM 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.

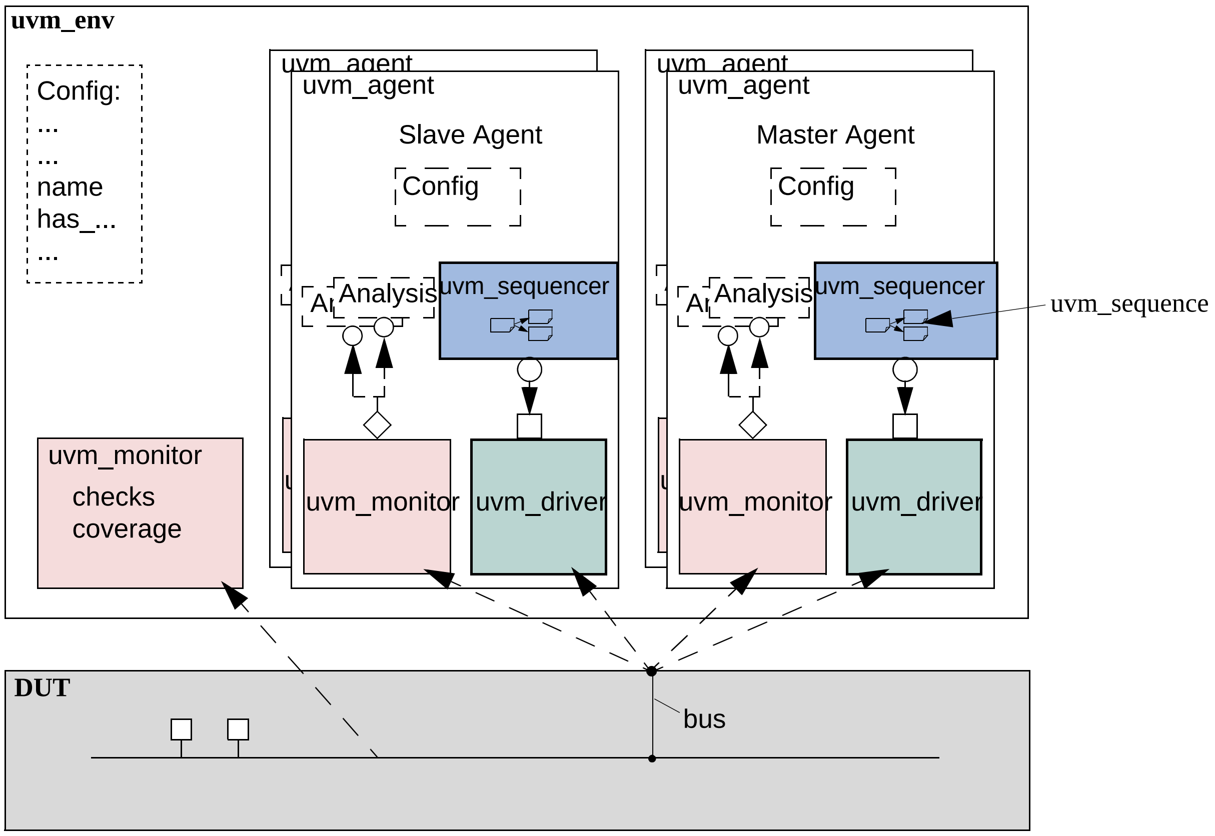


Figure 4—Typical UVM Environment using Library Classes

1.4 Other UVM Facilities

The UVM 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 (UVM factory).

The UVM 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.

1.4.1 UVM 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 UVM 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).

Using the UVM 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.

1.4.2 Transaction-Level Modeling (TLM)

UVM components communicate via standard TLM interfaces, which improves reuse. Using a SystemVerilog implementation of TLM in UVM, 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.

2. Normative References

The following referenced documents are indispensable for the application of this specification (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.[1, 2]

3. Definitions, Acronyms, and Abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary: Glossary of Terms & Definitions*[3] should be referenced for terms not defined in this chapter.

3.1 Definitions

agent: An abstract container used to emulate and verify DUT devices; agents encapsulate a **driver**, **sequencer**, and **monitor**.

blocking: An interface where tasks block execution until they complete. See also: **non blocking**.

component: A piece of VIP that provides functionality and interfaces. Also referred to as a *transactor*.

consumer: A verification component that receives **transaction**s from another **component**.

driver: A component responsible for executing or otherwise processing **transaction**s, usually interacting with the device under test (DUT) to do so.

environment: The container object that defines the **testbench** topology.

export: A transaction level modeling (TLM) interface that provides the implementation of methods used for communication. Used in UVM to connect to a port.

factory method: A classic software design pattern used to create generic code by deferring, until run time, the exact specification of the object to be created.

foreign methodology: A verification methodology that is different from the methodology being used for the majority of the verification environment.

generator: A verification component that provides transactions to another **component**. Also referred to as a *producer*.

monitor: A passive entity that samples DUT signals, but does not drive them.

non blocking: A call that returns immediately. See also: **blocking**.

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[2]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[3]The *IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.

port: A TLM interface that defines the set of methods used for communication. Used in UVM to connect to an export.

primary (host) methodology: The methodology that manages the top-level operation of the verification environment and with which the user/integrator is presumably more familiar.

request: A **transaction** that provides information to initiate the processing of a particular operation.

response: A **transaction** that provides information about the completion or status of a particular operation.

scoreboard: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually refers to the entire dynamic response-checking structure.

sequence: An UVM object that procedurally defines a set of **transaction**s to be executed and/or controls the execution of other sequences.

sequencer: An advanced stimulus generator which executes **sequence**s that define the **transaction**s provided to the **driver** for execution.

test: Specific customization of an environment to exercise required functionality of the DUT.

testbench: The structural definition of a set of verification components used to verify a DUT. Also referred to as a *verification environment*.

transaction: A class instance that encapsulates information used to communicate between two or more **component**s.

transactor: See *component*.

virtual sequence: A conceptual term for a **sequence** that controls the execution of **sequence**s on other **sequencer**s.

3.2 Acronyms and Abbreviations

API	application programming interface
CDV	coverage-driven verification
CBCL	common base class library
EDA	electronic design automation
FIFO	first-in, first-out
HDL	hardware description language
HVL	high-level verification language
IP	intellectual property
DUT	device under test
OSCI	Open SystemC Initiative

OVM	Open Verification Methodology
TLM	transaction level modeling
UVM	Universal Verification Methodology
VIP	verification intellectual property
VMM	Verification Methodology Manual

4. Transaction-Level Modeling (TLM)

4.1 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 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.

UVM 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 UVM, TLM promotes reuse by allowing any component to be swapped for another, as long as they have the same interfaces. This concept also allows UVM 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 *verification components*, 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 UVM, 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.

4.2 Basics

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

4.2.1 Transactions

In UVM, a transaction is a class object, `uvm_transaction` (extended from `uvm_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 uvm_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.

4.2.2 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 application programming interface (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.

4.2.3 Basic TLM Communication

The most basic transaction-level operation allows one component to *put* a transaction to another. Consider Figure 5.

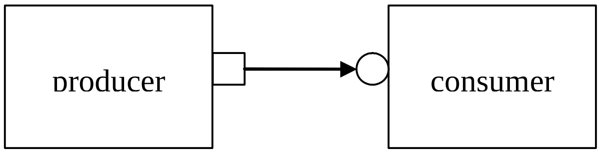


Figure 5—Simple Producer/Consumer

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 uvm_component;
  uvm_blocking_put_port #(simple_trans) put_port; // 1 parameter
  function new( string name, uvm_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 `uvm_*_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 uvm_component;
  uvm_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 `uvm_*_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 (`uvm_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.

The converse operation to put is *get*. Consider Figure 6.

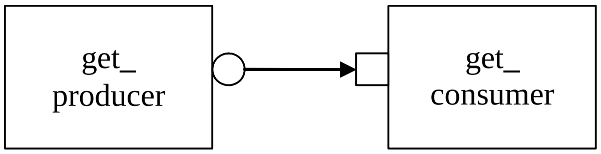


Figure 6—Consumer gets from Producer

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

```
class get_consumer extends uvm_component;
  uvm_blocking_get_port #(simple_trans) get_port;
  function new( string name, uvm_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 uvm_component;
  uvm_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.

4.2.4 Communicating between Processes

In the basic `put` example above, the consumer will be active only 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. UVM 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 7.

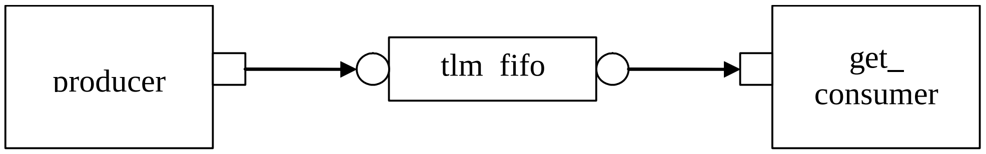


Figure 7—Using a tlm_fifo

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.

4.2.5 Blocking versus Nonblocking

The interfaces that we have looked at so far are blocking—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 UVM, nonblocking calls are modeled as functions.

```
class consumer extends uvm_component;
  uvm_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 returns `TRUE` if the transaction is sent.

4.2.6 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.

4.2.7 Peer-to-Peer connections

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

```
class my_env extends uvm_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
```

4.2.8 Port/Export Compatibility

Another advantage of TLM communication in UVM 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()`.

4.3 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 Chapter 5). 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.

4.3.1 Hierarchical Connections

Making connections across hierarchical boundaries involves some additional issues, which are discussed in this section. Consider the hierarchical design shown in Figure 8.

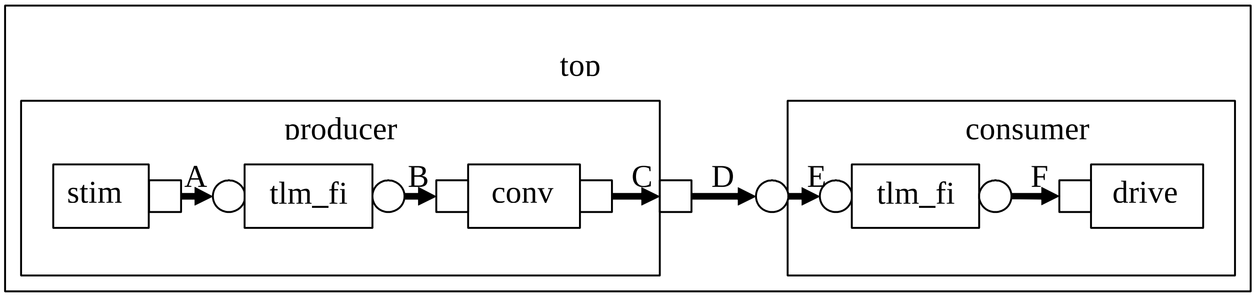


Figure 8—Hierarchy in TLM

The hierarchy of this design contains two components, `producer` and `consumer`. `producer` contains three components, `stim`, `tlm_fi`, and `conv`. `consumer` contains two components, `tlm_fi` and `drive`. Notice that, from the perspective of `top`, the producer and consumer appear identical to those in Figure 5, 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 8, 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 uvm_component;
         uvm_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 uvm_component;
     uvm_put_port #(trans) put_port;
     conv c;
     ...
  function void connect();
       c.put_port.connect(put_port);
       ...
     endfunction
```

4.3.2 Connection Types

Table 1 summarizes connection types and elaboration functions.

Table 1—TLM Connection Types

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.

4.4 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.

4.4.1 Analysis Ports

The `uvm_analysis_port` (represented as a diamond on the monitor in Figure 9) 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 zero, one, 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.

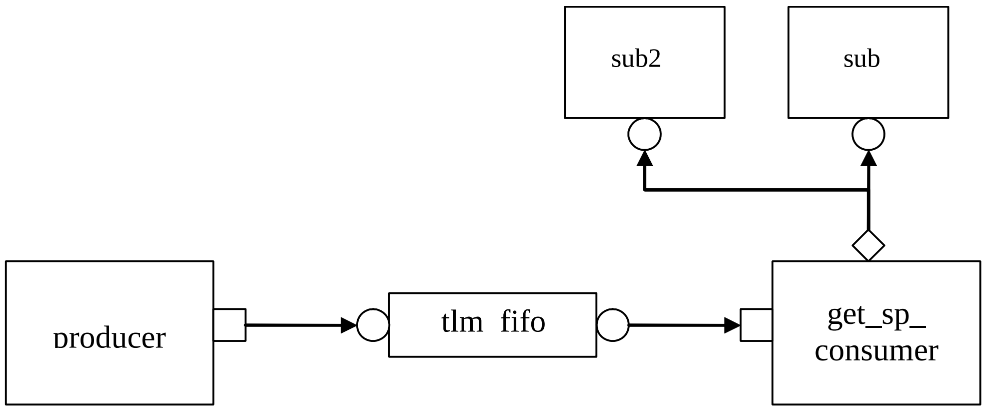


Figure 9—Analysis Communication

```
class get_ap_consumer extends get_consumer;
      uvm_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.

4.4.2 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`. UVM provides the `uvm_subscriber` base component to simplify this operation, so a typical analysis component would extend `uvm_subscriber` as:

```
class sub1 #(type T = simple_trans) extends uvm_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 uvm_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.

UVM 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.

5. Developing Reusable Verification Components

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 verification components. The sections in this chapter follow the same order you should follow when developing a verification component:

— *Modeling Data Items for Generation*
— *Transaction-Level Components*
— *Creating the Driver*
— *Creating the Sequencer*
— *Creating the Monitor*
— *Instantiating Components*
— *Creating the Agent*
— *Creating the Environment*
— *Enabling Scenario Creation*
— *Managing End of Test*
— *Implementing Checks and Coverage*

NOTE—This chapter builds upon concepts described in Chapter 1 and Chapter 4.

5.1 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 UVM Class Library provides the `uvm_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:

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

UVM 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.

UVM 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 `uvm_transaction` base class includes the `m_transaction_id` field. In addition, the `uvm_sequence_item` base class (extended from uvm_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 UVM macros implement various utilities that operate on this class, such as copy, compare, print, and so on. In particular, the `uvm_object_utils macro registers the class type with the common factory.

```
1  class simple_item extends uvm_sequence_item;
2    rand int unsigned addr;
3    rand int unsigned data;
4    rand int unsigned delay;
5    constraint c1 { addr < 16'h2000; }
6    constraint c2 { data < 16'h1000; }
7    // UVM automation macros for general objects
8    `uvm_object_utils_begin(simple_item)
9      `uvm_field_int(addr, UVM_ALL_ON)
10     `uvm_field_int(data, UVM_ALL_ON)
11     `uvm_field_int(delay, UVM_ALL_ON)
12   `uvm_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 uvm_sequence_item so they can be generated in a procedural sequence. See Section 5.9.2 for more information.

Line 5 and Line 6Add 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.

Line 7-Line 12 Use the UVM macros to automatically implement functions such as copy(), compare(), print(), pack(), and so on. Refer to "UVM Macros" in the *UVM Class Reference* for information on the `uvm_object_utils_begin, `uvm_object_utils_end, `uvm_field_*, and their associated macros.

NOTE—UVM 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.

5.1.1 Inheritance and Constraint Layering

In order to meet verification goals, the verification component 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; }
     `uvm_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.

5.1.2 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 Section 5.1, 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 uvm_sequence_item;
     rand int unsigned addr;
     rand int unsigned data;
     rand int unsigned delay;
     rand simple_item_delay_e delay_kind; // Control field
     // UVM automation macros for general objects
     `uvm_object_utils_begin(simple_item)
       `uvm_field_int(addr, UVM_ALL_ON)
       `uvm_field_enum(simple_item_delay_e, delay_kind, UVM_ALL_ON)
     `uvm_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 == LARGE) -> 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.

5.2 Transaction-Level Components

As discussed in Chapter 4, TLM interfaces in UVM 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 10.

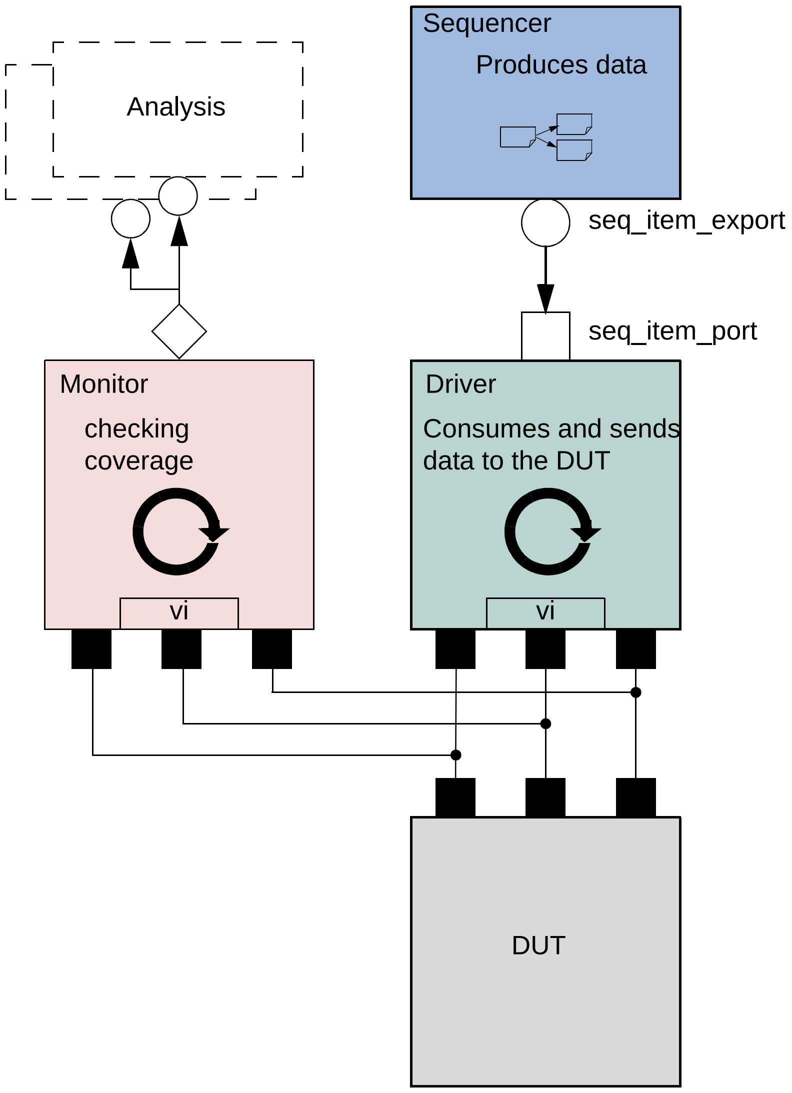


Figure 10—Simplified Transaction-Level Testbench

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

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

b) A driver to convert these transactions to signal-level stimulus at the DUT interface.

c) A monitor to recognize signal-level activity on the DUT interface and convert it into transactions.

d) 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 UVM 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 (see Figure 11). This chapter discusses how to encapsulate these types of components into a proven architecture, a verification component, to improve reuse even further.

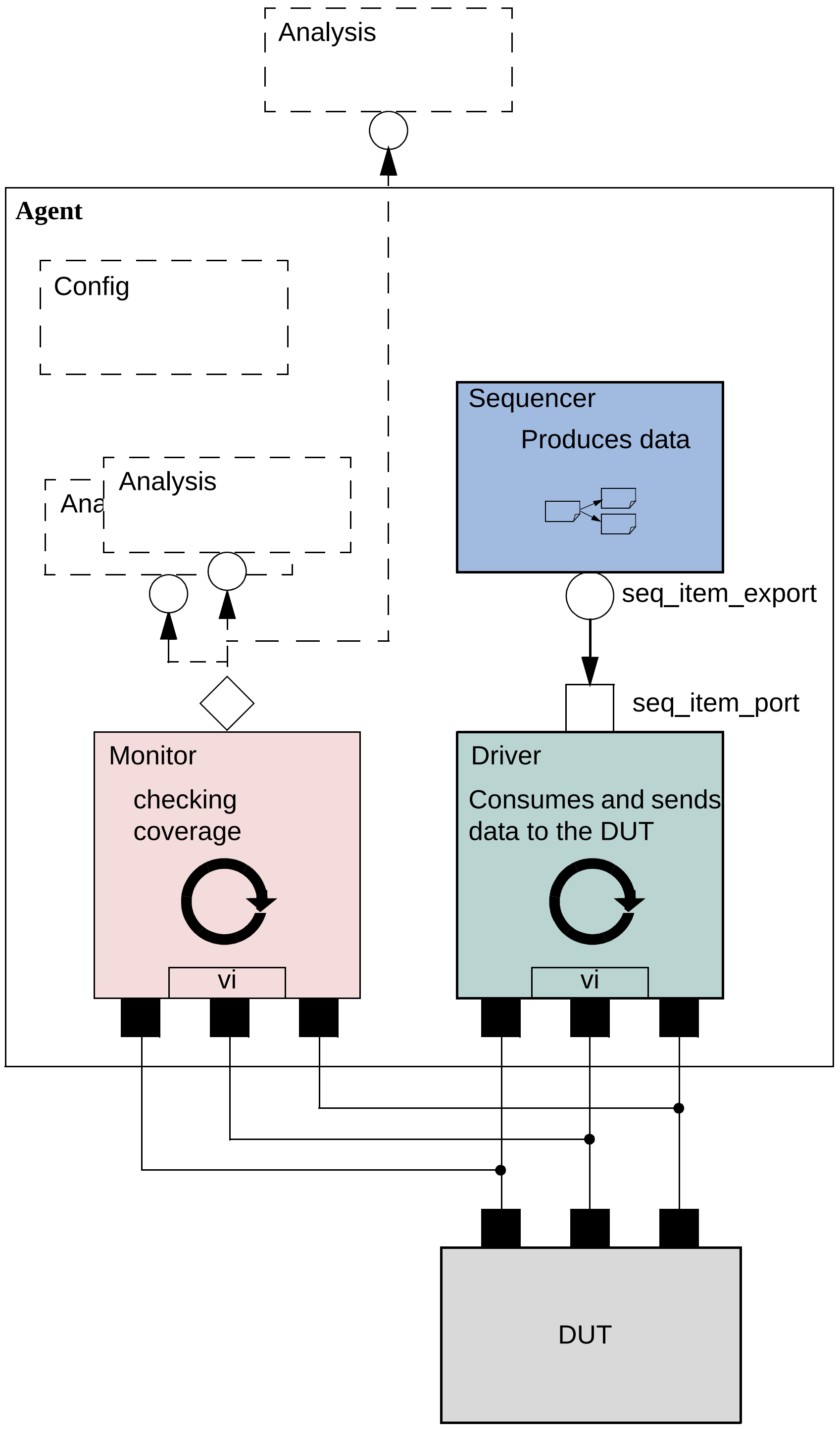


Figure 11—Highly Reusable Verification Component Agent

Figure 11 shows the recommended grouping of individual components into a reusable interface-level verification component 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.

5.3 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 UVM Class Library provides the `uvm_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:

a) Derive a driver from the `uvm_driver` base class.

b) If desired, add UVM infrastructure macros for class properties to implement utilities for printing, copying, comparing, and so on.

c) Obtain the next data item from the sequencer and execute it as outlined above.

d) Declare a virtual interface in the driver to connect the driver to the DUT.

Refer to Section 5.9.2 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 `uvm_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 `` `uvm_component_utils `` macro to register the driver type with the common factory.

```
class simple_driver extends uvm_driver #(simple_item);
  simple_item s_item;
  virtual dut_if vif;
  // UVM automation macros for general components
  `uvm_component_utils(simple_driver)
  // Constructor
  function new (string name = "simple_driver", uvm_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.
      drive_item(s_item);
      seq_item_port.item_done(); // Consume the request.
    end
  endtask : run

  task drive_item (input simple_item item);
    ... // Add your logic here.
  endtask : drive_item
endclass : simple_driver
```

Line 1 Derive the driver.

Line 5 Add UVM 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 Section 5.4.1.

5.4 Creating the Sequencer

The sequencer generates stimulus data and passes it to a driver for execution. The UVM Class Library provides the `uvm_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:

a) Derive a sequencer from the `uvm_sequencer` base class and specify the `request` and `response` type parameters.

b) Use `` `uvm_sequencer_utils `` and `` `uvm_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 Section 5.9.2 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 `uvm_sequencer` and parameterizes it to use the `simple_item` type.

```
class simple_sequencer extends uvm_sequencer #(simple_item);
     // UVM automation macro for sequencers
     `uvm_sequencer_utils(simple_sequencer)
     // Constructor
     function new (string name="simple_sequencer", uvm_component parent);
       super.new(name, parent);
       `uvm_update_sequence_lib_and_item(simple_item)
     endfunction : new
   endclass : simple_sequencer
```

The following also apply.

— 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 `uvm_sequencer` base type:

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

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

— Call `` `uvm_update_sequence_lib_and_item `` 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 “UVM Macros” in the *UVM Class Reference* for more information.

5.4.1 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 12). 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 Section 5.7.

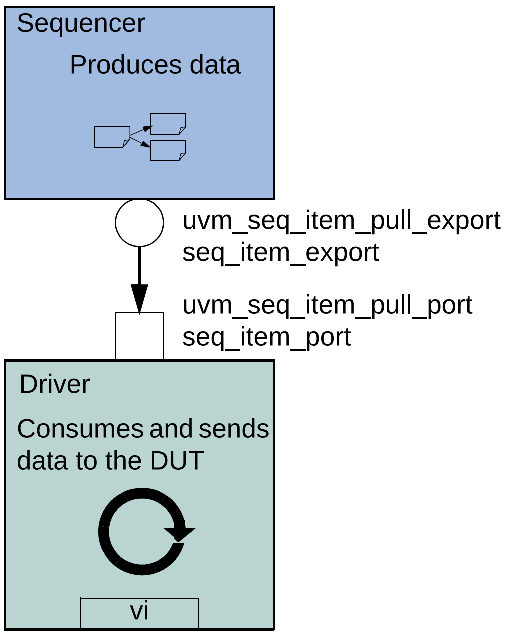


Figure 12—Sequencer-Driver Interaction

The `seq_item_port` in `uvm_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.

5.4.1.1 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 Section 5.3, 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.

5.4.1.2 Querying for the Randomized Item

In addition to the `get_next_item()` task, the `uvm_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 Section 5.3), 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
```

5.4.2 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{
       ...
     }
   }
```

5.4.3 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 be 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 `uvm_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.

5.4.4 Using TLM-Based Drivers

The `seq_item_port`, which is built into `uvm_driver`, is a bidirectional port. It also includes the 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 `uvm_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);
// Allow sequencer to proceed immediately upon driver receiving transaction.
   get(req);
// Process req operation.
```

The following also apply.

— `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.

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.

5.5 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 *UVM 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 uvm_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.
  uvm_analysis_port #(simple_item) item_collected_port;
     event cov_transaction; // Events needed to trigger covergroups
  protected simple_item trans_collected;
  `uvm_component_utils_begin(master_monitor)
       `uvm_field_int(checks_enable, UVM_ALL_ON)
       `uvm_field_int(coverage_enable, UVM_ALL_ON)
     `uvm_component_utils_end
  covergroup cov_trans @cov_transaction;
       option.per_instance = 1;
       ... // Coverage bins definition
     endgroup : cov_trans
  function new (string name, uvm_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
    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.

5.6 Instantiating Components

The isolation provided by object-oriented practices and TLM interfaces between components facilitate reuse in UVM 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 UVM.

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

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

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

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

The factory operation is explained in Section 7.3. 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:

a) 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
```

b) 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.

5.7 Creating the Agent

An agent (see Figure 13) 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 Section 1.2.5, UVM recommends that the verification component 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.

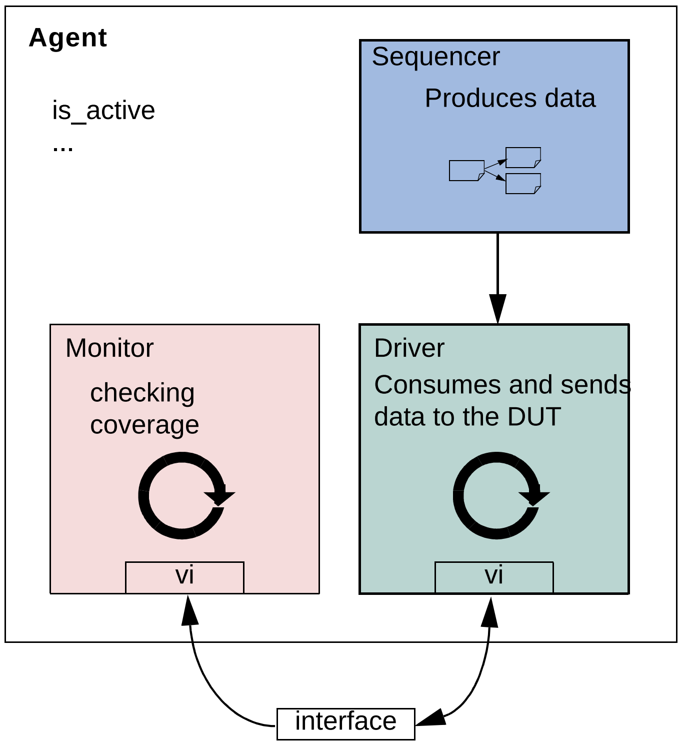


Figure 13—Agent

5.7.1 Operating Modes

An agent has two basic operating modes:

— Active mode, where 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, where 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.

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 UVM `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 `type_id::create()` is used to instantiate the subcomponents. The example in “To change the driver for a specific test:” in Section 5.7 illustrates how you can override existing behavior using `extends`.

```
1  class simple_agent extends uvm_agent;
2    uvm_active_passive_enum is_active;
3    ... // Constructor and UVM automation macros
4    simple_sequencer sequencer;
5    simple_driver driver;
6    simple_monitor monitor;
7    // Use build() phase to create agents's subcomponents.
8    virtual function void build();
9      super.build()
10     monitor = simple_monitor::type_id::create("monitor",this);
11     if (is_active == UVM_ACTIVE) begin
12       // Build the sequencer and driver.
13       sequencer = simple_sequencer::type_id::create("sequencer",this);
14       driver = simple_driver::type_id::create("driver",this);
15     end
16   endfunction : build
17   virtual function void connect();
18     if(is_active == UVM_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()`.

Line 11 - Line 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 = UVM_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 Chapter 8 for a complete interface verification component 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.

Line 18 - Line 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()`.

5.7.2 Connecting Components

The `connect()` phase, which happens after the build is complete, should be used to connect the components inside the agent. See Line 18 - Line 20 in the example in Section 5.7.1.

5.8 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 (see Figure 14). By following the guidelines here, you can ensure that your environment will be architecturally correct, consistent with other verification components, and reusable. The following sections describe how to create and connect environment sub-components.

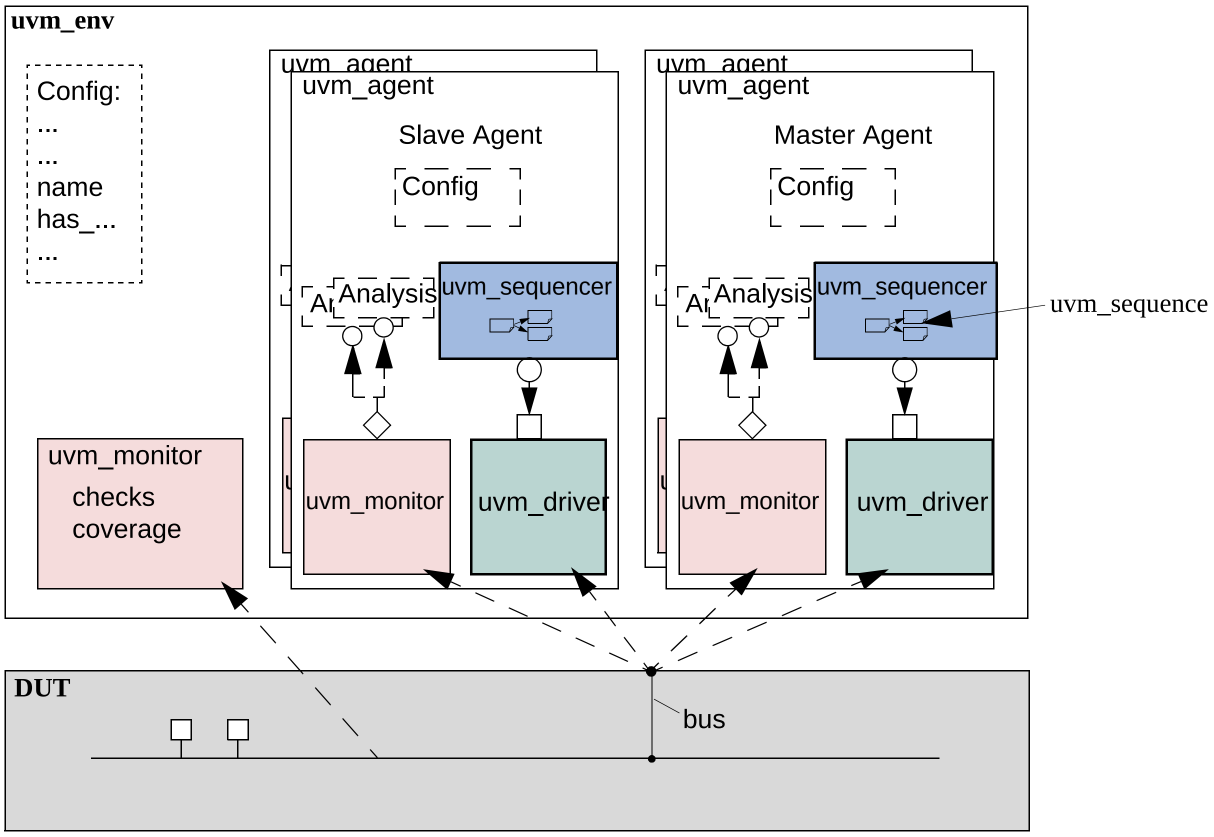


Figure 14—Typical UVM Environment Architecture

5.8.1 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 uvm_env;
     int num_masters;
     ahb_master_agent masters[];
  `uvm_component_utils_begin(ahb_env)
       `uvm_field_int(num_masters, UVM_ALL_ON)
     `uvm_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] = ahb_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, uvm_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 UVM 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.

5.8.2 The UVM Configuration Mechanism

A verification component 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. UVM provides a standard configuration mechanism which allows you to define the verification component's configuration to suit the current project's requirements. The verification component 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 UVM fields using the `uvm_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 UVM 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.

5.8.2.1 Making the Verification Component Reusable

There are times when you as the developer know the context in which the verification component you are developing will be used. In such cases you should take care to separate the requirements of the verification component's protocol from those of the project. It is strongly recommended that you use only the interface-protocol documentation in developing the verification component. 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, when a few bits are defined as reserved in a protocol frame, they should stay reserved within the verification component. 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.

5.8.2.2 How to Create a Configurable Attribute

Making an attribute configurable is part of the built-in automation that the UVM 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 Section 5.8.1, `num_master` is a configuration parameter that allows changing the master agent numbers as needed. Since the `` `uvm_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 specify:

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

This can be done in procedural code within the testbench. For more information, see Section 6.3.

The following also apply.

— 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 takes precedence.

5.9 Enabling Scenario Creation

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

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 Section 6.5.

5.9.1 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 UVM Class Library provides the `uvm_sequence` base class. You should derive all sequence classes directly or indirectly from this class.

To create a user-defined sequence:

a) Derive a sequence from the `uvm_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`.

b) Use the ``uvm_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.

c) 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 ``uvm_do`` (see Section 5.9.2.2.1) and ``uvm_do_with`` (see Section 5.9.2.2.2).

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

```
class simple_seq_do extends uvm_sequence #(simple_item);
     rand int count;
     constraint c1 { count >0; count <50; }
     // Constructor
     function new(string name="simple_seq_do");
       super.new(name);
     endfunction
     // UVM automation macros for sequences
     `uvm_sequence_utils(simple_seq_do, simple_sequencer)
     // The body() task is the actual logic of the sequence.
     virtual task body();
       repeat(count)
         `uvm_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 ``uvm_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 `` `uvm_object_utils `` macro (and its variations), except it takes a second argument, which is the sequencer type name this sequence is associated with.

NOTE—Do not use the `` `uvm_object_utils `` macro when using the `` `uvm_sequence_utils `` macro. The functionality of `` `uvm_object_utils `` is included in `` `uvm_sequence_utils ``.

5.9.2 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.

5.9.2.1 Getting Started with Sequences

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

Figure 15 and Figure 16 show the complete flow for sequence items and sequences when used with the `uvm_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.

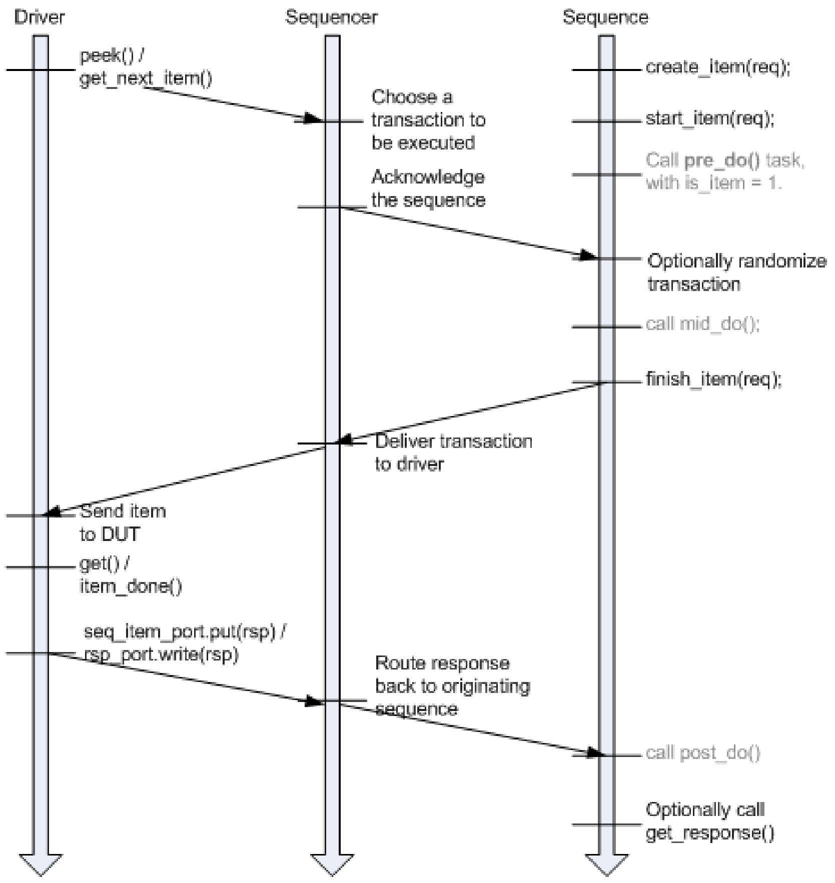
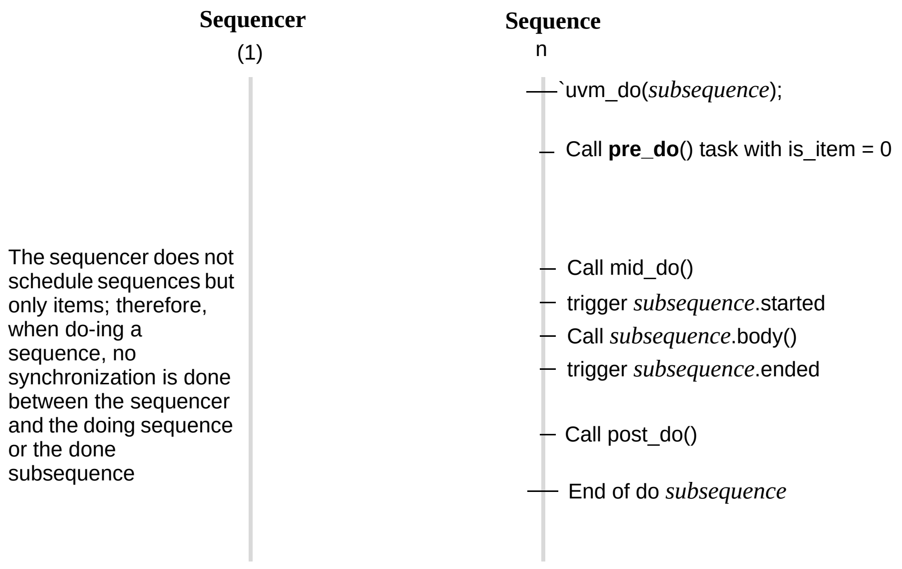

Driver
Sequencer
Sequence
peek() / get_next_item()
create_item(req);
Choose a transaction to be executed
start_item(req);
Call pre_do() task, with is_item = 1.
Acknowledge the sequence
Optionally randomize transaction
call mid_do();
finish_item(req);
Deliver transaction to driver
Send item to DUT
get() / item_done()
seq_item_port.put(rsp) / rsp_port.write(rsp)
Route response back to originating sequence
call post_do()
Optionally call get_response()

Figure 15—Sequence Item Flow in Pull Mode

The `uvm_do macro and all related macros provide a convenient set of calls to create, randomize, and send transaction items in a sequence. The `uvm_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 (uvm_do_with) or bypass the randomization altogether. The individual methods wrapped by `uvm_do in Figure 15 may be called individually with no loss of functionality:

a) Call create_item() to create the item via the factory.
b) Call start_item().
c) Optionally call pre_do() or some other functionality.
d) Optionally randomize *item*.
e) Optionally call mid_do() or some other functionality, if desired.
f) Call finish_item().
g) Optionally call post_do() or some other functionality.
h) Optionally call get_response().


Sequencer
(1)
Sequence
n
`uvm_do(subsequence);
Call pre_do() task with is_item = 0
The sequencer does not schedule sequences but only items; therefore, when do-ing a sequence, no synchronization is done between the sequencer and the doing sequence or the done subsequence
Call mid_do()
trigger subsequence.started
Call subsequence.body()
trigger subsequence.ended
Call post_do()
End of do subsequence

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

Figure 16—Subsequence Flow

5.9.2.2 Sequence and Sequence Item Macros

This section describes the sequence and sequence item macros, `` `uvm_do `` and `` `uvm_do_with ``.

5.9.2.2.1 `uvm_do

This macro takes as an argument a variable of type `uvm_sequence` or `uvm_sequence_item`. An object is created using the factory settings and assigned to the specified variable. Based on the processing in Figure 15, 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 Section 5.9.1 is repeated here. The body of the sequence invokes an item of type `simple_item`, using the `` `uvm_do `` macro.

```
class simple_seq_do extends uvm_sequence #(simple_item);
     ... // Constructor and UVM automation macros
     // See Section 6.5.2.3
     virtual task body();
       `uvm_do(req)
     endtask : body
   endclass : simple_seq_do
```

Similarly, a sequence variable can be provided and will be processed as shown in Figure 16. The following example declares another sequence (`simple_seq_sub_seqs`), which uses `` `uvm_do `` to execute a sequence of type `simple_seq_do`, which was defined earlier.

```
class simple_seq_sub_seqs extends uvm_sequence #(simple_item);
     ... // Constructor and UVM automation macros
     // See Section 6.5.2.3
     simple_seq_do seq_do;
```

```
    virtual task body();
      `uvm_do(seq_do)
    endtask : body
  endclass : simple_seq_sub_seqs
```

5.9.2.2.2 `uvm_do_with

This macro is similar to `` `uvm_do `` (Section 5.9.2.2.1). The first argument is a variable of a type derived from `uvm_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 uvm_sequence #(simple_item);
      ... // Constructor and UVM automation macros
      // See Section 6.5.2.3
      virtual task body();
        `uvm_do_with(req, { req.addr == 16'h0120; req.data == 16'h0444; } )
        `uvm_do_with(req, { req.addr == 16'h0124; req.data == 16'h0666; } )
      endtask : body
    endclass : simple_seq_do_with
```

5.9.3 Configuring the Sequencer's Default Sequence

Sequencers execute an `uvm_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 is used as the default sequence for the instance of the sequencer.

To override the default sequence:

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

   The example in Section 5.9.1 provides a declaration example of a sequence named `simple_seq_do`.

b) 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`.

5.9.4 Overriding Sequence Items and Sequences

In a user-defined `uvm_test`, for example `base_test_xbus_demo` (discussed in Section 6.4.1), 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 Section 7.3 for more information.

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

a) 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`.

b) Invoke the appropriate `uvm_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 Section 5.9.1). 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 UVM 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());
```

c) Use any of the sequence macros that allocate an object (as defined in Section 5.9.2.2), for example, the `` `uvm_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`.

5.9.5 Building a Reusable Sequence Library

A reusable sequence library is a set of user-defined sequences. Creating a verification component 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 verification component files. Leave it to the user to decide whether to use them.

5.10 Managing End of Test

UVM provides an objection mechanism to allow hierarchical status communication among components. The built-in objection, `uvm_test_done`, provides a way for components and objects to synchronize their testing activity and indicate when it is safe to end the test.

In general, the process is for a component or sequence to raise an `uvm_test_done` objection at the beginning of an activity that must be completed before the simulation stops and to drop the objection at the end of that activity. Once all of the raised objections are dropped, the `run` phase terminates via an implicit stop request.

In simulation, agents may have a meaningful agenda to be achieved before the test goals can be declared as done. For example, a master agent may need to complete all its read and write operations before the `run` phase should be allowed to stop. A reactive slave agent may not object to the end-of-test as it is merely serving requests as they appear without a well-defined agenda.

A typical use model of objections is for a sequence from an active agent to raise an `uvm_test_done` objection when it is started as a root sequence (a sequence which has no parent sequence), and to drop the objection when it is finished as a root sequence. This would look like the following:

```
class interesting_sequence extends uvm_sequence#(data_item);
  task pre_body();
       // raise objection if started as a root sequence
       uvm_test_done.raise_objection(this);
     endtask
  task body();
       //do interesting activity
       ...
     endtask
  task post_body();
       // drop objection if started as a root sequence
       uvm_test_done.drop_objection(this);
     endtask
endclass
```

When all objections are dropped, an implicit stop request is made to end the currently running phase (that is, `run`). In practice, there are times in simulation when the “all objections dropped” condition is temporary. For example, concurrently running processes may need some additional cycles to convey the last transaction to a scoreboard.

To accommodate this, you may set a drain time to inject a delay between the time a component’s total objection count reaches zero and when the drop is passed to its parent. If any objections are re-raised during this delay, the drop is cancelled and the raise is not propagated further. While a drain time can be set at each level of the component hierarchy with the adding effect, typical usage would be to set a single drain time at the `env` or `test` level. If you require control over drain times beyond a simple time value (for example, waiting for a few clock cycles or other user-defined events), you can also use the `all_dropped` callback to calculate drain times more precisely. For more information on the `all_dropped` callback, refer to the `uvm_objection` section in the *UVM Class Reference*.

Vertical reuse means building larger systems out of existing ones. What was once a top-level environment becomes a sub-environment of a large testbench. The objection mechanism allows sub-system environment developers to define a drain time per sub-system.

5.11 Implementing Checks and Coverage

Checks and coverage are crucial to a coverage-driven verification flow. SystemVerilog allows the usage shown in Table 2 for **assert**, **cover**, and **covergroup** constructs.

NOTE—This overview is for concurrent assertions. Immediate assertions can be used in any procedural statement. Refer to the SystemVerilog IEEE1800 LRM for more information.

Table 2—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 a verification component, checks and coverage are defined in multiple locations depending on the category of functionality being analyzed. In Figure 17, checks and coverage are depicted in the `uvm_monitor` and interface. The following sections describe how the **assert**, **cover**, and **covergroup** constructs are used in the XBus verification component example (described in Chapter 8).

5.11.1 Implementing Checks and Coverage in Classes

Class checks and coverage should be implemented in the classes derived from `uvm_monitor`. The derived class of `uvm_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 as the check becomes more complicated, so does the debugging of that check.

NOTE—Concurrent assertions are not allowed in SystemVerilog classes per the IEEE1800 LRM.

The following is a simple example of an assertion check. This assertion verifies the size field of the transfer is `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
```

The following is a simple example of a function check. This function verifies 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.

The 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 `uvm_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 `perform_transfer_checks()`, be called procedurally after the item has been collected by the monitor.

5.11.2 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 Chapter 5.11.3.

5.11.3 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 UVM bit field for this purpose. The field can be controlled using the `uvm_component` `set_config*` interface. Refer to `uvm_threaded_component` in the *UVM Class Reference* for more information. The following is an example of using the `checks_enable` bit to control the 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.

6. Using Verification Components

This chapter covers the steps needed to build a testbench from a set of reusable verification components. UVM accelerates the development process and facilitates reuse. UVM 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 UVM 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 verification components are:

a) Review the reusable verification component configuration parameters.
b) Instantiate and configure reusable verification components.
c) Create reusable sequences for interface verification components (optional).
d) Add a virtual sequencer (optional).
e) Add checking and functional coverage extensions.
f) Create tests to achieve coverage goals.

Before reading this chapter make sure you read Chapter 1. It is also recommended (but not required) that you read Chapter 5 to get a deeper understanding of verification components.

6.1 Using a Verification Component

As illustrated in Figure 1, 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.

6.1.1 Test Class

The `uvm_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 UVM 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 Section 6.4.

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

6.1.2 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, UVM recommends using a testbench class. The testbench class is derived from the `uvm_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 17 shows a typical verification environment that includes the test class containing the testbench class. Other environments (verification components) are contained inside the testbench class.

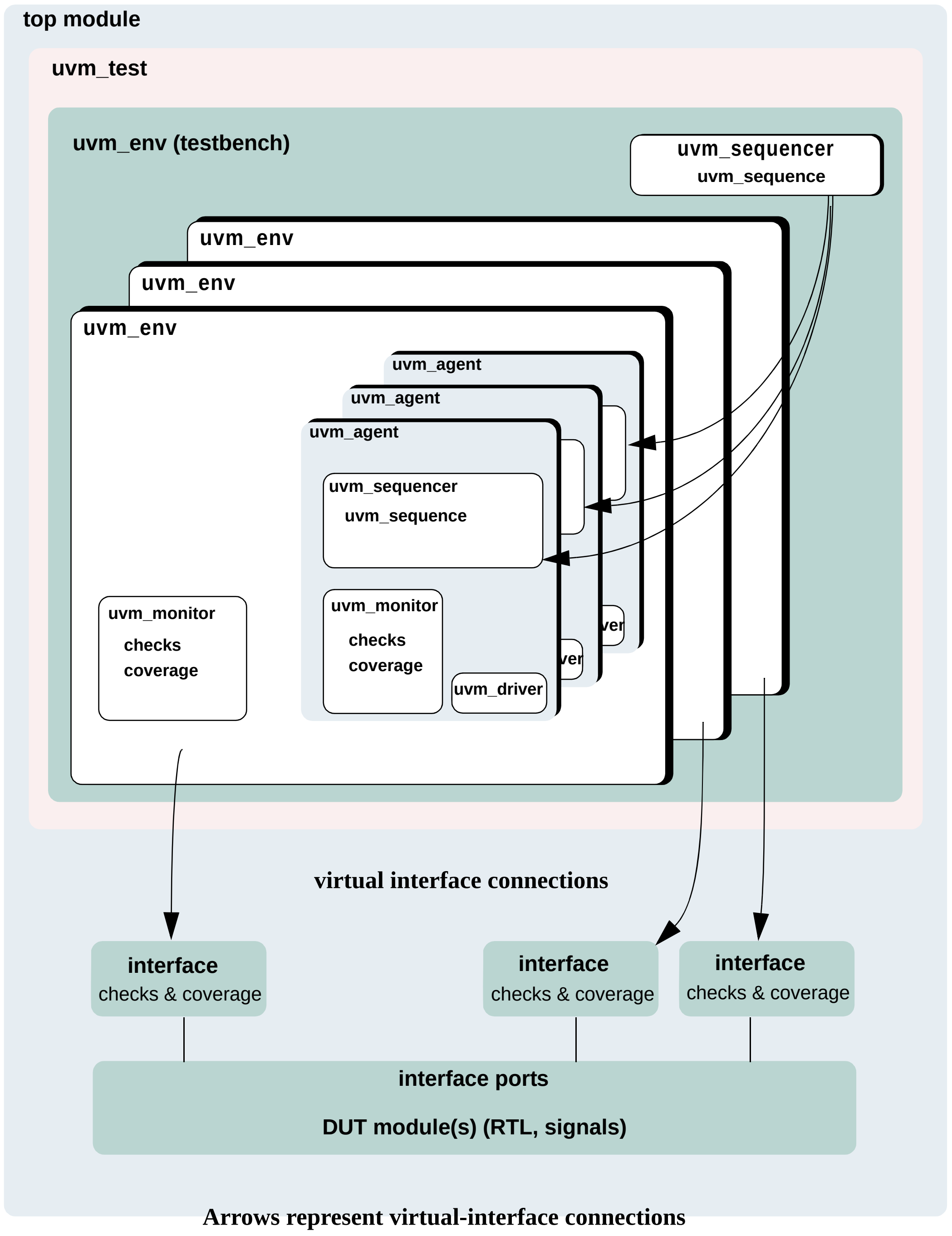


Figure 17—Verification Environment Class Diagram

6.2 Instantiating Verification Components

This section describes how you can use verification components to create a testbench that can be reused for multiple tests. The following example uses the verification IP in Chapter 8. This interface verification component 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.

The following also apply.

— 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 uvm_env;
    // Provide implementations of virtual methods such as get_type_name().
      `uvm_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, uvm_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", UVM_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", UVM_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 UVM 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 the `xbus_env` or the scoreboard classes can be replaced with derivative classes without changing the testbench file. See Section 7.3.3 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 SystemVerilog module. `connect()` is a built-in UVM 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.

6.3 Verification Component Configuration

6.3.1 Verification Component 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 verification component.

— The operation modes or speeds of a bus.

A verification component should support the standard configuration parameters and provide user-defined configuration parameters as needed. Refer to the verification component documentation for information about its user-defined parameters.

6.3.2 Verification Component Configuration Mechanism

UVM provides a configuration mechanism (see Figure 18) to allow integrators to configure an environment without needing to know the verification component implementation and hook-up scheme. The 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);
```

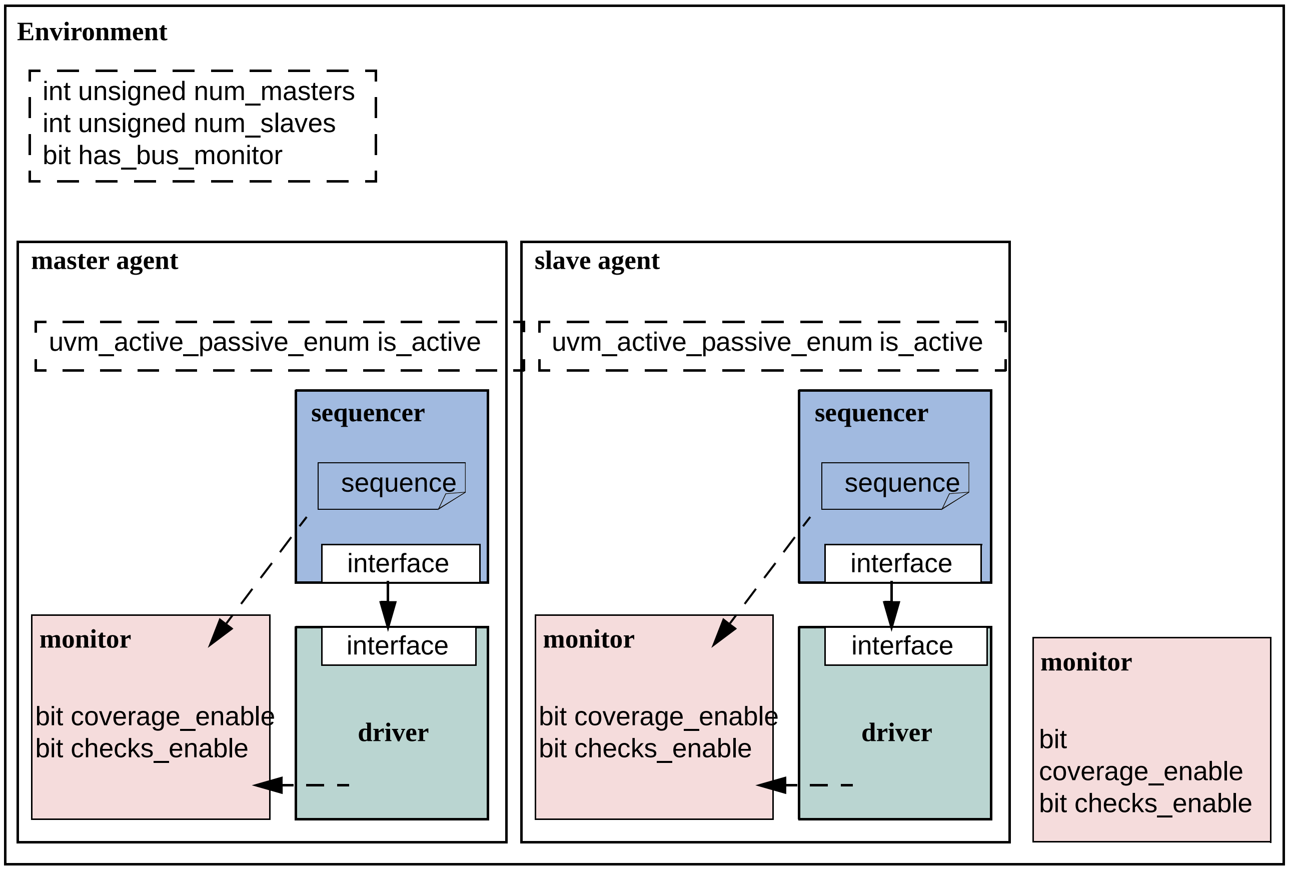


Figure 18—Standard Configuration Fields and Locations

6.3.3 Using a Configuration Class

Some verification components 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.

6.4 Creating and Selecting a User-Defined Test

In UVM, 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.

6.4.1 Creating the Base Test

The following example shows a base test that uses the xbus_demo_tb defined in Section 6.2. 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 uvm_test;
     `uvm_component_utils(xbus_demo_base_test)
     xbus_demo_tb xbus_demo_tb0;
     // The test's constructor
     function new (string name = "xbus_demo_base_test",
       uvm_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
endclass
```

The build() function of the base test creates the xbus_demo_tb. The UVM 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.

All of the definitions in the base test are inherited by any test that derives from xbus_demo_base_test. This means 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.

6.4.2 Creating Tests from a Test-Family Base Class

You can derive from the base test defined in Section 6.4.1 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.) The following is a simple test that inherits from xbus_demo_base_test.

```
class test_read_modify_write extends xbus_demo_base_test;
  `uvm_component_utils(test_read_modify_write)
  // The test's constructor
     function new (string name = "test_read_modify_write",
        uvm_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 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.

6.4.3 Test Selection

After you have declared a user-defined test (described in Section 6.4.2), invoke the global UVM `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 Section 6.4.2) 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 `+UVM_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 hardcode 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 Section 6.4.2) using simulator command-line option, use the following command:

```
% simulator-command other-options +UVM_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 `` `uvm_component_utils`` macro was not used.

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

6.5 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. 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 (see Section 6.5.1).

— Use UVM sequences to control the order of multiple data items. This method provides more flexibility and control (see Section 6.5.2).

6.5.1 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:

a) Identify the data item classes and their generated fields in the verification component.

b) Create a derivation of the data item class that adds or overrides default constraints.

c) In a test, adjust the environment (or a subset of it) to use the newly-defined data items.

d) 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 uvm_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
     `uvm_object_utils_begin(uart_frame)
       `uvm_field_int(start_bit, UVM_ALL_ON)
       `uvm_field_int(payload, UVM_ALL_ON)
       `uvm_field_int(parity, UVM_ALL_ON)
       `uvm_field_enum(parity_e, parity_type, UVM_ALL_ON + UVM_NOCOMPARE)
       `uvm_field_int(xmit_delay,  UVM_ALL_ON + UVM_DEC + UVM_NOCOMPARE)
     `uvm_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.

6.5.1.1 Data Item Definitions

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 verification component 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 in Section 6.5.1), the parity bit is constrained to be 90-percent legal (good parity) and 10-percent illegal (bad parity).

6.5.1.2 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;}
    `uvm_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 verification component frame. The UVM Class Library provides a mechanism that allows you to introduce the derived class to the environment.

```
  class short_delay_test extends uvm_test;
        `uvm_component_utils(short_delay_test)
        uart_tb uart_tb0;
        function new (string name = "short_delay_test",uvm_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();
          uvm_top.print_topology();
        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 UVM 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());
```

6.5.2 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. UVM sequences are library base classes that allow you to create meaningful ordered scenarios. This section describes UVM sequencers and sequences.

6.5.2.1 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 executes a single data item.

Section 6.5.2.2 shows how you can use the configuration mechanism to modify the count to adjust the sequence generated pattern. Subsequent sections introduce 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.

6.5.2.2 Controlling the Number of Sequences Created by uvm_random_sequence

The default number of generated sequences is a random number between `0` and `uvm_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 Section 6.5.2.3.

6.5.2.3 Creating and Adding a New Sequence

To create a user-defined sequence:

a) Derive a sequence from the `uvm_sequence` base class.

b) Use the `` `uvm_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 `` `uvm_object_utils `` macro (and its variations), except 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.

c) 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 `` `uvm_do `` (see Section 5.9.2.2.1) and `` `uvm_do_with `` (see Section 5.9.2.2.2).

Example

The class `retry_seq` in this example a new sequence. It is derived from `uvm_sequence` and uses the `` `uvm_sequence_utils `` macro to associate this sequence with `uart_tx_sequencer` and to declare the various utilities `` `uvm_object_utils `` provides.

```
// Send one BAD_PARITY frame followed by a GOOD_PARITY
    // frame with the same payload.
class retry_seq extends uvm_sequence #(uart_frame);
      rand bit [7:0] pload; // Randomizable sequence parameter
      ...
  // UVM automation for sequences
      'uvm_sequence_utils_begin(retry_seq, uart_tx_sequencer)
        'uvm_field_object(frame, UVM_ALL_ON)
        'uvm_field_int(pload, UVM_ALL_ON)
      'uvm_sequence_utils_end
  // Constructor
      function new(string name="retry_seq");
        super.new(name);
      endfunction
  task body ( ); // Sequence behavior
        'uvm_do_with(req, {req.payload == pload; req.parity == BAD_PARITY;} )
        'uvm_do_with(req, {req.payload == pload; req.parity == GOOD_PARITY;} )
      endtask : body
endclass: retry_seq
```

Sequences can have parameters which can be randomized (e.g., `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 `` `uvm_do_with `` macro randomizes and executes an item with inline constraints. The `` `uvm_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 `` `uvm_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 `` `uvm_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.

6.5.2.4 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 uvm_sequence #(uart_frame);
     // Constructor, and so on
  ...
     `uvm_sequence_utils(rand_retry_rand_seq, uart_tx_sequencer)
     retry_seq retry_sequence; // Variable of a previously declared sequence
  task body (); // Sequence behavior
       `uvm_do (req)
       `uvm_do_with(retry_sequence, {retry_sequence.pload inside {[0:31]};})
       `uvm_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 UVM sequences have.

6.5.2.5 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 (see Figure 19).

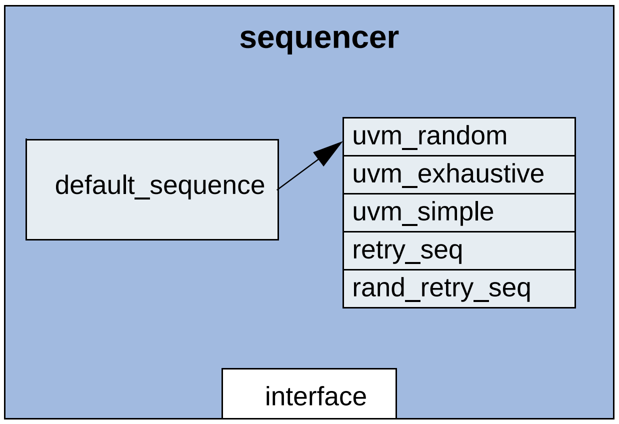
To override the default sequence:

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

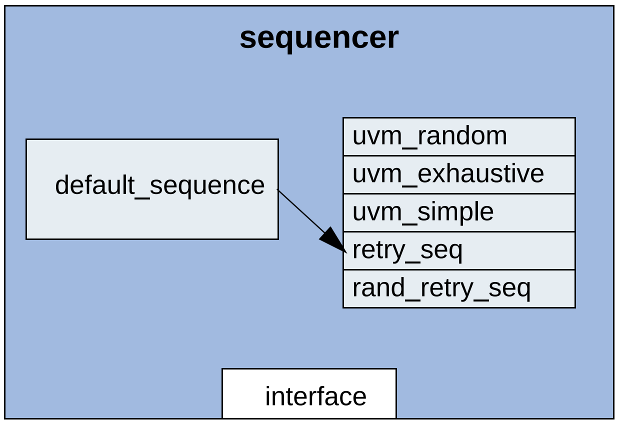
b) Configure the `default_sequence` property for a specific sequencer or a group of sequencers. This is typically done inside the test class, before creating the component that includes the relevant sequencer(s). For example,

```
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`.



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

Figure 19—Sequencer with a Sequence Library

6.5.2.6 Sequence Libraries and Reuse

Use of sequences is an important part of verification component reuse. The environment developer who knows and understands the verification component 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 verification component'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      uvm_random_sequence
sequences         da(string)             4      -
   [0]            string                19      uvm_random_sequence
   [1]            string                23      uvm_exhaustive_sequence
   [2]            string                19      uvm_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 `uvm_random_sequence`, which means sequences will be randomly generated in a loop by default.

This sequencer has five sequences associated with it. Three sequences are built-in sequences (`uvm_random_sequence`, `uvm_exhaustive_sequence`, and `uvm_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 selects and executes once each sequence from the sequencer's sequence library, excluding `uvm_random_sequence` and `uvm_exhaustive_sequence`. If *`count`* equals `0`, the sequencer will not automatically start a sequence. If desired, the user may start a sequence manually. This operation typically is used for virtual sequencers. If *`count`* is not equal to `0`, the sequencer automatically starts the default sequence, which may use the *`count`* variable.

The exhaustive sequence does not use the *`count`* variable. However, the subsequences started by the exhaustive sequence may use *`count`*.

The value of *`count`* in this sequencer is `-1`, which means 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 Section 7.5.

6.5.2.7 Directed-Test Style Interface

The sequence style discussed in Section 6.5.2 is the recommended way to create tests. Focus on creating reusable sequences 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, UVM support this method using the sequencer's `execute_item()` task. Before using directed tests, consider their disadvantages compared to the UVM-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.

```
class directed_test extends xbus_demo_base_test;
     `uvm_component_utils(directed_test)
  xbus_demo_tb xbus_demo_tb0;
     function new (string name = "directed_test",
       uvm_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
```

The following also apply.

a) 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.

b) 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.

c) 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.

6.6 Virtual Sequences

Section 6.5 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 20, the virtual sequencer invokes configuration sequences on the ethernet and cpu verification components. The configuration sequences are developed during block-level testing.

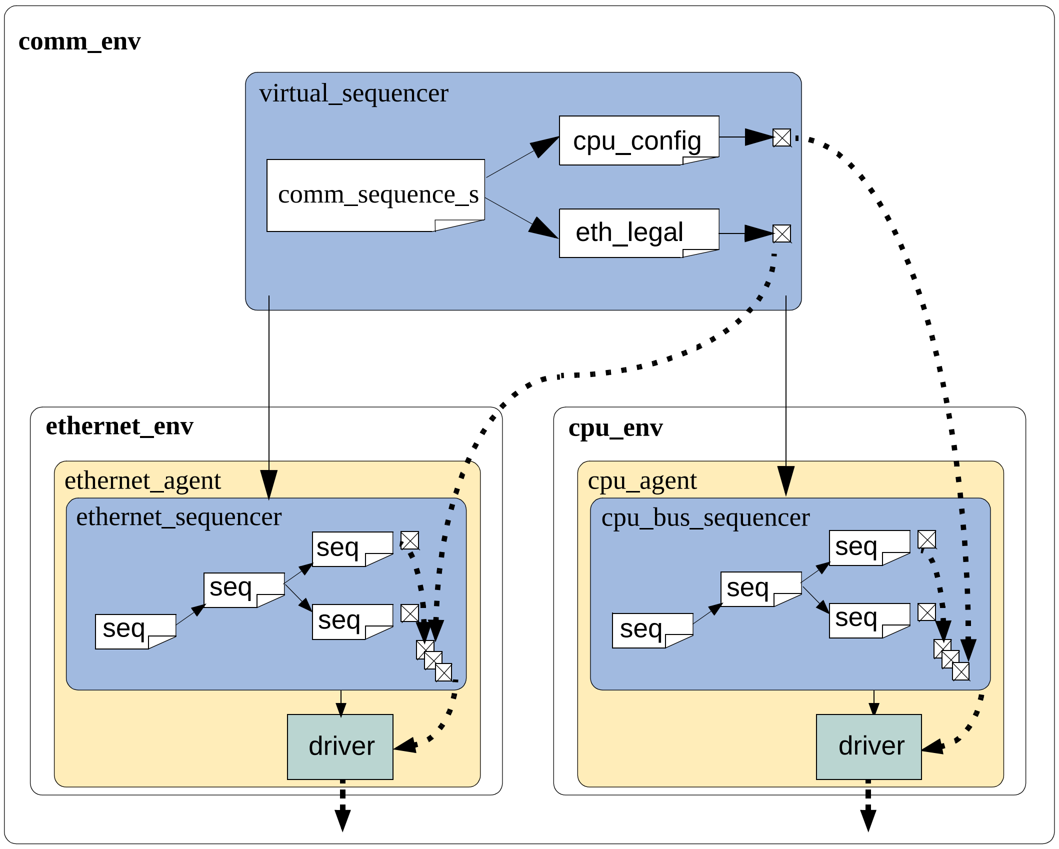


Figure 20—Virtual Sequence

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

d) “Business as usual”—Virtual subsequencers and subsequencers send transactions simultaneously.

e) Disable subsequencers—Virtual sequencer is the only one driving.

f) 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 Section 6.6.3.

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

Use the appropriate `do` macro.

Use the sequence `start()` method.

6.6.1 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:

a) Derive a virtual sequencer class from the `uvm_sequencer` class.

b) Add references to the sequencers where 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 sub-sequencers.

```
class simple_virtual_sequencer extends uvm_sequencer;
     eth_sequencer eth_seqr;
     cpu_sequencer cpu_seqr;
  // Constructor
     function new(input string name="simple_virtual_sequencer",
       input uvm_component parent=null);
       super.new(name, parent);
       // Automation macro for virtual sequencer (no data item)
       `uvm_update_sequence_lib
     endfunction
  // UVM automation macros for sequencers
     `uvm_sequencer_utils(simple_virtual_sequencer)
endclass: simple_virtual_sequencer
```

NOTE—The `` `uvm_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 `uvm_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 `` `uvm_update_sequence_lib_and_item `` macro. See Section 5.4 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 Section 6.6.4.

6.6.2 Creating a Virtual Sequence

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

— A virtual sequence uses `` `uvm_do_on `` or `` `uvm_do_on_with `` to execute sequences on any of the subsequencers connected to the current virtual sequencer.

— A virtual sequence uses `` `uvm_do `` or `` `uvm_do_with `` to execute other virtual sequences of this sequencer. A virtual sequence cannot use `` `uvm_do `` or `` `uvm_do_with `` to execute items. Virtual sequencers do not have items associated with them, only sequences.

To create a virtual sequence:

c) Declare a sequence class by deriving it from `uvm_sequence`, just like a driver sequence.

d) Define a `body()` method that implements the desired logic of the sequence.

e) Use the `` `uvm_do_on `` (or `` `uvm_do_on_with ``) macro to invoke sequences in the underlying subsequencers.

f) Use the `` `uvm_do `` (or `` `uvm_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 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 uvm_sequence;
     ... // Constructor and UVM automation macros
     // See Section 6.5.2.3
     // 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.
       `uvm_do_on(conf_seq, p_sequencer.cpu_seqr)
       // Invoke a sequence in the ethernet subsequencer.
       `uvm_do_on(frame_seq, p_sequencer.eth_seqr)
       // Invoke another virtual sequence in this sequencer.
       `uvm_do(rand_virt_seq)
     endtask : body
endclass : simple_virt_seq
```

6.6.3 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:

a) 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.

b) Disable the subsequencers—Using the `set_config` routines, you can set the count property of the subsequencers to `0` and disable their default behavior. Recall that, by default, sequencers start their `uvm_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 Section 6.6.4.

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

c) Use `grab()` and `ungrab()`—In this case, 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, make sure a given subsequencer is not a virtual sequencer before you attempt to grab it. The following example illustrates this using the functions `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.
      `uvm_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.

6.6.4 Connecting a Virtual Sequencer to Subsequencers

To connect a virtual sequencer to its subsequencers:

a) 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;
```

b) 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 uvm_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 UVM 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
```

6.7 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:

a) Assertions—Derived from the specification or from the implementation and ensure correct timing behavior. Assertions typically focus on signal-level activity.

b) Data checkers—Ensure overall device correctness.

As was mentioned in Section 1.2.4, checking and coverage should be done in the monitor regardless of the driving logic. Reusable assertions are part of reusable components. See Chapter 5 for more information. Designers can also place assertions in the DUT RTL. Refer to your ABV documentation for more information.

6.8 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 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 21.

In this example, the user has created a testbench with 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.

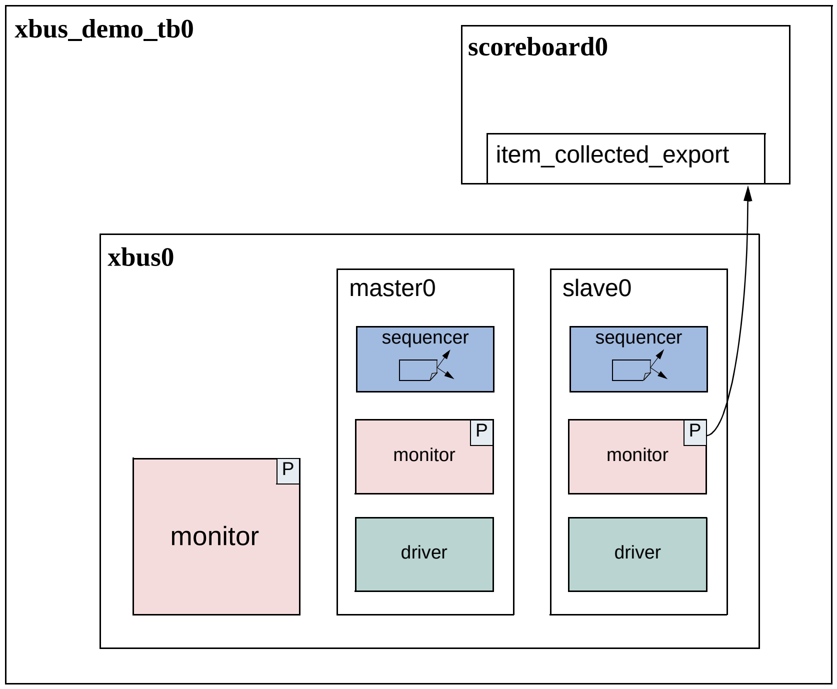


Figure 21—XBus Demo Environment

6.8.1 Creating the Scoreboard

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

To define the scoreboard:

a) Add the TLM export necessary to communicate with the environment monitor(s).

b) Implement the necessary functions and tasks required by the TLM export.

c) Define the action taken when the export is called.

6.8.2 Adding Exports to uvm_scoreboard

In the example shown in Figure 21, 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 `uvm_analysis_port`(s), the scoreboard will provide the TLM `uvm_analysis_imp`.

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

```
class xbus_demo_scoreboard extends uvm_scoreboard;
  uvm_analysis_imp #(xbus_transfer, xbus_demo_scoreboard)
    item_collected_export;
  ...
  function new (string name, uvm_component parent);
    super.new(name, parent);
    item_collected_export = new("item_collected_export", this);
  endfunction : new
  ...
```

Line 2 declares the uvm_analysis_export. The first parameter, xbus_transfer, defines the uvm_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.

6.8.3 Requirements of the TLM Implementation

Since the scoreboard provides an uvm_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.

6.8.4 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 *UVM Class Reference*. As seen in Section 6.8.3, 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 not discussed here. The xbus_demo_scoreboard.sv file shows the implementation.

6.8.5 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 *UVM Class Reference*.

6.8.6 Summary

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

a) Create the scoreboard component.
   1) Add the necessary exports.
   2) Implement the required functions and tasks.
   3) Create the functions necessary to perform the implementation-specific functionality.
b) Add the scoreboard to the environment.
   1) Declare and instantiate the scoreboard component.
   2) Connect the scoreboard implementation(s) to the environment ports of interest.

The XBus demo has a complete scoreboard example. See Chapter 8 for more information.

6.9 Implementing a Coverage Model

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

6.9.1 Selecting a Coverage Method

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

a) 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 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.
b) 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 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 the functional coverage needs to be refined and enhanced.

6.9.2 Implementing a Functional Coverage Model

A verification component should come with a protocol-specific functional-coverage model. 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 verification component coverage and other attributes in the system or other interface verification components. 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.

6.9.3 Enabling and Disabling Coverage

The verification IP developer should provide configuration properties that allow you to control the interesting aspects of the coverage (see Section 5.11.3). 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 SystemVerilog assignment to a class property (or variable).

7. Advanced Topics

This chapter discusses UVM topics and capabilities of the UVM Class Library that are beyond the essential material covered in the previous chapters. Consult this chapter as needed.

7.1 The uvm_component Base Class

All the infrastructure components in an UVM verification environment, including environments and tests, are derived either directly or indirectly from the `uvm_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 `uvm_component`. However, understanding the `uvm_component` is important because many of the facilities that the methodology classes offer are derived from this class.

NOTE—The `uvm_threaded_component` class has been deprecated in UVM and is now simply a `typedef` for `uvm_component`.

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

— Phasing and execution control

— Configuration methods

— Factory convenience methods

— Hierarchical reporting control.

7.2 Simulation Phase Methods

The UVM 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 `uvm_phase` in the *UVM Class Reference* for more information on using built-in phases.

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

a) *build()*

b) *connect()*

c) *end_of_elaboration()*

d) *start_of_simulation()*

e) *run()*

f) *extract()*

g) *check()*

h) *report()*

7.2.1 build()

The first phase of the UVM 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 `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 uvm_component;
     ...
  virtual void function build();
       super.build();
       // Get configuration information.
       // Create child components.
       // configure child components
     endfunction
  ...
endclass
```

7.2.2 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 uvm_component;
     ...
  virtual void function connect();
       if(is_active == UVM_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
```

7.2.3 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 the entire environment is created and connected before this method is called. This phase is a function and executes in zero time.

7.2.4 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.

7.2.5 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 four ways:

a) **stop**—If 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.

b) **kill**—When called, all the component's `run` processes are killed immediately. While `kill` can be called directly, it is recommended components use the stopping mechanism. which affords a more ordered and safe shutdown.

c) **timeout**—If a timeout was set, the phase ends if it expires before either `stop` or `kill` occur.

d) Any `uvm_test_done` objections that have been raised are dropped (the objection count goes to `0`).

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

— 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 automatically handled by the UVM Class Library.

— 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.

7.2.6 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. The 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.

7.2.7 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.

7.2.8 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.

7.2.9 Adding User-Defined Phases

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

To define a new phase:

a) Derive a subclass of `uvm_phase` that implements 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 uvm_component;
2    ...
3    virtual my_task();  return; endtask // make virtual
4    ...
5  endclass
6
7  class my_task_phase extends uvm_phase;
8    function new();
9      super.new("my_task",1,1);
10   endfunction
11   task call_task(uvm_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:

1) The name of the phase, which is typically the name of the callback method.

2) A bit to indicate whether the method is to be called top-down (`1`) or bottom-up (`0`).

3) A bit to indicate whether the method is a task (`1`) or a function (`0`).

NOTE—UVM includes several macros to simplify the definition of new phases:

```
`define uvm_phase_task_decl(NAME,TOP_DOWN)
`define uvm_phase_func_topdown_decl(NAME) `uvm_phase_func_decl(NAME,1)
`define uvm_phase_func_bottomup_decl(NAME) `uvm_phase_func_decl(NAME,0)
`define uvm_phase_task_topdown_decl(NAME) `uvm_phase_task_decl(NAME,1)
`define uvm_phase_task_bottomup_decl(NAME) `uvm_phase_task_decl(NAME,0)
```

b) Declare an instance of the new phase object:

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

c) Register the phase with the UVM phase controller, `uvm_top`.

```
uvm_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`.

7.3 The Built-In Factory and Overrides

7.3.1 About the Factory

UVM 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, uvm_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`.

7.3.2 Factory Registration

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

— Use the `` `uvm_object_utils(T) `` or `` `uvm_component_utils(T) `` macro in a derivative `uvm_object` or `uvm_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

```
`uvm_object_utils(packet)
'uvm_component_utils(my_driver)
```

— Use the `` `uvm_object_registry(T,S) `` or `` `uvm_component_registry(T,S) `` registration macros. 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 `uvm_*_utils` macros, so you might use them only if you do not use the `uvm_*_utils` macros.

7.3.3 Component Overrides

A global factory 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. 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:

a) Define a class that derives from the appropriate UVM base class.
b) Execute the override (described in the following sections).
c) Build the environment.

7.3.3.1 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 22.

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`

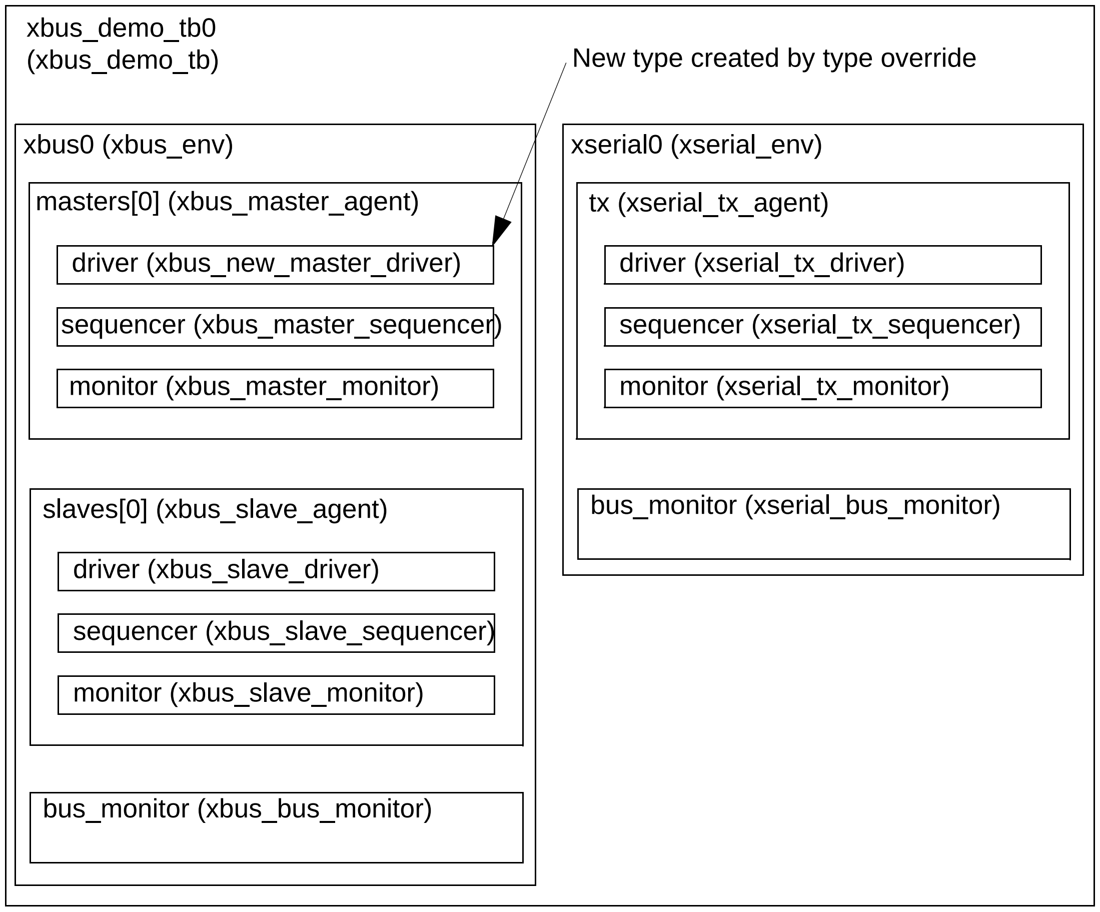

xbus_demo_tb0
(xbus_demo_tb)
New type created by type override
xbus0 (xbus_env)
masters[0] (xbus_master_agent)
driver (xbus_new_master_driver)
sequencer (xbus_master_sequencer)
monitor (xbus_master_monitor)
slaves[0] (xbus_slave_agent)
driver (xbus_slave_driver)
sequencer (xbus_slave_sequencer)
monitor (xbus_slave_monitor)
bus_monitor (xbus_bus_monitor)
xserial0 (xserial_env)
tx (xserial_tx_agent)
driver (xserial_tx_driver)
sequencer (xserial_tx_sequencer)
monitor (xserial_tx_monitor)
bus_monitor (xserial_bus_monitor)

Figure 22—Hierarchy Created with set_type_override() Applied

7.3.3.2 Instance Overrides

The second component override replaces targeted components of the matching instance path with the new specified type. The prototype for `uvm_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 23. For illustration purposes, this hierarchy assumes both overrides have been executed.

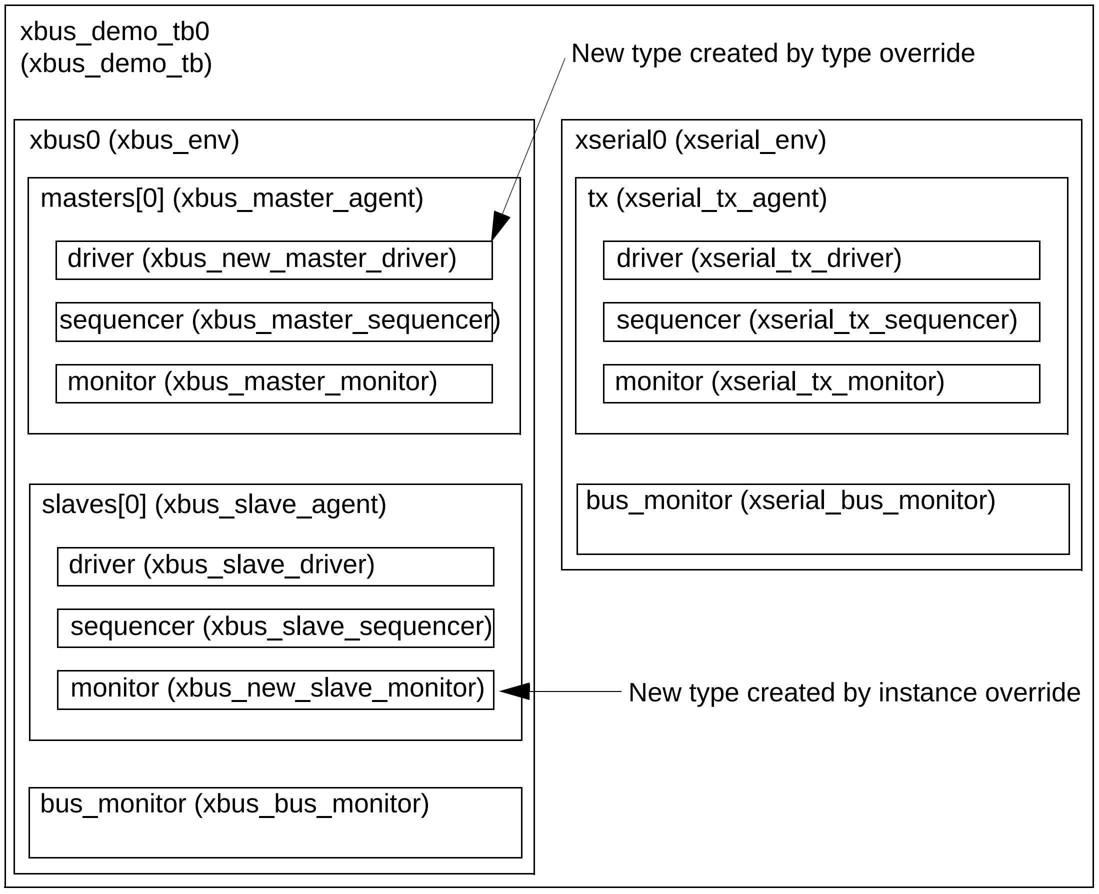

xbus_demo_tb0
(xbus_demo_tb)
New type created by type override
xbus0 (xbus_env)
masters[0] (xbus_master_agent)
driver (xbus_new_master_driver)
sequencer (xbus_master_sequencer)
monitor (xbus_master_monitor)
slaves[0] (xbus_slave_agent)
driver (xbus_slave_driver)
sequencer (xbus_slave_sequencer)
monitor (xbus_new_slave_monitor)
bus_monitor (xbus_bus_monitor)
xserial0 (xserial_env)
tx (xserial_tx_agent)
driver (xserial_tx_driver)
sequencer (xserial_tx_sequencer)
monitor (xserial_tx_monitor)
bus_monitor (xserial_bus_monitor)
New type created by instance override

Figure 23—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());
```

7.4 Callbacks

Callbacks are an optional facility end users can use to augment component behavior

7.4.1 Use Model

To provide a callback facility to end-users, the component developer needs to:

a) Derive a callback class from the `uvm_callback` base. It should declare one or more methods that comprise the “callback interface”.

b) Optionally, define a `typedef` to the `uvm_callbacks` pool typed to our specific component-callback combination.

c) Define the component to support the callback class defined in Step (a) by defining virtual methods corresponding to each of the methods in the callback interface. Implement each method to execute the corresponding method in all of the registered callbacks using a default algorithm (for example, sequential, concurrent, random, and so on). Invoke each virtual method at the desired location within a component main body of code, typically its `run` task.

To use callbacks, the user needs to:

d) Define a new callback class extending from the callback base class provided by the developer, overriding one or more of the available callback methods.

e) Register one or more instances of the callback with the component(s) you wish to extend.

These steps are illustrated in the following simple example.

7.4.2 Example

The example below demonstrates callback usage. The component developer defines a driver component and a driver-specific callback class. The callback class defines the hooks available for users to override. The component using the callbacks (that is, calling the callback methods) also defines corresponding virtual methods for each callback hook. The developer implements each virtual methods to call the corresponding callback method in all registered callback objects using default algorithm. The end-user may then define either a callback or a driver subtype to extend driver’s behavior.

7.4.2.1 Developer Code

a) Define a callback class extending from `uvm_callback`.

The callback class defines an application-specific interface consisting of one or more function or task prototypes. The signatures of each method have no restrictions.

In the example below, a new `bus_bfm_cb` class extending from `uvm_callback` is defined. The developer of the `bus_bfm` component decides to add two hooks for users, `trans_received` and `trans_executed`:

1) `trans_received`—the bus driver calls this after it first receives a new transaction item. It provides a handle to both itself and the new transaction. The return value determines whether to drop (`1`) or execute (`0`) the transaction.
2) `trans_executed`—the bus driver calls this after executing the transaction, passing in a handle to itself and the transaction, which may contain read data or other status information.

```
virtual class bus_bfm_cb extends uvm_callback;
  virtual function bit trans_received(bus_bfm driver, bus_tr tr);
        return 0;
      endfunction
  virtual task trans_executed(bus_bfm driver, bus_tr tr);
      endtask
  function new(string name="bus_bfm_cb_inst");
        super.new(name);
      endfunction
endclass
```

b) Define a `typedef` to the `uvm_callbacks` pool typed to our specific component-callback combination.

   UVM callbacks are type-safe, meaning any attempt to register a callback to a component not designed for that callback simply will not compile. In exchange for this type-safety we must endure a bit of parameterized syntax as follows:

```
typedef uvm_callbacks #(bus_bfm, bus_bfm_cb) bd_cb;
```

   The alias `bd_cb` can help both the component developer and the end-user produce more readable code.

c) Embed the callback feature in the component that will use it.

An important aspect of adding support for callbacks is to define virtual methods in the component that correspond to each of the methods in the callback interface defined in Step (a). The definition for each of these virtual methods should implement the algorithm that traverses and executes the potentially multiple callback objects registered with the component. The algorithm may be to execute them sequentially, concurrently in separate processes, or to aggregate return values. Encapsulating the algorithm in a virtual method allows the end-user to override it, disable it, change the default execution order, or add a pre- and post-callback logic.

The developer of the `bus_bfm` adds the `trans_received` and `trans_executed` virtual methods, with their default implementations utilizing some macros that implement the most common algorithms for executing all registered callbacks. With this in place, end-users can now customize component behavior in two ways:

— extend `bus_bfm` and override one or more of the virtual methods `trans_received` or `trans_executed`. Then configure the factory to use the new type via a type or instance override.

— extend `bus_bfm_cb` and override one or more of the virtual methods `trans_received` or `trans_executed`. Then register an instance of the new callback type with an instance of `bus_driver`. This of course requires access to the handle of the `bus_bfm`.

```
class bus_bfm extends uvm_component;
  uvm_blocking_put_imp #(bus_tr,bus_bfm) in;
     function new (string name, uvm_component parent=null);
       super.new(name,parent);
       in = new("in",this);
     endfunction
  `uvm_register_cb(bus_bfm, bus_bfm_cb);
  virtual function bit trans_received(bus_tr tr);
       `uvm_do_callbacks_exit_on(bus_bfm_cb,bus_bfm,
          trans_received(this,tr),1)
     endfunction
  virtual task trans_executed(bus_tr tr);
      `uvm_do_callbacks(bus_bfm_cb,bus_bfm,trans_executed(this,tr))
     endtask
  virtual task put(bus_tr t);
  uvm_report_info("bus_tr received",t.convert2string());
       if (!trans_received(t)) begin
          uvm_report_info("bus_tr dropped",
            "user callback indicated DROPPED\n");
       return;
     end
    #100;
    trans_executed(t);
    uvm_report_info("bus_tr executed",{t.convert2string(),"\n"});
     endtask
endclass
```

The driver's `put` task, which implements the component's primary functionality, merely calls the virtual methods at the appropriate times during execution.

7.4.2.2 End User Code

Using the callback feature of a component involves the following steps:

a) Extend the developer-supplied callback class.

   Define a new callback class that extends from the class provided by the component developer, implementing any or all of the methods of the callback interface.

   In our example, we define both hooks, `trans_received` and `trans_executed`. For `trans_received`, we randomly choose whether to return `0` or `1`. When `1`, the `bus_driver` will "drop" the received transaction. For `trans_executed`, we delay `#10` to prevent back-to-back transactions.

```
class my_bus_bfm_cb extends bus_bfm_cb;
  function new(string name="bus_bfm_cb_inst");
       super.new(name);
     endfunction
  `uvm_object_utils(my_bus_bfm_cb)
  virtual function bit trans_received(bus_bfm driver, bus_tr tr);
       driver.uvm_report_info("trans_received_cb",
         {" bus_bfm=",driver.get_full_name(),"
          tr=",tr.convert2string()});
     return $urandom & 1;
     endfunction
  virtual task trans_executed(bus_bfm driver, bus_tr tr);
       driver.uvm_report_info("trans_executed_cb",
         {" bus_bfm=",driver.get_full_name(),"
```

```
          tr=",tr.convert2string()});
        #10;
      endtask
endclass
```

b) Create callback object(s) and register with component you wish to extend.

   To keep the example simple and focus on callback usage, we do not show a complete or compliant UVM environment.

   In the top module, we instantiate the `bus_bfm` and an instance of our custom callback class. To register the callback object with the driver, we first get a handle to the global callback pool for our specific driver-callback combination. Luckily, the developer provided a convenient `typedef` in his Step (b) that makes our code a little more readable.

   Then, we associate (register) the callback object with a driver using the callback pool's `add_cb` method. After calling `display_cbs` to show the registration was successful, we push several transactions into the driver. The output shows that the methods in our custom callback implementation are called for each transaction the driver receives.

```
module top;
  bus_tr           tr     = new;
  bus_bfm          driver = new("driver");
  my_bus_bfm_cb    cb     = new("cb");

      initial begin
        bd_cb::add(driver,cb);
        cbs.display_cbs();
        for (int i=1; i<=5; i++) begin
          tr.addr = i;
          tr.data = 6-i;
          driver.in.put(tr);
        end
      end
endmodule
```

c) Instance-specific callback registrations can only be performed after the component instance exists. Therefore, those are typically done in the `build()` and `end_of_elaboration()` for extensions that need to apply for the entire duration of the test and in the `run()` method for extensions that need to apply for a specific portion of the testcase.

```
class error_test extends uvm_test;
   function new(name = “error_test”, uvm_component parent = null);
      super.new(name, parent);
   endfunction

   virtual task run();
      cbs = new;
      #1000;
      bd_cb::add_by_name(cb, “top.bfm”);
      #100;
      bd_cb::delete(cb);
   endfunction
endclass
```

7.5 Advanced Sequence Control

This section discusses advanced techniques for sequence control.

7.5.1 Implementing Complex Scenarios

This section highlights how to implement various complex scenarios.

7.5.1.1 Executing Multiple Sequences Concurrently

There are two ways you can create concurrently-executing sequences: the following subsections show an example of each method.

7.5.1.1.1 Using the uvm_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 uvm_sequence #(simple_item);
     ... // Constructor and UVM automation macros go here.
         // See Section 6.5.2.3
     a_seq a;
     b_seq b;
  virtual task body();
       fork
         `uvm_do(a)
         `uvm_do(b)
       join
     endtask : body
   endclass : fork_join_sequence
```

7.5.1.1.2 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 uvm_sequence #(simple_item);
     ... // Constructor and UVM automation macros go here.
         // See Section 6.5.2.3
     a_seq a;
     b_seq b;
  virtual task body();
       // Initialize the sequence variables with the factory.
       `uvm_create(a)
       `uvm_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 `uvm_create` in the *UVM Class Reference* for additional information.

7.5.1.1.3 Using the pre_body() and post_body() Callbacks

The UVM 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 of 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 UVM automation macros go here.
         // See Section 6.5.2.3
  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 `` `uvm_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 `` `uvm_sequence_utils`` macro was used. This prevents the user from having to declare a variable of the appropriate type and initialize it using `$cast`.

7.5.1.2 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:

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

b) 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

Define an interrupt handler sequence.

```
// Upon an interrupt, grab the sequencer, and execute a
    // read_status_seq sequence.
    class interrupt_handler_seq extends uvm_sequence #(bus_transfer);
      ... // Constructor and UVM automation macros here
        // See Section 6.5.2.3
  read_status_seq interrupt_clear_seq;
      virtual task body();
        forever begin
          // Initialize the sequence variables with the factory.
          @p_sequencer.interrupt;
          grab(p_sequencer);
          `uvm_do(interrupt_clear_seq)
          ungrab(p_sequencer);
        end
      endtask : body
    endclass : interrupt_handler_seq
```

Then, 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 uvm_sequencer;
      ... // Constructor and UVM automation macros here
          // See Section 6.5.2.3
  interrupt_handler_seq interrupt_seq;
  virtual task run();
        interrupt_seq =
           interrupt_handler_seq::type_id::create("interrupt_seq");
        fork
          interrupt_seq.start(this);
        join_none
        super.run();
      endtask : run
    endclass : my_sequncer
```

NOTE—In this step, we cannot use any of the `` `uvm_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.

7.5.1.3 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:

a) Another `do` action is added to the queue.

b) A new sequence grabs the sequencer, or the current grabber ungrabs the sequencer.

c) Any one of the blocked sequence's `wait_for_relevant()` task returns. See Section 7.5.1.4 for more information.

When calling `try_next_item()`, if the sequencer does not succeed in choosing a `do` action before the time specified in `uvm_driver::wait_for_sequences()`, `uvm_driver::try_next_item()` returns with `NULL`.

7.5.1.4 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 Section 7.5.1.3.

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 uvm_sequence #(bus_transfer);
     ... // Constructor and UVM automation macros go here.
         // See Section 6.5.2.3
  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) `uvm_do(frame)
     endtask : send_frames
  virtual task body();
       fork
         monitor_credits();
         send_frames();
       join_any
     endtask : body
   endclass : flow_control_seq
```

7.5.2 Protocol Layering

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

7.5.2.1 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.

7.5.2.2 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 there are two sequencers. The low-layer sequencer drives `lower_layer_items`, defined as:

```
class lower_layer_item extends uvm_sequence_item;
     ... // Constructor and UVM automation macros go here.
         // See Section 6.5.2.3.
     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 uvm_sequence #(lower_layer_item);
     ... // Constructor and UVM automation macros go here.
         // See Section 6.5.2
     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 `lower_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 uvm_sequence #(higher_layer_item);
     ... // Constructor and UVM automation macros
         // See Section 6.5.2
     higher_layer_item item;
     virtual task body();
       ...
     endtask : body
   endclass : higher_layer_seq_base
```

7.5.2.3 Layering and Sequences

Layering is best implemented with sequences. There are two ways to do layering using sequences: the following subsections show an example of each method.

7.5.2.3.1 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 UVM automation macros go here.
         // See Section 6.5.2
  higher_layer_item hli;
      lower_layer_item lli;
  task body();
        // Create a higher-level item.
        `uvm_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.
          `uvm_create(lli);
          ... // Slice and dice hli to form property values of lli.
          `uvm_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 `uvm_create` in the *UVM Class Reference* for more information.

7.5.2.3.2 Layering of Several Sequencers

This general approach to layering several sequencers uses multiple sequencers as shown in Figure 24.

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 a verification component so that layering can be done by connecting the verification components.

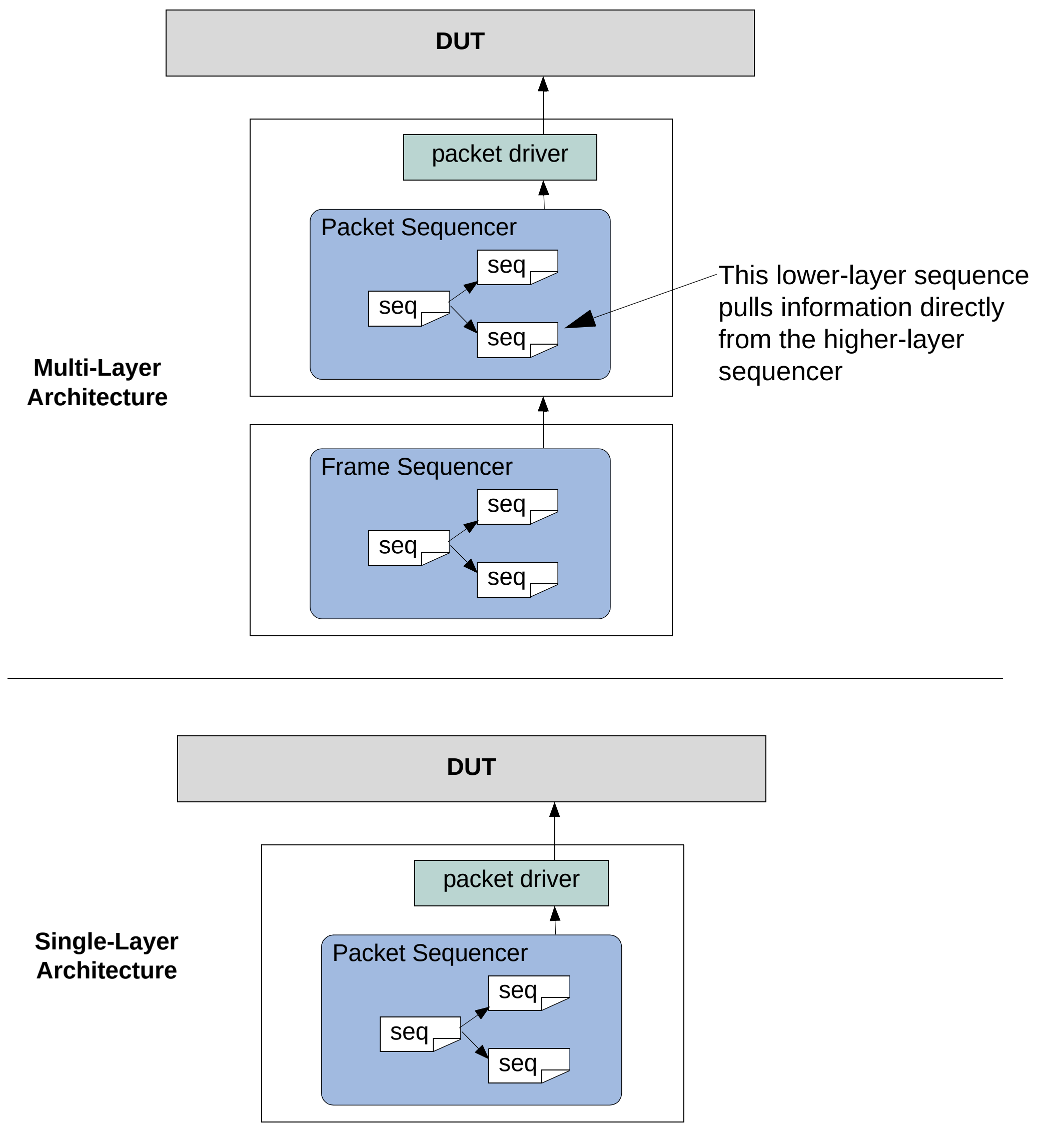


Figure 24—Layering Architecture

7.5.2.4 Styles of Layering

This section explores the various layering styles.

7.5.2.4.1 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 (see Figure 25).

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

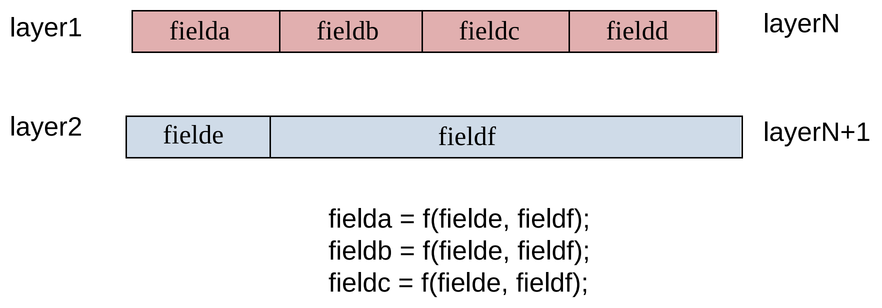

layer1
fielda
fieldb
fieldc
fieldd
layerN
layer2
fielde
fieldf
layerN+1
fielda = f(fielde, fieldf);
fieldb = f(fielde, fieldf);
fieldc = f(fielde, fieldf);

Figure 25—Protocol Layering

7.5.2.4.2 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 26):

a) One-to-one—One high-layer item must be converted into one low-layer item.

b) One-to-many—One large high-layer item must be broken into many low-layer items.

c) Many-to-one—Many high-layer items must be combined into one large low-layer item (as in Sonet, for example).

d) Many-to-many—Multiple higher-layer items must be taken in and converted into multiple lower-layer items. For example, high-layer packets are 10-bytes long, and low-layer packets are 3- to 35-bytes long. In this case, there could be remainders.

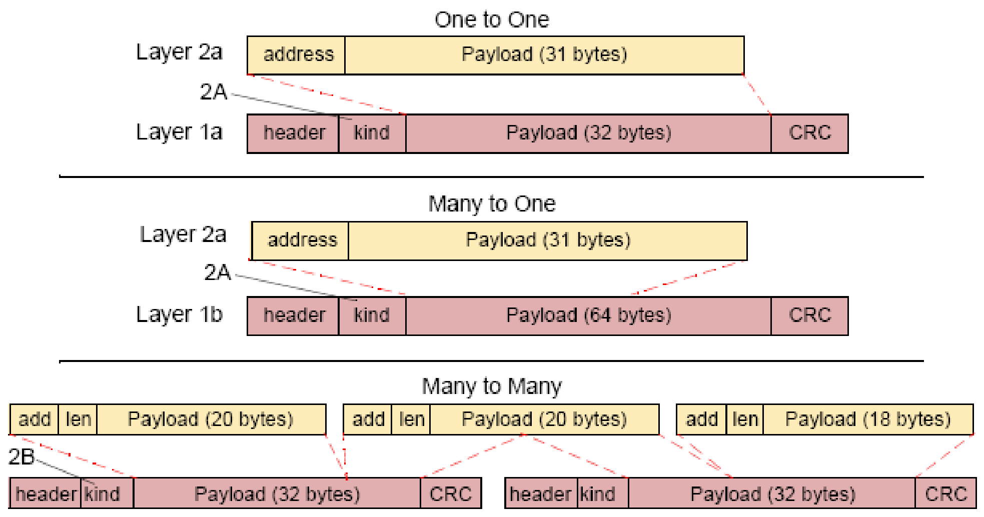


Figure 26—Layer Mapping

7.5.2.4.3 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 `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.

7.5.2.4.4 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. `uvm_driver:try_next_item()` would be used by the lower-level driver in this case.

7.5.2.4.5 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.

7.5.2.4.6 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 27, a low-layer sequencer (`L1B`) gets input from multiple high-layer sequencers (two instances of `L2A`), as well as from a controlling sequencer.

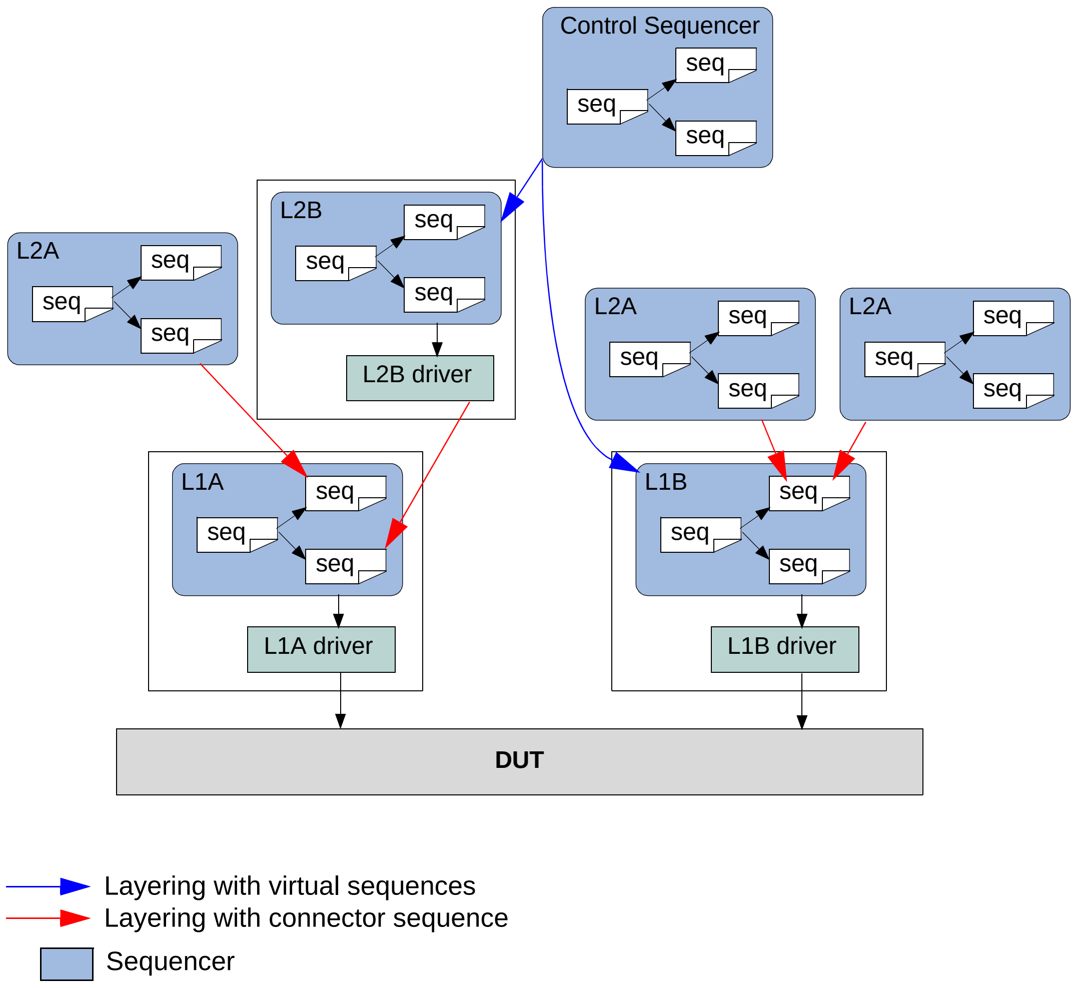


Figure 27—Most-General Case of using Virtual Sequencers

7.5.2.5 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, declare a corresponding `uvm_seq_item_pull_port` in the lower-layer sequencer (see Section 7.5.2.6. 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 to create the layering:

a) Create a lower-layer sequence which does the following:

    1) Repeatedly pulls upper-layer items from the upper-layer sequencer.

    2) Translates them to lower-layer items.

    3) 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 the upper-layer item will be randomized only when the lower-layer driver is ready to start processing the matching lower-layer items.

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

c) Configure the lower-layer sequencer's default sequence to be the sequence you created in Step (a).

7.5.2.6 Layered Sequencers Examples

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 uvm_sequence_item;
      ...
    endclass : upper_item
class upper_sequencer extends uvm_sequencer #(upper_item);
      ...
    endclass : upper_sequencer
// Lower-layer classes
    class lower_item extends uvm_sequence_item;
      ...
    endclass : lower_item
class lower_sequencer extends uvm_sequencer #(lower_item);
      uvm_seq_item_pull_port #(upper_item) upper_seq_item_port;
      ...
      function new (string name, uvm_component parent);
        super.new(name, parent);
        upper_seq_item_port = new("upper_seq_item_port",this);
        `uvm_update_sequence_lib_and_item(...)
      endfunction : new
      ...
    endclass : lower_sequencer
class lower_driver extends uvm_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 uvm_sequence #(lower_item);
      ... // Constructor and UVM automation macros go here.
          // See Section 6.5.2
      upper_item u_item;
      lower_item l_item;
  virtual task body();
        forever begin
          `uvm_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(uvm_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 verification component; 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`, whereas 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();
      super.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
```

7.5.3 Generating the Item or Sequence in Advance

The various `` `uvm_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 UVM Class Library provides additional macros that enable finer control of these various steps. This section describes these macros.

7.5.3.1 `uvm_create

This macro allocates an object using the common factory and initializes its properties. Its argument is a variable of type `uvm_sequence_item` or `uvm_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 `uvm_create(item0) call. After the macro call, the rand_mode() and constraint_mode() functions are used and some direct assignments to properties of item0 occur. 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 uvm_sequence #(my_item);
      ... // Constructor and UVM automation macros go here.
          // See Section 6.5.2.3
      virtual task body();
        `uvm_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 `uvm_create.

NOTE—You might need to disable a constraint to avoid a conflict.

7.5.3.2 `uvm_send

This macro processes the uvm_sequence_item or uvm_sequence class handle argument as shown in Figure 15 and Figure 16, 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 uvm_create() to pre-allocate a sequence item along with `uvm_send, which processes it as shown in Figure 15, without allocation or randomization.

```
class my_seq2 extends uvm_sequence #(my_item);
      ... // Constructor and UVM automation macros go here.
          // See Section 6.5.2.3
      virtual task body();
        `uvm_create(req)
        req.addr = 27;
        req.data = 4;
        // No randomization. Use a purely pre-generated item.
        `uvm_send(req)
      endtask : body
    endclass: my_seq2
```

Similarly, a sequence variable could be provided to the `uvm_create and `uvm_send calls above, in which case the sequence would be processed in the manner shown in Figure 16, without allocation or randomization.

7.5.3.3 `uvm_rand_send, `uvm_rand_send_with

These macros are identical to `uvm_send (see Section 7.5.3.2), 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. `uvm_rand_send() takes just the object handle. `uvm_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 `` `uvm_create `` to pre-allocate a sequence item along with the `` `uvm_rand_send* `` macros, which process it as shown in Figure 15, 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 uvm_sequence #(my_item);
      ... // Constructor and UVM automation macros go here.
          // See Section 6.5.2.3
      virtual task body();
        `uvm_create(req)
        req.addr.rand_mode(0);
        req.dc1.constraint_mode(0);
        req.addr = 27;
        // Randomize and process the item.
        `uvm_rand_send(req)
     // Randomize and process again, this time with inline constraints.
        `uvm_rand_send_with(req, {data < 1000;})
      endtask : body
    endclass: my_seq3
```

7.5.4 Executing Sequences and Items on other Sequencers

In the preceding sections, all `uvm_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.

‘uvm_do_on, **‘uvm_do_on_with**, **‘uvm_do_on_pri**, and **‘uvm_do_on_pri_with**

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

```
‘uvm_do_on(s_seq, that_sequencer);
‘uvm_do_on_with(s_seq, that_sequencer, {s_seq.foo == 32’h3;})
```

8. XBus Verification Component Example

This chapter introduces the basic architecture of the XBus verification component. 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 verification component architecture. When developing your own simulation environment, you should follow the XBus structure and not its protocol-specific functionality.

All XBus verification component subcomponents inherit from some base class in the UVM Class Library, so make sure you have the *UVM 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.

You should also familiarize yourself with the XBus specification in Chapter 9. While not a prerequisite, understanding the XBus protocol will help you distinguish XBus protocol-specific features from verification component protocol-independent architecture.

8.1 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 and the XBus master sequence `read_modify_write` 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 UVM 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 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 `UVM_VERBOSITY = UVM_LOW`:

```
UVM_INFO @ 0: reporter [RNTST] Running test test_read_modify_write...
   UVM_INFO @ 0: uvm_test_top [test_read_modify_write] Printing the test
   topology :
----------------------------------------------------------------------
Name                      Type                Size                Value
----------------------------------------------------------------------
uvm_test_top              test_read_modify_w+ -                    @727
  xbus_demo_tb0           xbus_demo_tb        -                    @841
    scoreboard0           xbus_demo_scoreboa+ -                    @942
      item_collected_ex+  uvm_uvm_analysis_imp -                   @1083
      disable_scoreboard  integral            1                     'h0
      num_writes          integral            32                    'd0
      num_init_reads      integral            32                    'd0
      num_uninit_reads    integral            32                    'd0
      recording_detail    uvm_verbosity       32                UVM_FULL
    xbus0                 xbus_env            -                    @843
      bus_monitor         xbus_bus_monitor    -                    @1015
      masters[0]          xbus_master_agent   -                    @1150
      slaves[0]           xbus_slave_agent    -                    @1231
      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   uvm_verbosity       32               UVM_FULL
    recording_detail     uvm_verbosity       32               UVM_FULL
----------------------------------------------------------------------
UVM_INFO @ 110: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard]
   READ to empty address...Updating address : 12 with data : 4c
UVM_INFO @ 110: uvm_test_top.xbus_demo_tb0.xbus0.bus_monitor
   [xbus_bus_monitor]
   Transfer collected :
----------------------------------------------------------------------
Name                     Type                Size                Value
----------------------------------------------------------------------
xbus_transfer_inst       xbus_transfer       -                   @1217
  addr                   integral            16                'h12
  read_write             xbus_read_write_en+ 32                   READ
  size                   integral            32                    'h1
  data                   da(integral)        1                       -
    [0]                  integral            8                    'h4c
  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                     70
  end_time               time                64                    110
----------------------------------------------------------------------
UVM_INFO @ 260: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard]
   WRITE to existing address...Updating address : 12 with data : 4d
UVM_INFO @ 260: uvm_test_top.xbus_demo_tb0.xbus0.bus_monitor
   [xbus_bus_monitor]
   Transfer collected :
----------------------------------------------------------------------
Name                     Type                Size                Value
----------------------------------------------------------------------
xbus_transfer_inst       xbus_transfer       -                   @1217
  addr                   integral            16                'h12
  read_write             xbus_read_write_en+ 32                  WRITE
  size                   integral            32                    'h1
  data                   da(integral)        1                       -
    [0]                  integral            8                    'h4d
  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                    220
  end_time               time                64                    260
----------------------------------------------------------------------
UVM_INFO @ 330: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard]
   READ to existing address...Checking address : 12 with data : 4d
UVM_INFO @ 330: uvm_test_top.xbus_demo_tb0.xbus0.bus_monitor
   [xbus_bus_monitor]
   Transfer collected :
----------------------------------------------------------------------
Name                     Type                Size                Value
----------------------------------------------------------------------
xbus_transfer_inst       xbus_transfer       -                   @1217
  addr                   integral            16                'h12
  read_write             xbus_read_write_en+ 32                   READ
  size                   integral            32                    'h1
  data                     da(integral)        1                   -
    [0]                    integral            8                'h4d
  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                290
  end_time                 time                64                330
----------------------------------------------------------------------
UVM_INFO @ 380: uvm_test_done [TEST_DONE] All end-of-test objections have been
    dropped, calling global_stop_request()
UVM_INFO @ 380: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard]
    Reporting scoreboard information...

----------------------------------------------------------------------
Name                       Type                Size                Value
----------------------------------------------------------------------
scoreboard0                xbus_demo_scoreboa+ -                    @942
  item_collected_export    uvm_connector_base  -                   @1083
    recording_detail       uvm_verbosity       32               UVM_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         uvm_verbosity       32               UVM_FULL
----------------------------------------------------------------------
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   10
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]      1
[TEST_DONE]      1
[test_read_modify_write]      1
[xbus_bus_monitor]      3
[xbus_demo_scoreboard]      4
Simulation complete via $finish(1) at time 380 NS + 7
```

8.2 XBus Demo Architecture

Figure 28 shows the testbench topology of the XBus simulation environment in the XBus demo example delivered with this release.

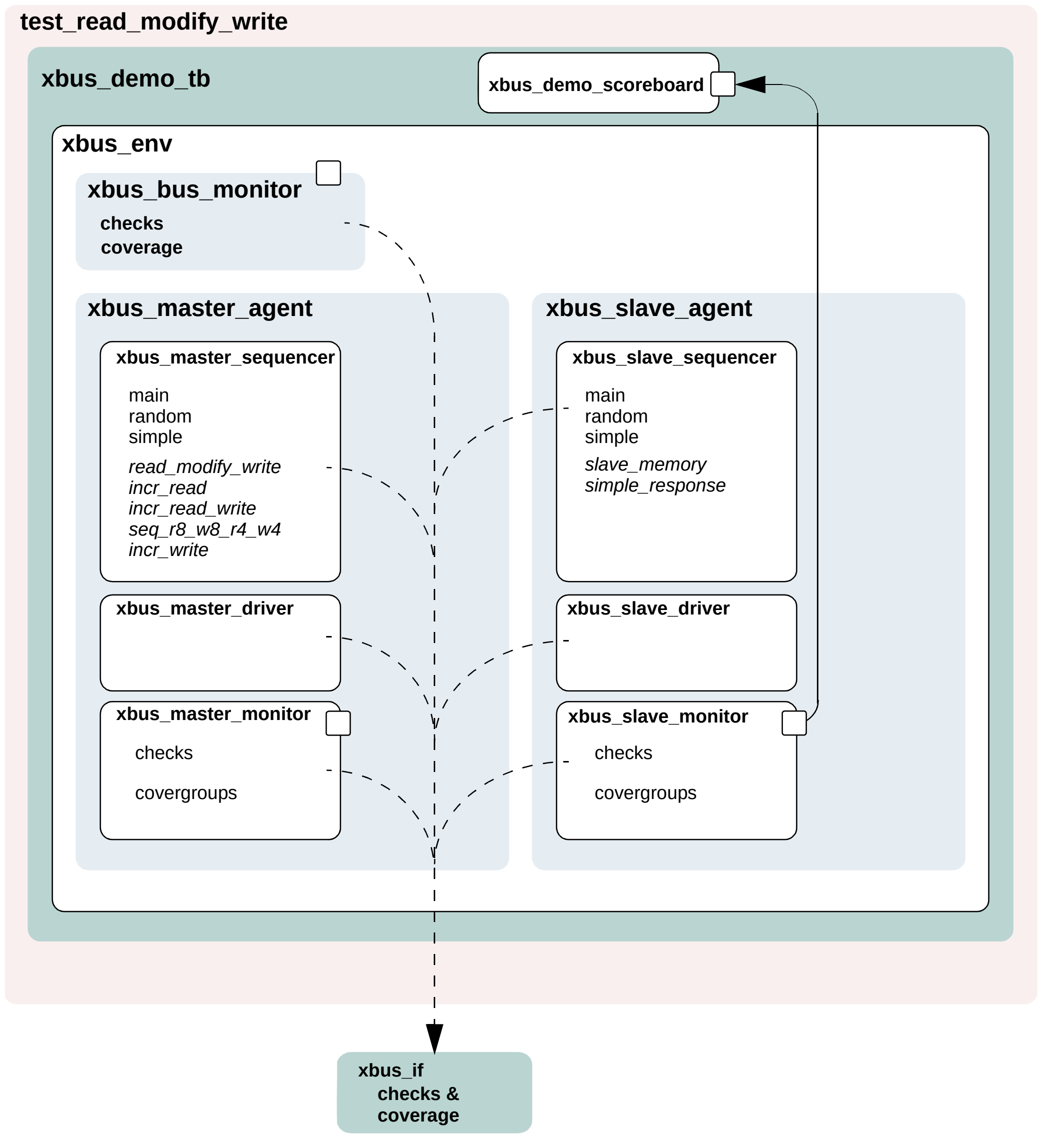


Figure 28—XBus Demo Architecture

8.3 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 intentionally trivial so the focus is on the XBus verification component 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: 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 Section 6.7 and Section 8.12 for more information.

8.4 The Test

In UVM, the test is defined in a separate class, `test_read_modify_write`. It derives from `xbus_demo_base_test` that, in turn, derives from `uvm_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 `` `uvm_component_utils `` macros are registered with a common factory, `uvm_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 +UVM_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 Section 6.4 for more information.

Example: test_lib.sv

```
1  `include "xbus_demo_tb.sv"
2
3  class xbus_demo_base_test extends uvm_test;
4
5    `uvm_component_utils(xbus_demo_base_test)
6
7    xbus_demo_tb xbus_demo_tb0; // XBus verification environment
8    uvm_table_printer printer;
9
10   function new(string name = "xbus_demo_base_test",
11     uvm_component parent=null);
12     super.new(name, parent);
13   endfunction
14   // UVM build() phase
15   virtual function void build();
16     super.build();
17     // Enable transaction recording.
18     set_config_int("*", "recording_detail", UVM_FULL);
19     // Create the testbench.
20     xbus_demo_tb0 = xbus_demo_tb::type_id::create("xbus_demo_tb0",
       this);
21     // Create a specific-depth printer for printing the topology.
22     printer = new();
23     printer.knobs.depth = 3;
24   endfunction
25    // Built-in UVM phase
26   function void end_of_elaboration();
27     // Set verbosity for the bus monitor.
28     xbus_demo_tb0.xbus0.bus_monitor.set_report_verbosity_level
      (UVM_FULL);
29     // Print the test topology.
30     this.print(printer);
31   endfunction : end_of_elaboration();
32   // UVM run() phase
33   task run();
34     // Set a drain time for the environment if desired.
35     uvm_test_done.set_drain_time(this, 50);
36   endtask: run
37 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 Section 8.5.

Line 3 - Line 5 All tests should derive from the `uvm_test` class and use the `` `uvm_component_utils `` or the `` `uvm_component_utils_begin/`uvm_component_utils_end `` macros. See the *UVM 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 `uvm_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 *UVM Class Reference* for different types of printers available.

Line 15 - Line 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 UVM library phasing mechanism during `build()`. The user is not required to explicitly call `xbus_demo_tb0.build()`.

Line 26 - Line 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.

Line 33 - Line 36 Specify the `run()` task for the base test. In this case, we set a drain time of 50 micro-seconds. Once all of the end-of-test objections were dropped, a 50 micro-second delay is introduced before the `run` phase it terminated.

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;
     `uvm_component_utils(test_read_modify_write)
     function new(string name = "test_read_modify_write",
        uvm_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.

8.5 Testbench Environment

This section discusses the testbench created in the *Example: test_lib.sv* in Section 8.4. The code that creates the `xbus_demo_tb` is repeated here.

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

In general, testbenches can contain any number of envs (verification components) of any type: xbus, pci, ahb, ethernet, and so on. The XBus demo creates a simple testbench consisting of a single XBus environment (verification component) with one master agent, slave agent, and bus monitor (see Figure 29).

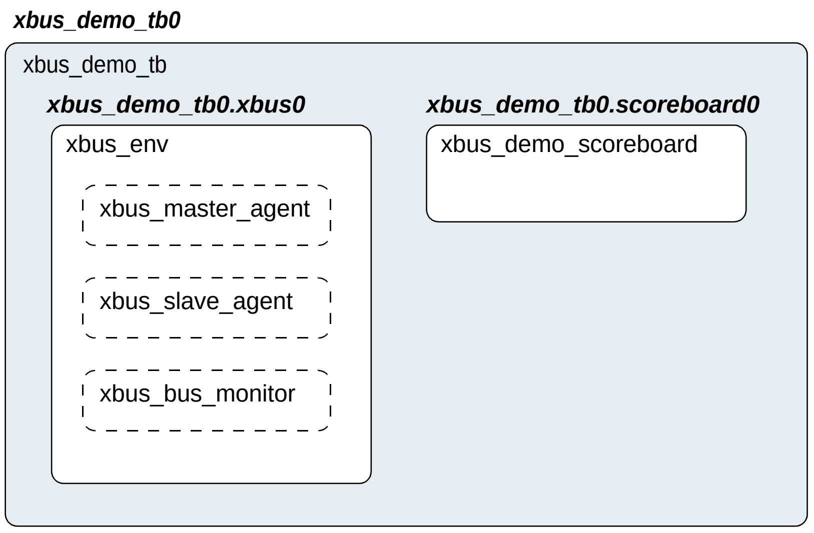


Figure 29—Testbench derived from uvm_env

The following code defines a class that specifies this configuration. The test will create an instance of this class.

Example: xbus_demo_tb.sv

```
1  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_elaboration
```

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.

Line 3 - Line 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 5 As with `xbus0`, the scoreboard is created.

Line 9 Declare the `connect()` function.

Line 10 - Line 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 UVM 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.

8.6 XBus Environment

The `xbus_env` component contains any number of XBus master and slave agents. In this demo, the `xbus_env` (shown in Figure 30) is configured to contain just one master and one slave agent.

NOTE—The bus monitor is created by default.

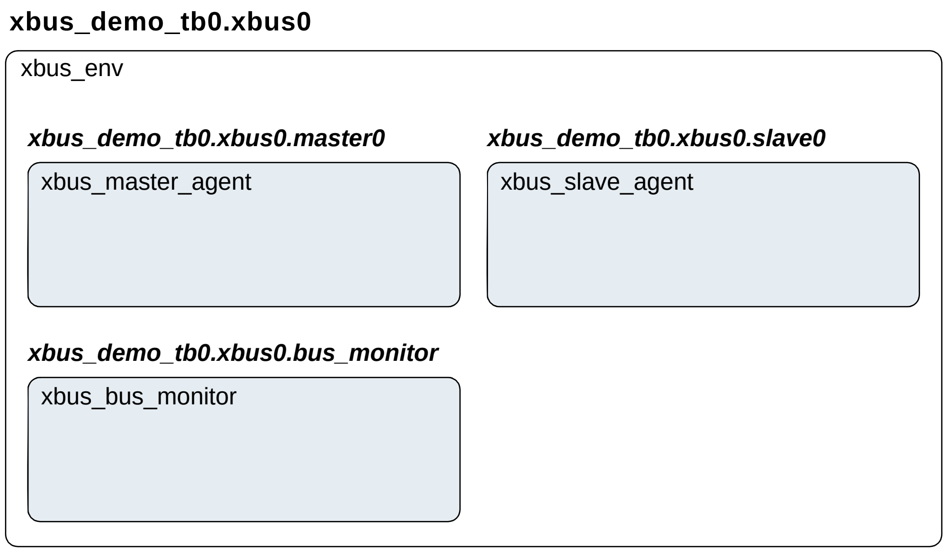

xbus_demo_tb0.xbus0
xbus_env
xbus_demo_tb0.xbus0.master0
xbus_master_agent
xbus_demo_tb0.xbus0.slave0
xbus_slave_agent
xbus_demo_tb0.xbus0.bus_monitor
xbus_bus_monitor

Figure 30—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.

```
1  function void xbus_env::build();
2    string inst_name;
3    super.build();
4    if(has_bus_monitor == 1) begin
5      bus_monitor = xbus_bus_monitor::type_id::create("bus_monitor",
       this);
6    end
7    masters = new[num_masters];
8    for(int i = 0; i < num_masters; i++) begin
9      $sformat(inst_name, "masters[%0d]", i);
10     masters[i] = xbus_master_agent::type_id::create(inst_name, this);
11     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.

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

Line 7 - Line 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 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.

Line 13 - Line 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.

8.7 XBus Agent

The `xbus_master_agent` (shown in Figure 31) 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=UVM_PASSIVE`), which disables the creation of the sequencer and driver. The `xbus_master_agent` is derived from `uvm_agent`.

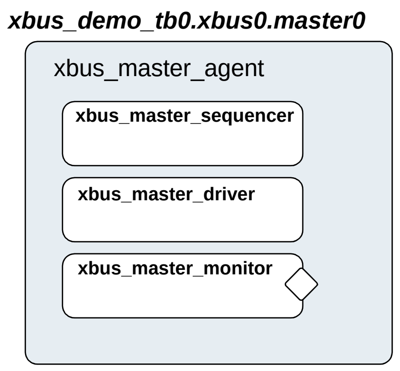


Figure 31—Instance of xbus_master_agent

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 == UVM_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 == UVM_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.

Line 4 - Line 7 Create the sequencer and driver if the `is_active` control field is set to `UVM_ACTIVE`. The `create_component` function is used for creation.

Line 10 Declare the `connect()` function.

Line 11 - Line 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.

8.8 XBus Sequencer

This component controls the flow of sequence items to the driver (see Figure 32).

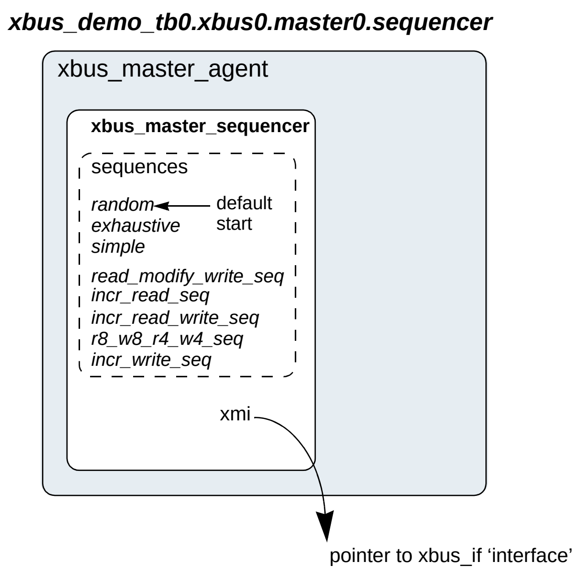


Figure 32—Instance of xbus_master_sequencer

The sequencer controls which sequence items are provided to the driver. The `uvm_sequencer` base class includes three built-in sequences: `uvm_random_sequence`, `uvm_exhaustive_sequence`, and `uvm_simple_sequence`. The `default_sequence` property selects the sequence to start. By default, a sequence of type `uvm_random_sequence` is started.

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 `` `uvm_update_sequence_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.

8.9 XBus Driver

This component drives the XBus bus-signals interface by way of the `xmi` virtual interface property (see Figure 33). 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.

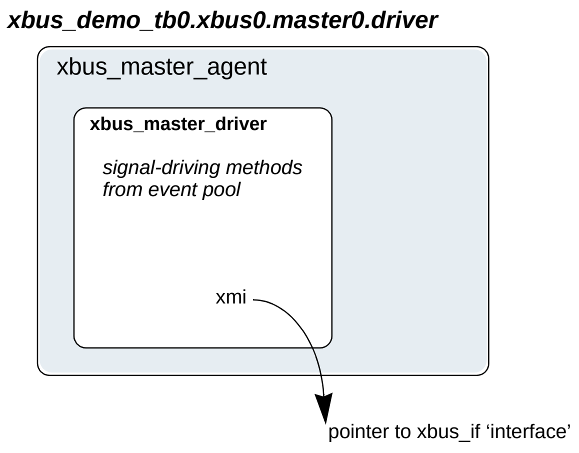

xbus_demo_tb0.xbus0.master0.driver
xbus_master_agent
xbus_master_driver
signal-driving methods from event pool
xmi
pointer to xbus_if 'interface'

Figure 33—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 UVM's built-in simulation phasing (discussed in Section 7.2). 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.

8.10 XBus Agent Monitor

The XBus monitor collects `xbus_transfers` seen on the XBus signal-level interface (see Figure 34). If the checks and coverage are present, those corresponding functions are performed as well.

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 Section 7.2 for more information about simulation phases.

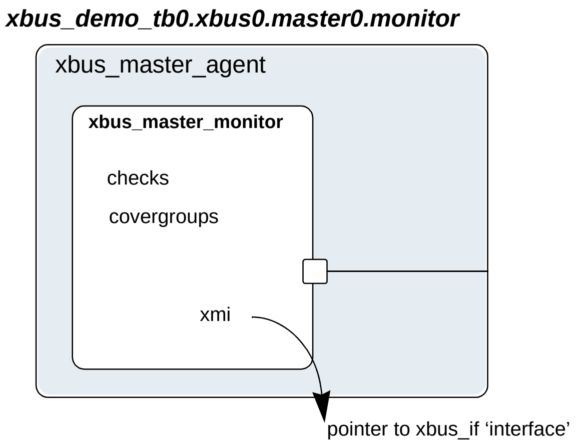

xbus_demo_tb0.xbus0.master0.monitor
xbus_master_agent
xbus_master_monitor
checks
covergroups
xmi
pointer to xbus_if 'interface'

Figure 34—Instance of xbus_master_monitor

The monitor's functionality is contained in an infinite loop defined with the `run()` task. Once all of the `end_of_test` objections were dropped, the `global_stop_request()` is called causing the `run()` tasks to finish, allowing other simulation phases to complete, and the simulation itself 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`.

8.11 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 `1`. 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 Section 8.6.

8.11.1 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 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 `grant0` is asserted, it assumes `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 to which the slave should respond. 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.

8.11.2 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.

8.11.3 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

a) `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.

b) `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.

8.11.4 Checks and Coverage

The XBus bus monitor performs protocol-specific checks using class checks and collects functional coverage from the collected `xbus_transfers`.

The UVM field `coverage_enable` and `checks_enable` are used to control whether coverage and checks, respectively, will be performed or not. Refer to Section 6.9 for more information.

8.12 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 Section 6.9.

Assertions are added to perform physical checks.The `xbus_env` field `intf_checks_enable` controls whether these checks are performed. Refer to Section 6.9 for more information.

The code below is an example of a physical check for the XBus interface, which confirms 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
```

9. XBus Specification

9.1 Introduction

9.1.1 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 verification component. 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.

9.1.2 Bus Overview

The XBus is a simple non-multiplexed, synchronous bus with no pipelining (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.

9.2 Bus Description

9.2.1 Bus Signals

The list of bus signals (not including arbitration signals) is shown in Table 3. All control signals are active high.

Table 3—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: 00 => 1 byte 01 => 2 bytes 10 => 4 bytes 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)

Table 3—Bus Signals (Continued)

Signal Name	Width (bits)	Driven By	Purpose
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 **read**s and **write**s
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

9.2.2 Clocking

All bus agents operate synchronous to the rising edge of the *clock* signal with the exception of *gnt* signals (see Section 9.3).

9.2.3 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.

9.3 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 “no operation” (NOP) Address Phase (see Section 9.4.1). The last cycle of a Data Phase is defined as a Data Phase cycle in which 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 *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 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 choose an appropriate arbitration system. Arbiters might allocate different priorities to each master or might choose randomly with each master having equal priority.

9.4 Address Phase

The Address Phase lasts for a single clock cycle and always immediately follows the Arbitration Phase.

9.4.1 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 the arbiter is connected to certain bus signals in addition to the arbitration signals and behaves as a "default master".

9.4.2 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.

9.5 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).

9.5.1 Write Transfer

The master shall drive the first byte of data onto the bus on the clock cycle after driving a write Address Phase. 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 where 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.

9.5.2 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.

9.5.3 Read Transfer

On the clock cycle after the master drives a read Address Phase, the slave can take one of two actions: drive the first byte of data onto the bus while driving the *wait* signal low or drive the *wait* signal high to indicate 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 where 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.

9.5.4 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.

9.6 How Data is Driven

Table 4 specifies how data is driven in the XBus specification.

Table 4—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 **write**s. Driven by slave during **read**s 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
error	Not driven	Not driven	Driven by slave

9.7 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; i.e., 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).

9.7.1 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*.

9.7.2 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.

9.7.3 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).

9.8 Example Timing Diagrams

Figure 35 and Figure 36 show sample timing diagrams.

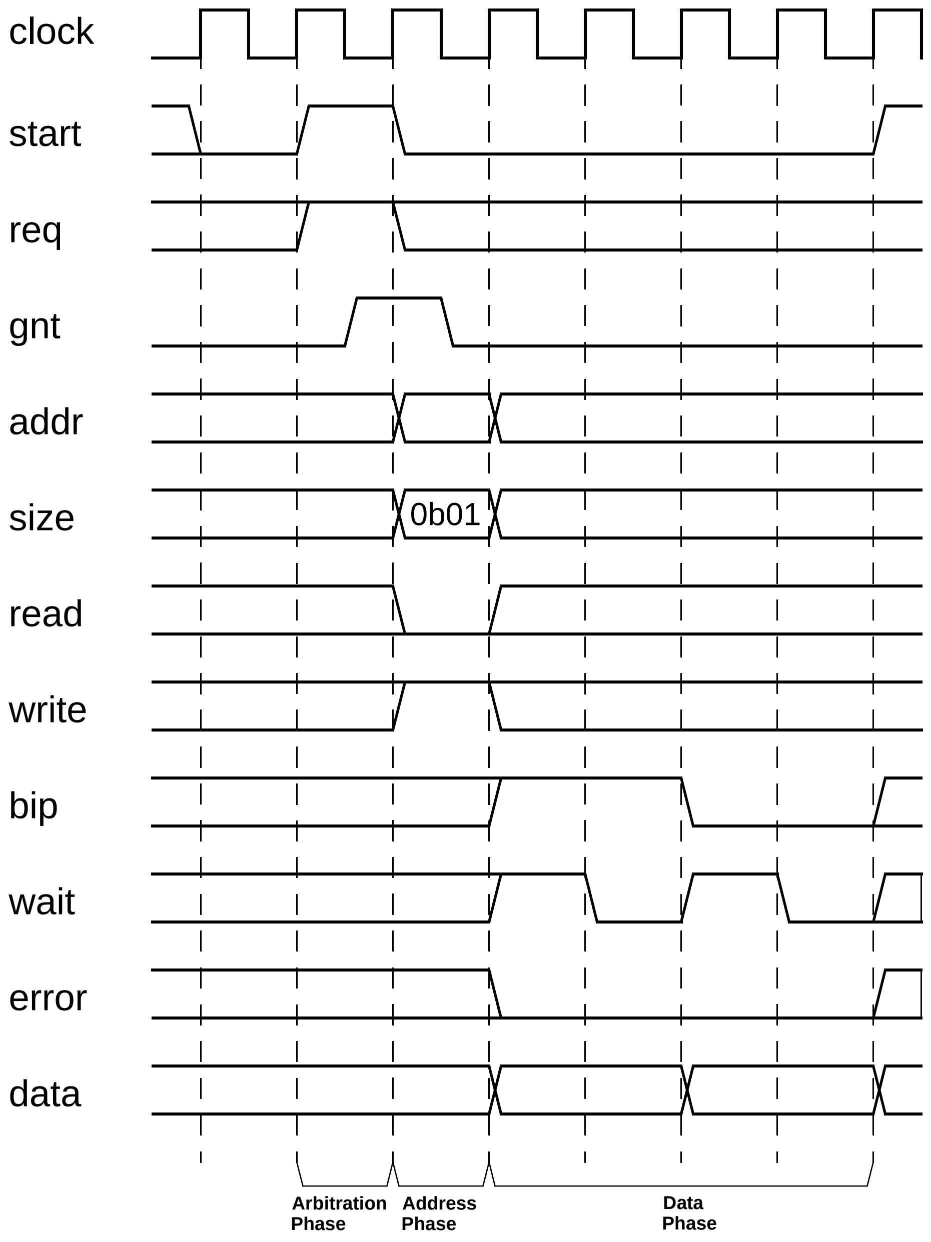


Figure 35—Example Write Waveform

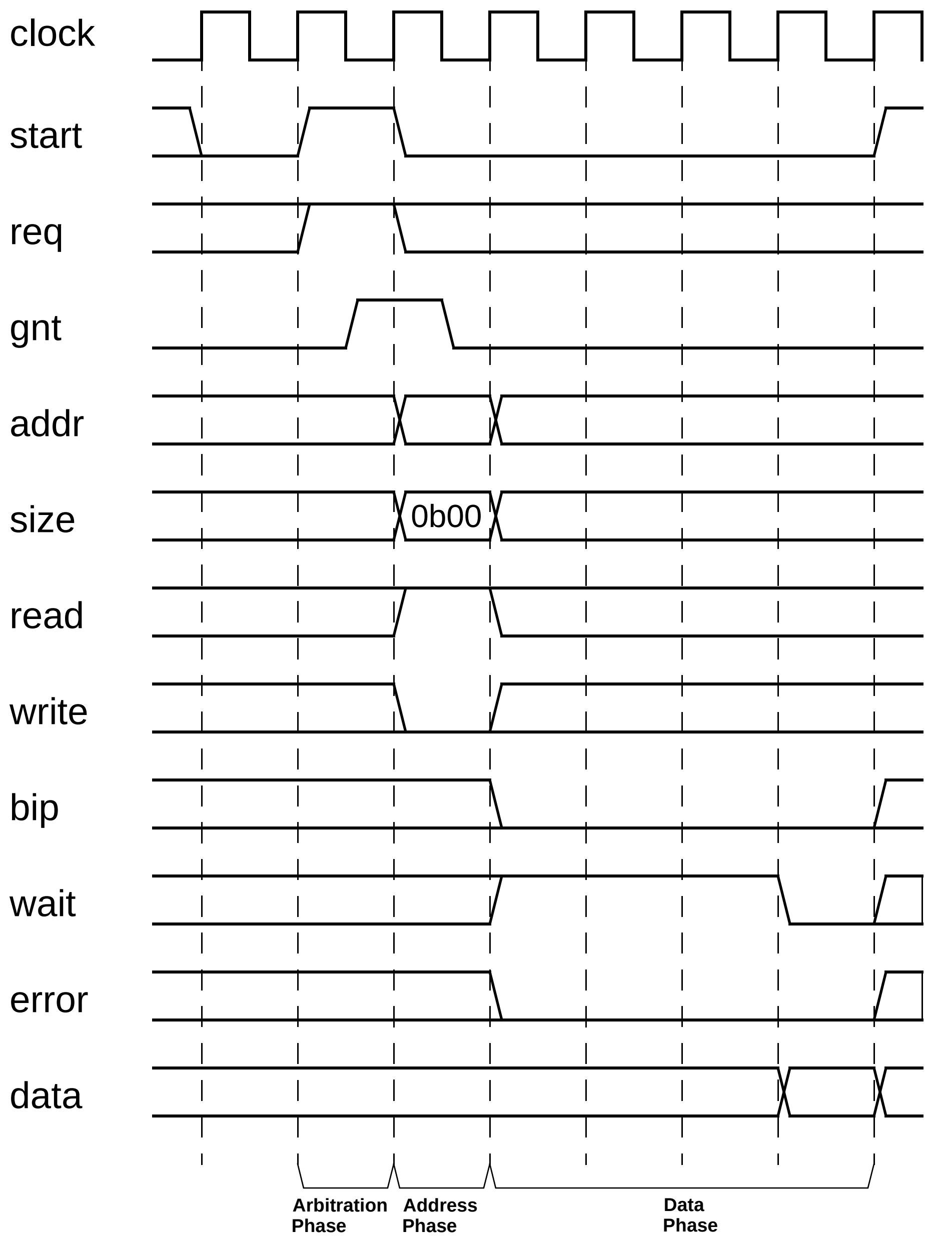


Figure 36—Example Read Waveform

Appendix A

(informative)

Bibliography

[B1] Open SystemC Initiative (OSCI), Transaction Level Modeling (TLM) Library, Release 1.0.

[B2] For a summary of OVM, see the following Internet location: http://www.ovmworld.org.

[B3] For a summary of VMM, see the following Internet location: http://www.vmmcentral.org.

[B4] For practical UVM examples, see the following Internet location:
http://www.accellera.org/activities/vip.