5 Developing Acceleratable Universal Verification Components (UVCs)

This chapter discusses the following topics:

- Introduction to UVM Acceleration
- UVC Architecture
- UVM Acceleration Package Interfaces
- SCE-MI Hardware Interface
- Building Acceleratable UVCs in SystemVerilog
- Building Acceleratable UVCs in e
- Collector and Monitor

5.1 Introduction to UVM Acceleration

The acceleratable Universal Verification Methodology (UVM) packages allow portions of a standard UVM environment to be accelerated using a hardware accelerator. The extended UVM acceleration packages include support for SystemVerilog and the ***e*** high-level verification languages (HVLs). Though this chapter only discusses UVM, acceleration for both the Open Verification Methodology (OVM) and UVM are supported. So, any references to UVM equally apply to OVM.

The purpose of extending UVM to include hardware acceleration is to enable the verification environment to execute faster. Hardware acceleration can dramatically increase run time performance and, therefore, allow more testing to be done in a shorter amount of time, and making the verification engineer more productive.

Although the main purpose of using the UVM acceleration library is to allow a hardware accelerator to be used, it is not restricted to hardware acceleration alone. UVM acceleration is truly an extension of the standard simulation-only UVM, and is fully backwards compatible with it. This means that Universal

Verification Components (UVCs) architected to be acceleratable can be used in either a simulation-only environment or a hardware-accelerated environment. However, UVCs that were not architected to leverage hardware acceleration will require some modifications to enable them to be used in a hardware-accelerated environment.

5.2 UVC Architecture

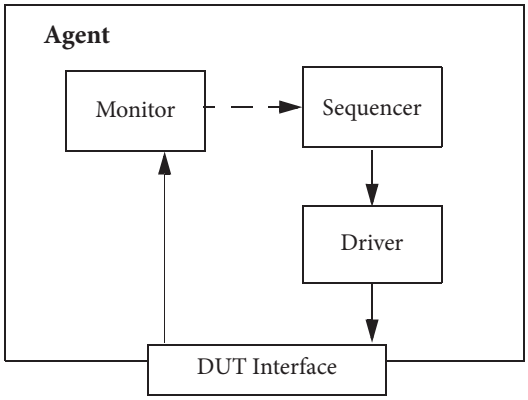
This section briefly describes the standard UVC architecture and explains how this differs from the acceleratable UVC architecture.

5.2.1 Standard UVC Architecture

UVCs based on the standard UVM typically contain the following three main components, which are themselves contained within an agent component, as shown in Figure 5-1 below. Each agent contains:

- A sequencer (also known as sequence driver)
- A driver (also known as BFM)
- A monitor

Figure 5-1 Standard UVC Architecture



5.2.2 Active Agent

The architecture shown in Figure 5-1 is typical of an agent that actively drives stimulus into the device under test (DUT).

Stimulus is provided by the sequencer in an abstract form known as a *data item*. Data items are transactions that only contain stimulus information; the interface and protocol details related to the DUT are abstracted out. Data items in SystemVerilog are classes extended from the `uvm_sequence_item` class. In ***e***, these are extended from the `any_sequence_item` item.

The driver connects to the DUT interface and applies the data items provided by the sequencer to this interface in accordance with the interface protocol.

A monitor is used to observe the activity on the DUT interface as well as activity on internal nodes of the DUT to collect coverage metrics about what parts of the DUT have been exercised. A standard UVM monitor usually includes a hard-coded connection to the interface as well as the coverage-collection functionality. Having a hard-coded connection to the interface is not ideal if the UVC is to be used to verify a DUT at multiple levels of abstraction because a new monitor will need to be created for each abstraction level.

5.2.3 Passive Agent

A UVC can be configured solely to collect DUT activity rather than to stimulate activity. The collected information can then be used by checkers, coverage tools, and the testbench itself for cases where up-to-date status is required. This is a typical scenario when the DUT is integrated into a system. Under these circumstances, the sequencer and driver components are disabled leaving only the monitor. The agent in this scenario is referred to as a *passive agent*.

Coverage information allows the verification team to ensure that the DUT is thoroughly tested by measuring the features that have been exercised, and the ones that have not. Coverage information can also be used by a scoreboard component that can be used to track the features that have been tested.

A UVC can be used to verify models at various levels of abstraction, each with different types of interfaces. Decoupling the stimulus generation from driving the physical DUT interface allows stimulus to be reused for verifying different abstractions of a given model by simply selecting the appropriate driver. This is most applicable to simulation environments that support the broadest range of HVL constructs.

5.2.4 Acceleratable UVCs

Acceleratable UVCs benefit from a slightly different architecture than simulation-only UVCs in order to maximize the performance gain provided by the hardware accelerator. Therefore, in order to describe acceleratable UVCs, a brief introduction to hardware acceleration must be given. More information about hardware acceleration can be found in the *UXE User's Guide,* which is included with the Cadence Palladium XP family of hardware accelerators.

5.2.4.1 Hardware Acceleration

Hardware acceleration is performed by combining a software simulator that executes on a workstation with a dedicated hardware-acceleration machine. The complete verification environment is partitioned to have some models executed by the simulator and others by the hardware accelerator. Models described using high-level verification language (HVL) constructs are executed by the simulator, and are said to reside in the *HVL partition*. Models described using hardware description language (HDL) constructs are executed by the hardware accelerator, and are said to reside in the *HDL partition*.

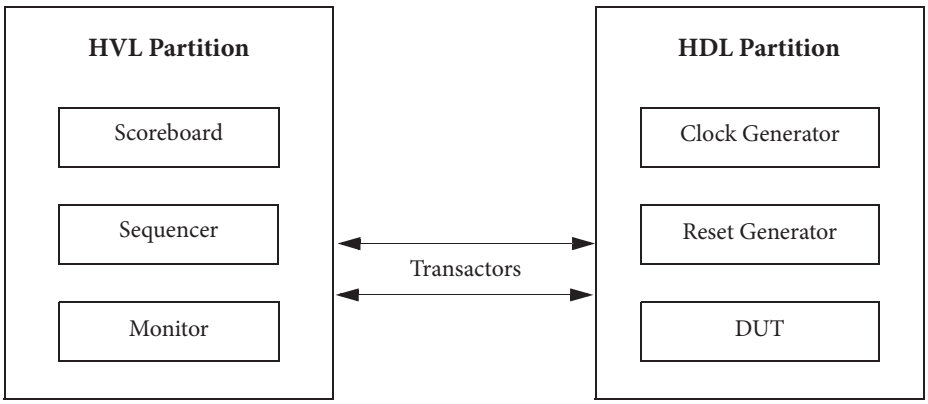
Hardware accelerators can only accelerate models that have been described using the acceleratable subset of an HDL. This subset is usually assumed to be the same as the register-transfer-level (RTL) subset defined for hardware synthesis, but this is often not the case. Hardware acceleration platforms usually accept a number of

behavioral constructs as well as synthesizable constructs. So, the level of support is greater than that contained in the synthesizable subset of constructs; however, it is still a subset of the complete HDL. Any component that cannot be modeled using this subset must remain in the HVL partition. One requirement that must be fulfilled is that all models in the HDL partition must have signal-level interfaces. However, signal-level connections joining components in the HVL partition to components in the HDL partition are not efficient for achieving high runtime performance. Instead, a transaction-based connection must be used. The industry recognized this concept as being a key requirement in order to achieve high runtime performance when connecting a software simulator to a hardware accelerator. This led to the creation and standardization of the Accellera Standard Co-Emulation API: Modeling Interface, more commonly referred to as SCE-MI.

The use of a transaction-based interface between the software simulator and the hardware accelerator not only allows the communication between the two engines to be made more efficient, it also allows the execution of simulation models to be made more efficient. This is because simulation performance is reduced when the models being executed become more detailed and require timing. Therefore, simulating untimed models at the transaction level improves simulation performance.

The partitioning of transaction-level components and cycle-accurate signal-level components between the software simulator and hardware accelerator respectively, leads to a change in the overall verification environment architecture. Two separate top levels of hierarchy are created for each of the two partitions, with all communication between the two partitions being performed at the transaction level. Components like scoreboards, sequencers and monitors are placed in the HVL partition, while components like clock generators, reset generators, and the signal-level DUT are placed in the HDL partition, as shown in Figure 5-2.

Figure 5-2 Components of the HVL and HDL Partitions

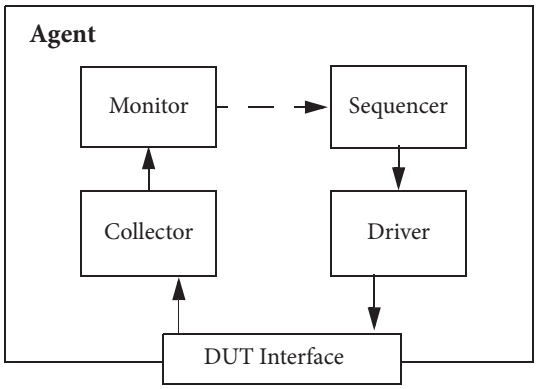


Acceleratable UVC Architecture

To enable high runtime performance to be achieved, all models that reside in the HVL partition should execute at the transaction level, and all models that require cycle-accurate timing should reside in the HDL partition. Transactors are used to allow components within each partition to communicate with each other efficiently. However, the architecture shown in Figure 5-1 on page 196 does not allow a clean division of

functionality to be made because the monitor operates at the same abstraction level as the DUT, which for acceleration would be at the signal level. To address this, the monitor should be split into two components, a monitor and a collector, as shown in Figure 5-3.

Figure 5-3 Acceleratable UVC Architecture



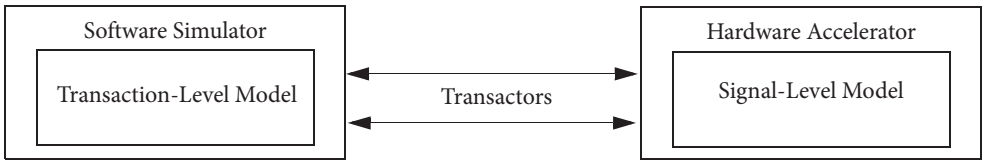
The purpose of the collector is to allow the physical interface required by the DUT to be separated from the functionality provided by the monitor. This means that the monitor and sequencer, and all hierarchical levels above, can operate at the transaction level, irrespective of the type of interface required by the DUT. Modeling these components at this level of abstraction is good for reuse as well as for increasing execution performance. The collector and driver components implement the physical interface required to enable the UVC to connect to the DUT, which can be easily altered depending on the type of interface required without affecting the rest of the UVC.

As mentioned previously, for the hardware acceleration mode, models that reside in the HVL partition operate at the transaction level, while those that reside in the HDL partition execute at the signal level. One consequence of configuring the UVC to use hardware acceleration is that the acceleratable collector and driver components must incorporate transactors to convert signal-level activity to transactions, and vice versa.

Acceleratable Transactors

Transactors are an abstraction bridge between the components that operate at the transaction level and the components that operate at the signal level. For hardware acceleration, transactors extend this capability by bridging between transaction-based components being executed by a software simulator and signal-based components being executed by a hardware accelerator as shown in Figure 5-4 on page 200.

Figure 5-4 Acceleratable Transactors



To bridge between the HVL partition and the HDL partition, the transactors have three main components:

- Proxy model

  The proxy model is instantiated in the HVL partition and accesses the communication channel by way of an Application Programming Interface (API).

- Bus Functional Model (BFM)

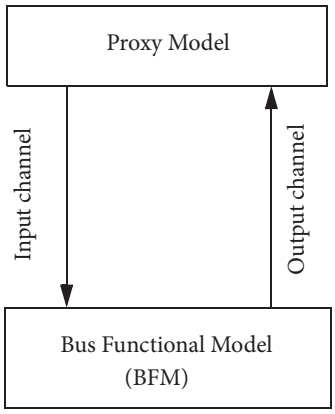
  The BFM is instantiated in the HDL partition and also accesses the communication channel by way of an API.

- Communication channel that connects between the Proxy and BFM

  Each channel is uni-directional and this is reflected in the choice of interface used within each of the partitions.

A simple transactor with one input and one output channel is shown in Figure 5-5.

Figure 5-5 Transactor Example with Input and Output Channels



To enable transactors to operate on different vendor's hardware acceleration platforms, a standard vendor-independent API was defined and standardized by Accellera for connecting any software simulator to any hardware accelerator. The standard, known as the **Standard Co-Emulation API: Modelling Interface (SCE-MI),** defines a multichannel communication interface.

SCE-MI initially defined a macro-based interface. But later, it added a simpler Direct Programming Interface (DPI) and a more complex but feature-rich pipes-based interface. All of the above interfaces are described in the *Standard Co-Emulation API: Modelling Interface (SCE-MI) Reference Manual*, Version 2.0, or later, and is available from Accellera. The UVM Acceleration interface uses SCE-MI pipes communications channels.

SCE-MI pipes are unidirectional channels that allow transactions to be streamed from components in the HVL partition to components in the HDL partition and vice versa. A C language API is available to components residing in the HVL partition and a SystemVerilog API is available to components residing in the HDL partition.

To simplify the use of SCE-MI in the development of acceleratable UVCs, a UVM Acceleration library, *uvm_accel*, is provided in the Cadence UXE software release to hide the semantics of the SCE-MI C API presented to models that reside in the HVL partition. The `uvm_accel` package provides a UVM-based API that is native to the verification language being used. Most UVM verification environments are built using SystemVerilog, ***e,*** or a combination of both, and the `uvm_accel` library supports both. The `uvm_accel` package allows the proxy model part of a transactor to be written in SystemVerilog or ***e***, whichever is the most suitable language, which is often the same as the language used to model the rest of the verification environment.

The BFM part of the transactor, implemented using the acceleratable subset of SystemVerilog and Verilog, uses the SCE-MI SystemVerilog interfaces to access the SCE-MI pipes based channels. These interfaces provide SystemVerilog tasks and functions that greatly simplifies the usage. More information about the SCE-MI Pipes interfaces can be obtained from the *Standard Co-Emulation API: Modeling Interface (SCE-MI) Reference Manual* from Accellera.

5.3 UVM Acceleration Package Interfaces

The UVM package provides two unidirectional interfaces, one to access input channels and the other to access output channels. The terms input and output are defined in relation to the hardware accelerator with input being into the hardware accelerator and output being out of the hardware accelerator.

For SystemVerilog, each interface is defined as a class that inherits from the `uvm_accel_pipe_proxy_base`. For ***e***, each interface is defined as a unit that inherits from the `uvm_accel_pipe_proxy_base` unit.

5.3.1 uvm_accel_pipe_proxy_base Task and Function Definitions (SystemVerilog)

Task / Function	Definition
`extern function void build_phase(phase);`	Called during the environment `build_phase` phase. Gets configuration parameters such as `hdl_path` and `autoflush` and uses them to configure the proxy. To improve performance, you should run with `pipe_proxies` configured with autoflush disabled, whenever possible. For more information on autoflush, refer to the *Standard Co-Emulation API: Modeling Interface (SCE-MI) Reference Manual*.
`extern function void end_of_elaboration_phase(phase);`	Called at the end of elaboration. `hdl_path` must be configured before this function is called since port binding occurs during this phase.
`extern function void set_pipe_name(string name);`	Used to define the full hierarchical instance name of a pipe. The pipe name must be defined before `end_of_elaboration_phase` if it is to take effect. The name is given as a string.
`extern function string get_pipe_name();`	Returns the hierarchical instance name of the pipe that will be, or has been bound.
`extern function bit set_autoflush(bit enable);`	Sets the autoflush semantics of the pipe. An input of 1 turns autoflush on for all subsequent messages, and an input of 0 turns it off for subsequent messages. This setting can be made at anytime. The default is autoflush enabled (1).
`extern function bit get_autoflush();`	Returns the autoflush setting of the pipe.
`extern function int unsigned get_pipe_depth();`	Returns the number of elements that the pipe holds.
`extern function int unsigned get_pipe_width();`	Returns the number of bytes of each element.
`extern function int unsigned get_pipe_handle();`	Returns the internal handle that is used by the proxy to communicate with the actual pipe. This handle should not be used directly. Each actual pipe will have a unique handle.

SystemVerilog uvm_accel_input_pipe_proxy

The SystemVerilog `uvm_accel_input_pipe_proxy` class definition is shown below:

```
class uvm_accel_input_pipe_proxy #(type T=uvm_object,
```

```
   type S=uvm_accel_object_serializer#(T))                  // Parameterizable
                                                           // serializer type
   extends uvm_accel_pipe_proxy_base;
   uvm_put_imp #(T,uvm_accel_input_pipe_proxy#(T)) put_export;  // TLM port
                                                                // binding
   uvm_analysis_port #(T) put_ap;                          // Analysis port
   extern function new(string name, uvm_component parent); // Constructor
   extern task put(T t);                                   // Blocking put
   extern function bit try_put (T t);                      // Non-blocking put
   extern function bit can_put();                          // Non-blocking can
                                                           // put test
endclass
```

Each input pipe proxy instance can be customized to accept different types of data item and use different serialization schemes. Customization is achieved via the parameters `uvm_object` and `uvm_accel_object_serializer`. Each data item is defined as a class in SystemVerilog which inherits from `uvm_sequence_item`. A data item typically contains data members that may or may not be randomized, UVM utility fields to enable or disable UVM automation for each of the data members, and constraints to constrain any data members that are to be randomized. In addition, serialization and de-serialization methods may also be provided for specific fields of the data item where the default serializer/de-serializer is not sufficient.

Table 5-1 SystemVerilog uvm_accel_input_pipe_proxy Task and Function Definitions

Task / Function	Definition
`extern task put(T t);`	Sends a user-defined data item of type T.
`extern function bit try_put (T t);`	Sends a user-defined data item of type T, if possible.
`extern function bit can_put();`	Returns 1 if the component is ready to accept the data item; 0 otherwise.

SystemVerilog uvm_accel_output_pipe_proxy

The SystemVerilog `uvm_accel_output_pipe_proxy` class definition is shown below:

```
class uvm_accel_output_pipe_proxy#(type T=uvm_object,
   type S=uvm_accel_object_serializer#(T))                 // Parameterizable
                                                           // deserializer type
   extends uvm_accel_pipe_proxy_base;
   uvm_get_imp#(T,uvm_accel_output_pipe_proxy#(T)) get_export; // TLM port
   binding
   uvm_analysis_port#(T) get_ap;                           // Analysis port
   extern function new(string name, uvm_component parent);// Constructor
   extern task get(inout T t);                             // Blocking get
   extern function bit try_get (T t);                      // Non-blocking get
   extern function bit can_get();                         // Non-blocking can get
endclass
```

Each output pipe proxy can be customized to accept different types of data item and use different de-serialization schemes. Customization is achieved by way of the parameters `uvm_object` and `uvm_accel_object_serializer`.

Table 5-2 SystemVerilog uvm_accel_output_pipe_proxy Task and Function Definitions

Task / Function	Definition
`extern task get(inout T t);`	Provides a new data item of type T.
`extern function bit try_get (T t);`	Provides a new data item of type T, if possible
`extern function bit can_get();`	Returns 1 if a new data item can be provided immediately upon request, 0 otherwise.

5.3.2 uvm_accel_pipe_proxy_base Task and Function Definitions (*e*)

e uvm_accel_input_pipe_proxy

Table 5-3 *e* uvm_accel_input_pipe_proxy Unit Definition

Task / Function	Definition
`get_pipe_full_path() : string`	Returns the hierarchical instance name of the pipe that will be, or has been bound.
`set_autoflush(enable : bool)`	Sets the autoflush semantics of the pipe. An input of `1` enables autoflush for all subsequent messages, and an input of `0` disables it for subsequent messages. This setting can be made at anytime. The default is autoflush enabled (`1`).
`get_pipe_autoflush() : bool`	Returns the autoflush setting of the pipe.
`get_pipe_depth() : uint`	Returns the number of elements that the pipe holds.

```
template unit uvm_accel_input_pipe_proxy of (<type>) like
uvm_accel_pipe_proxy_base {
    !value : <type>;
    m_in : interface_imp of tlm_put of <type> is instance; //TLM Interface
    put(value: <type> )                                    //Blocking put
    try_put(value: <type>) : bool                          //Non-blocking put
    can_put(): bool                                       //Non-blocking can
                                                           //put test
};
```

Each input pipe proxy instance can accept different types of data items. Each data item is defined as a unit in ***e***, and is like `any_sequence_item`. A data item typically contains data members that may or may not be randomized, and includes constraints to constrain data members that are to be randomized. In addition,

pack and unpack methods may also be provided for specific fields of the data item where the default packer or unpacker is not sufficient.

Table 5-4 *e* uvm_accel_input_pipe_proxy Task and Function Definitions

Task / Function	Definition
put(value: <*T*>)	Sends a user-defined data item of type T.
try_put(value: <*T*>) : bool	Sends a user-defined data item of type T, if possible.
can_put(): bool	Returns TRUE if the component is ready to accept the data item; FALSE otherwise.

e uvm_accel_output_pipe_proxy

The *e* uvm_accel_output_pipe_proxy unit definition is shown below:

```
template unit uvm_accel_output_pipe_proxy of (<type>) like
uvm_accel_pipe_proxy_base {
          !m_value : <type>;
          m_out : interface_imp of tlm_get of <type> is instance; // TLM Interface
          get(value: *<type>)                                     // Blocking get
          try_get(value: *<type>): bool                           // Non-blocking
                                                                  // get
          can_get(): bool                                         // Non-blocking
                                                                  // can get
};
```

Each output pipe proxy can be customized to accept different types of data item.

Table 5-5 *e* uvm_accel_output_pipe_proxy Task and Function Definitions

Task / Function	Definition
get(value: *<*T*>)	Sends a user-defined data item of type T.
try_get(value: *<*T*>): bool	Sends a user-defined data item of type T, if possible
can_get(): bool	Returns 1 if a new data item can be provided immediately upon request, 0 otherwise.

5.4 SCE-MI Hardware Interface

The SCE-MI API used by the BFMs that exist in the HDL partition are defined in the *Standard Co-Emulation API: Modeling Interface (SCE-MI) Reference Manual*. The HDL side API for input and output interfaces are given here for reference. For complete details, refer to the SCE-MI Reference Manual.

5.4.1 SCE-MI Input Pipe Interface

```
interface scemi_input_pipe();
   parameter BYTES_PER_ELEMENT = 1;
   parameter PAYLOAD_MAX_ELEMENTS = 1;
   parameter BUFFER_MAX_ELEMENTS = <vendor specified>;
   localparam PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

   task receive(
       input int num_elements,                    // # elements to be read
       output int num_elements_valid,             // # elements that are valid
       output bit [PAYLOAD_MAX_BITS-1:0] data,    // data
       output bit eom );                          // end-of-message marker flag
       <implementation goes here>
   endtask

   function int try_receive(                 // return: #requested elements
                                             // that are actually received
       input int byte_offset,                    // byte_offset into data
       input int num_elements,                   // # elements to be read
       output bit [PAYLOAD_MAX_BITS-1:0] data,   // data
       output bit eom );                         // end-of-message marker flag
       <implementation goes here>
   endfunction

   function int can_receive();               // return: #elements that can
                                             // be received
       <implementation goes here>
   endfunction

   modport receive_if(import receive, try_receive, can_receive );
endinterface
```

5.4.2 SCE-MI Output Pipe Interface

```
interface scemi_output_pipe();
   parameter BYTES_PER_ELEMENT = 1;
   parameter PAYLOAD_MAX_ELEMENTS = 1;
   parameter BUFFER_MAX_ELEMENTS = <vendor specified>;
   localparam PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

   task send(
       input int num_elements,                 // input: #elements     to be
   written
       input bit [PAYLOAD_MAX_BITS-1:0] data, // input: data
       input bit eom ); // input: end-of-message marker flag
       <implementation goes here>
   endtask

   task flush;
```

```
    <implementation goes here>
  endtask

  function int try_send(                      // return: #requested elements
                                              // that are actually sent
      input int byte_offset,                  // input: byte_offset into
                                              // data, below
      input int num_elements,                 // input: #elements to be sent
      input bit [PAYLOAD_MAX_BITS-1:0] data,  // input: data
      input bit eom );                        // input: end-of-message marker
                                              // flag
      <implementation goes here>
  endfunction

  function int can_send();              // return: #elements that can be sent
      <implementation goes here>
  endfunction

  modport send_if( import send, flush, try_send, can_send );
endinterface
```

5.5 Building Acceleratable UVCs in SystemVerilog

5.5.1 Data Items

Data items are transactions, which are implemented as class objects that are inherited from `uvm_sequence_item`, that itself inherits from `uvm_transaction`. A data item contains data members, UVM utility fields to enable or disable UVM automation for each of the data members, and constraints to constrain any data members that are to be randomized. In addition, you may provide your own serialization and de-serialization methods. The code snippet below, taken from a simple SystemVerilog example, yamp, shows the class definition of a data item called `yamp_transfer` along with its data members.

```
  typedef enum bit { READ, WRITE } direction_t;    // Enumerated type used to
                                                   // define memory access
                                                   // direction
  class yamp_transfer extends uvm_sequence_item;   // yamp_transfer class
                                                   // inherited from
                                                   //'uvm_sequence_item'
                                                   // class
      rand direction_t direction;                  // Memory access direction
                                                   //(READ OR WRITE)
      rand bit [2:0] wait_states;                  // Used by the Driver to
                                                   // insert wait states
      rand bit [3:0] transfer_delay;               // Used by the Driver to
                                                   // insert a transfer delay
      rand bit [7:0] size;                         // Size of data transfer
      rand bit [15:0] addr;                        // Start address of memory
                                                   // access
```

```
    rand bit [15:0] data [];                    // Data to be read or
                                                // written to memory
```

SystemVerilog data members can be randomized as shown by preceding their declaration with the keyword `rand`. Data items can contain statically sized data members as well as dynamically sized data members such as `data[]` shown in the example.

Data items that contain randomly assigned data members require constraints to constrain the range of values they will be assigned. Constraints can be defined within the class definition as shown below or in a separate constraints file.

```
constraint default_wr_size_c {(direction == WRITE) -> data.size() == size;
                              (direction == READ) -> data.size() == 0; }
constraint default_size_c { size inside { [1:10] }; }
constraint default_delay_c { transfer_delay inside {[1:5]};}
```

`uvm_object_utils` macros are used to enable common operations declared in `uvm_object` such as copy, compare, and print as shown below.

```
`uvm_object_utils_begin(yamp_transfer)              // Start of UVM utility
                                                    // macro definitions
`uvm_field_enum(direction_t, direction, UVM_ALL_ON)
`uvm_field_int(wait_states, UVM_ALL_ON)
`uvm_field_int(transfer_delay, UVM_ALL_ON)
`uvm_field_int(size, UVM_ALL_ON)
`uvm_field_int(addr, UVM_ALL_ON + uvm_HEX)
`uvm_field_array_int(data, UVM_ALL_ON + UVM_HEX + UVM_NOPACK)
`uvm_object_utils_end                                 // End of UVM utility
                                                      // macro definitions
```

In order to transfer a data item from the proxy in the HVL partition to the BFM in the HDL partition, the data members must be packed, or serialized, into a vector of bits as shown below.

Figure 5-6 Packed Implementation of Data Item yamp_transfer

dir	wait_states	transfer_delay	size	addr	data []

UVM provides packing capabilities which may or may not be suitable for the data item to be transferred. When data members are statically sized the standard packer is usually sufficient but alternative packing schemes may be required for dynamically sized data members if they have specific requirements. If a field is to be packed using a customized serializer the attribute `UVM_NOPACK` should be set using the `` `uvm_object_util_* `` macro. If the dynamic members do not have any specific requirements then the standard UVM packer can be used for static and dynamic data members. An example of specific `pack` function required by the `yamp` example is shown below.

```
function void do_pack (uvm_packer packer);
    foreach(data[i]) packer.pack_field_int(data[i],16);
endfunction
```

Data items received by the proxy in the HVL partition, from the BFM in the HDL partition, must be unpacked back into the data item class structure. It is the `unpack` operation that usually dictates whether

custom pack and unpack functions are required. The reverse operation employed by the packer must be used by the unpacker. Therefore, if a customized packer was defined then a customized unpacker or deserializer must also be defined. The code snippet below shows the custom unpacker used by the `yamp` example.

```
function void do_unpack (uvm_packer packer);
   data = new [size];                    //size was automatically unpacked
   foreach(data[i]) data[i] = packer.unpack_field_int(16);
endfunction
```

5.5.2 Acceleratable Driver (SystemVerilog)

The *driver* is responsible for taking data items from the sequencer and driving them onto the DUT interface. The DUT can be modeled at multiple levels of abstraction. So, the driver must be able to accommodate each of the interfaces presented by each type of model. This not only affects the type of physical interface used it also affects the functionality of the driver itself. To be able to reconfigure the driver to operate at different levels of abstraction, an enumerated type `uvm_abstraction_level_enum` is used. This enumerated type is defined in the `uvm_accel` package provided by Cadence.

In SystemVerilog, the enumerated type is defined as follows:

```
typedef enum bit [1:0] {UVM_SIGNAL, UVM_TLM, UVM_ACCEL}
uvm_abstraction_level_enum
```

The values defined by this type configure the UVC to operate in pure simulation at the signal level (`UVM_SIGNAL`) or transaction level (`UVM_TLM`) or use hardware acceleration (`UVM_ACCEL`).

When configured for hardware acceleration an acceleratable transactor is used to bridge the gap between the components that operate at the transaction level, which are executed by the software simulator, and the components that operate at the signal level, which are executed by the hardware accelerator. This same acceleratable transactor can also be used for signal based simulation. However, UVCs that have been created for simulation typically use a virtual interface to connect the driver to the DUT and implement the BFM using behavioral constructs. This implementation can continue to be used for simulation to allow a gradual migration to hardware acceleration if required. When using the behavioral BFM the `uvm_abstraction_level_enum` should be set to `UVM_SIGNAL`. If the UVC is to be used to verify abstract SystemC TLM models, the `uvm_abstraction_level_enum` should be set to `UVM_TLM`. The behavior of the driver along with the interface it uses to connect to this model should be customized to suit this type of model.

The following code shows the SystemVerilog code that defines the part of the driver that resides in the HVL partition for the `yamp` example.

```
class yamp_master_driver extends uvm_driver #(yamp_transfer);
   // Virtual interface used to drive HDL signals
   virtual interface yamp_if vif;
   // UVM abstraction level
   protected uvm_abstraction_level_enum abstraction_level = UVM_SIGNAL;
   // SCE-MI input pipe interface
   protected uvm_accel_input_pipe_proxy#(yamp_transfer) m_ip;
   // SCE-MI output pipe interface
```

```
   protected uvm_accel_output_pipe_proxy#(yamp_transfer) m_op;
   // UVM build function
   extern virtual function void build_phase(uvm_phase phase);
   // UVM run task
   extern virtual task run_phase(uvm_phase phase);
   // Task used to drive signals in UVM_SIGNAL mode
   extern virtual protected task get_and_drive();
   // Task used to drive signals in UVM_ACCEL mode
   extern virtual protected task get_and_drive_accel();
endclass : yamp_master_driver
```

The `yamp_master_driver` inherits from the `uvm_driver` class and operates on a data item of type `yamp_transfer`. This example shows a virtual interface, `vif`, which is used for signal level simulation, and two `uvm_accel` pipe proxy interfaces, `m_ip` and `m_op` that are used for hardware acceleration.

Two `uvm_accel` pipe proxy interfaces are required for the `yamp` example since bidirectional communication is required. Each `uvm_accel` pipe proxy interface is unidirectional; therefore, the need for one input interface and one output interface. For most protocols, bidirectional communication is required so it is typical for two or more interfaces to be instantiated. Each `uvm` pipe proxy interface takes a `data item` type as a parameter.

Standard UVM tasks and functions must be defined for each driver. It is recommended that different tasks for each level of abstraction are defined rather than implementing the driver functionality in one task for all the supported levels of abstraction. In the `yamp` example, the `get_and_drive()` task implements the signal-level simulation driver functionality and the `get_and_drive_accel()` task implements the hardware-acceleratable driver functionality. Separating the code into distinct task makes the code easier to understand and debug.

5.5.2.1 build_phase(uvm_phase phase) Function

Each UVM component that inherits from the `uvm_component` class should provide an implementation for a `build_phase` function. Each `build_phase` function is called during the UVM `build_phase` simulation phase to construct the environment hierarchy. In the example shown below, the `abstraction_level` is used to determine the type of interface required by the driver.

```
function void yamp_master_driver::build_phase(uvm_phase phase);
   super.build_phase(phase);
   if (abstraction_level == UVM_ACCEL)
      begin
          m_ip = new("m_ip", this);    // Construct an input port
          m_op = new("m_op", this);    // Construct an output port
         uvm_config_db#(string)::set(this,"m_ip", "hdl_path", "inbox0");
                   // hdl_path used for input port binding
         uvm_config_db#(string)::set(this,"m_op", "hdl_path", "outbox0");
                  // hdl_path used for output port binding
      end
endfunction : build_phase
```

For hardware acceleration, the `abstraction_level` must be set to `UVM_ACCEL` to inform the driver to build and configure a transaction based interface. For the `yamp` example, two ports are constructed: an input port called `m_ip` and an output port called `m_op`. These ports must be bound to valid channels before they can be used and this is achieved by defining a string called `hdl_path` for each port.

Port binding is configured by calling the UVM `uvm_config_db#(string)::set` function for each port defined in the HVL partition. The `uvm_config_db#(string)::set` function causes configuration settings to be created and placed in the `uvm_config` database. The `uvm_config_db#(string)::set` function requires the name of the port instance in the HVL partition, the name of the string variable to be configured (which is `hdl_path` for port binding), and the full hierarchical path from the top level of the HDL partition down to the appropriate port instance in the HDL design hierarchy. In the example given, the full hierarchical path is defined by concatenating the `m_hdl_path` variable with the specific port instance name. The `m_hdl_path` variable is set by the test environment and is the hierarchical path from the top level of the HDL partition down to the BFM instance. The agent then appends the specific port instance name to this path.

If the HDL port, defined by `hdl_path`, is compatible with the HVL port, it will be bound during the `end_of_elaboration_phase` phase; if not, an error will occur. Therefore, the `hdl_path` for each port must be defined before the `end_of_elaboration_phase` phase; it is common to do this during the `build_phase` phase as shown.

5.5.2.2 run_phase(uvm_phase phase) Task

Each UVM component that inherits from the `uvm_component` class, should provide an implementation for a `run_phase` task. Each `run_phase` task is called during the UVM `run_phase` simulation phase and defines the behavior of the driver. The required functionality of the driver will differ depending on the level of abstraction used to implement the DUT. Therefore, the `abstraction_level` is tested and used to alter the driver's behavior as shown below.

```
task yamp_master_driver::run_phase(uvm_phase phase);
    if (abstraction_level == UVM_SIGNAL)      // Signal level simulation
         fork
              get_and_drive();                // Drive signal level DUT interface
         join
    else if (abstraction_level == UVM_ACCEL)  // Hardware acceleration fork
         fork
              get_and_drive_accel();          // Drive SCE-MI transaction level
                                              // interface
         join
endtask
```

If the `abstraction_level` is set to `UVM_SIGNAL`, and a signal-level behavioral BFM has been created for simulation, which is typical of legacy UVCs, a `get_and_drive()` task should be called. This task implements the functionality required to drive this type of interface.

If the `abstraction_level` is set to `UVM_ACCEL`, a `get_and_drive_accel()` task should be called. Different tasks are defined for simulation and acceleratable drivers to allow a legacy behavioral implementation to be used, and coexist with an acceleratable implementation. Acceleratable drivers can be

used with hardware acceleration or simulation. Therefore, the same task could be called, irrespective of whether the `abstraction_level` is set to `UVM_SIGNAL` or `UVM_ACCEL`. This is configured in the `run_phase` task.

5.5.2.3 get_and_drive() Task

The `get_and_drive()` task requests data items from the sequencer, and when appropriate drives the virtual DUT interface signals. It implements the signal-level protocol required by the DUT and drives the DUT signals directly, as shown in the code snippet from the `yamp` example below.

```
task yamp_master_driver::get_and_drive();
   if(vif.sig_reset!==0) @(negedge vif.sig_reset);
   forever begin
      @(posedge vif.clk);
      seq_item_port.get_next_item(req);                // Get new item from
                                                       // the sequencer
      if (transfer.direction == WRITE) begin           // Drive the virtual
                                                       // virtual interface
                                                       // signals
         vif.rd <= 0;
         for (int i=0;i < transfer.size; i++) begin
           repeat (transfer.wait_states) @(posedge vif.clk);
           vif.we <= 1;
           vif.di <= transfer.data[i];
           @(posedge vif.clk);
           vif.addr <= vif.addr + 1;
           vif.we <= 0;
      <rest of implementation>
      seq_item_port.item_done();                    // Communicate item done
                                                    // to the sequencer
endtask : get_and_drive
```

5.5.2.4 get_and_drive_accel() Task

The `get_and_drive_accel()` task uses the `uvm_accel` interfaces to send and receive data items as transactions from the HVL partition into the HDL partition where a hardware BFM drives the DUT signals. The `get_and_drive_accel()` task does not implement any signal-level protocol functionality it operates purely at the transaction level. The HDL BFM is implemented as a separate module and is instantiated in the HDL hierarchy partition which will be described in the next section.

The `uvm_accel` ports use standard transaction-level modeling (TLM) semantics to send and receive transactions by way of SCE-MI communication channels. The code snippet below shows the blocking put and blocking get tasks being used to send and receive data items.

```
task yamp_master_driver::get_and_drive_accel();
   forever begin
      seq_item_port.get_next_item(req);    // Get new item from the sequencer
      m_ip.put(req);                       // Drive the item
      if(req.direction == READ) begin
```

```
            m_op.get(req);
            seq_item_port.item_done(req);
        end
        else begin
            //Communicate item done to the sequencer
            seq_item_port.item_done();
        end
        <rest of implementation>
    end
endtask
```

Once a data item has been taken from the sequencer, it can be put into an input channel using the blocking `put()` function associated with the port that is bound to that channel. The blocking `put()` function blocks until the transaction has been taken from the channel at the opposite end. This means that the `get_and_drive_accel()` task does not need to implement any sort of *wait* before informing the sequencer that the current sequence item has been done. This is simpler than in the non-accelerated case where you must implement any code required to allow one sequence to be completed before the next one is started.

5.5.2.5 Acceleratable Driver BFM (SystemVerilog)

The acceleratable driver BFM resides in the HDL partition and implements the signal level protocol functionality required to drive the DUT. The acceleratable driver BFM contains SCE-MI pipes interfaces which are bound to ports within the driver component that resides in the HVL partition. The `get_and_drive_accel()` task passes transactions through a SCE-MI pipe to the driver BFM which must extract the transaction and apply it to the DUT signal level interface.

The acceleratable driver BFM must be written in acceleratable SystemVerilog or Verilog for it to be accelerated by a hardware accelerator. The driver code should be partitioned into separate files to reflect code that is to be simulated and code that is to be accelerated. This simplifies the overall compilation process and makes the code easier to maintain.

Note The Cadence UVM Acceleration package provides ***e*** and SystemVerilog interfaces to allow access to the pipes on the HVL side. Therefore, the same acceleratable driver BFM can be used in both environments.

Each driver BFM must instantiate appropriate SCE-MI pipes ports to mirror those defined in the driver's proxy which exists in the HVL partition. If the ports at each end of the communication channel are not compatible, they will not be bound and elaboration will fail. The code snippet below, taken from the `yamp` example, shows a SCE-MI input pipe called `inbox0()` and a SCE-MI output pipe called `outbox0()`.

```
module yamp_master_driver_bfm (
   input   wire       clk,
   output  reg        cmd,
   output  reg[7:0]   len,
   output  reg        we,
   output reg         ce,
   output  reg        rd,
   output  reg[15:0]  addr, di,
   input   wire[15:0] dout,
```

```
   input   wire       scemi_mode
);                                          // SCE-MI input pipe instantiation
    scemi_input_pipe #(2, 1) inbox0 ();  // SCE-MI output pipe instantiation
    scemi_output_pipe #(2, 1) outbox0 ();
<rest of implementation>
```

Both `inbox0` and `outbox0` have the parameters `BYTES_PER_ELEMENT` set to `2` and `PAYLOAD_MAX_ELEMENTS` set to `1`.

`BYTES_PER_ELEMENT = 2` means that each message element received will contain two bytes.

`PAYLOAD_MAX_ELEMENTS = 1` means that only one message element will be received at a time.

These two parameters define the width of the data that can be received by an input port or sent by an output port. Each ports width is defined by the parameter `PAYLOAD_MAX_BITS` that is defined as shown in the following formula:

```
PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;
```

Therefore, the ports in the example above are capable of receiving or sending messages only 16-bits wide during each transfer.

A UVC can contain different types of driver to suit the level of abstraction used to model the DUT. If a simulation-based driver and an acceleratable driver have both been implemented, it is important to ensure that only one driver drives the DUT at any one time. The `abstraction_level_enum` should be used to define the value of `scemi_mode`. When the SCE-MI hardware acceleratable driver is to be used `scemi_mode` should be set to `1`; for all other scenarios, `scemi_mode` should be set to `0`. This is usually defined at the top level of the UVC. The code snippet below, taken from the `yamp` example, shows that the output `we_r, ce_r`, and `rd_r` are tri-stated, unless `scemi_mode` has been set to 1.

```
// Output tri-state logic
   always@(we_r or scemi_mode) we <= scemi_mode3we_r:1'bz;
   always@(ce_r or scemi_mode) ce <= scemi_mode3ce_r:1'bz;
   always@(rd_r or scemi_mode) rd <= scemi_mode3rd_r:1'bz;
   <rest of implementation>
```

The SCE-MI pipes HDL API provides blocking and non-blocking tasks and functions. The code snippet below, taken from the `yamp` example shows how the blocking `receive()` task is used.

```
always@(posedge clk) begin
   if(scemi_mode) begin
      inbox0.receive(1, num_recv, idata, eom);
      {len_r, delc, ws, cmd_r} = idata;
<rest of implementation>
```

At the positive edge of the clock called `clk`, the `receive()` task associated with `inbox0` is called with the following arguments:

```
Num_elements = 1
Num_elements_valid = num_recv
Output_data = idata
EOM = eom
```

`Num_elements` defines how many elements are to be put into the variable idata when a transaction has been received. This example deals with one message element at a time. A transaction can contain many message elements, and the BFM designer needs to decide the most efficient implementation.

`Num_elements_valid` defines the number of received elements that are valid. This can be used by the BFM to determine the elements to be used when multiple elements are received in one transfer. This is not relevant in this example because only one element can be received at one time.

`Output_data` defines the variable in which received data will be written into. The width of this variable should be defined by `PAYLOAD_MAX_BITS` as described above.

`EOM` defines whether the message element received is a single message element or a part of a continuous stream of message elements. Using `EOM`, it is possible to send transactions that contain a variable number of message elements during each transfer. When `EOM` is set to `1`, the element received is the last element. When `EOM` is set to 0, there are more elements available to read.

More information about the SCE-MI hardware API can be found in the *Standard Co-Emulation API: Modeling Interface (SCE-MI) Reference Manual*.

5.6 Building Acceleratable UVCs in *e*

5.6.1 Data Items

Data items are transactions that are implemented as struct objects that derive from `any_sequence_item`. A data item contains data members, constraints to constrain any data members that are to be randomized, and methods for manipulating the data members or the struct itself. The code snippet below, taken from the ***e*** `yamp` example, shows the `struct` definition of a data item called `transfer_s` along with its data members.

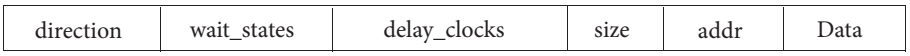
```
struct transfer_s like any_sequence_item {           // transfer_s struct
       %direction    : yamp_direction_t;             // Memory access
                                                     // direction
                                                     // (READ OR WRITE)
       %wait_states  : uint (bits : 3);              // Used by the driver
                                                     // to insert wait states
       %delay_clocks : uint (bits : 4);              // Used by the driver
                                                     // to insert a transfer
                                                     // delay
       %size         : uint (bits : 8);              // Size of data transfer
       %addr         : yamp_addr_t;                  // Start address of
                                                     // memory access
       %data         : list of uint (bits : YAMP_DATA_WIDTH);   // Data to be
                                                     // read or written to
                                                     // memory
```

Data items that contain randomly assigned data members require constraints to constrain the range of values they will be assigned. Constraints can be defined within the struct definition as shown below or in a separate constraints file.

```
keep soft data.size() == size;
keep direction == WRITE  =>  data.size() == size;
keep soft size > 0;
keep soft size < 10;
```

To transfer a data item from the proxy in the HVL partition to the BFM in the HDL partition, the data members must be packed into a vector of bits as shown in Figure 5-7 below.

Figure 5-7 Packed Implementation of Data Item yamp_transfer



e provides built-in `pack` and `unpack` methods to create a list of bits that is a concatenation of the members contained in the data item `struct`. The acceleratable driver must understand the packing scheme used in order to extract each member from the data item received.

5.6.2 Acceleratable Driver (*e*)

The UVC BFM is responsible for taking data items from the sequencer and driving them onto the DUT interface. The DUT can be modeled at multiple levels of abstraction. So, the BFM must be able to accommodate each of the interfaces presented by each type of model. This not only affects the type of physical interface used, it also affects the functionality of the BFM itself. To be able to reconfigure the BFM to operate at different levels of abstraction, an enumerated type `uvm_abstraction_level_t` is used. This enumerated type is defined in the `uvm_accel` package provided by Cadence.

In ***e***, the enumerated type is defined as follows:

```
type uvm_abstraction_level_t : [UVM_SIGNAL, UVM_TLM, UVM_ACCEL]
                                    (bits : 2);
```

The values defined by this type configure the UVC to operate at one of the following levels:

- Pure simulation at the signal level (`UVM_SIGNAL`)
- Pure simulation at the transaction level (`UVM_TLM`)
- Use hardware acceleration (`UVM_ACCEL`)

When configured for hardware acceleration, an acceleratable transactor is used to bridge the gap between the components that operate at the transaction level and the signal level. Transaction-level components are executed by the software simulator, and signal-level components are executed by the hardware accelerator. The same acceleratable transactor can be used in a simulation-only environment as well as with hardware acceleration. However, multi-purpose UVCs that are configured to operate in `UVM_SIGNAL` mode typically implement the BFM in behavioral ***e*** code. This implementation can continue to be used for simulation to allow a gradual migration to hardware acceleration, if required. When using the behavioral BFM, the `uvm_abstraction_level_t` should be set to `UVM_SIGNAL`. If the UVC is to be used to verify abstract SystemC TLM models, the `uvm_abstraction_level_t` should be set to `UVM_TLM`. The behavior of the driver along with the interface that it uses to connect to this model, should be customized to suit this type of model.

One of the main features of ***e*** is that it provides aspect orientation. This means that objects can be extended to accommodate new functionality or manipulate existing functionality. For UVM Acceleration it is common for the different abstraction levels to be implemented by extending existing units.

The following code shows the ***e*** code which defines the part of the driver that resides in the HVL partition for the `yamp` example.

```
extend UVM_ACCEL master_bfm {
       keep hdl_path() == "xi0";               // HDL path
       m_ip : uvm_accel_input_pipe_proxy of transfer_s is instance; // Input
                                                                    // Port
       keep m_ip.hdl_path() == "inbox0";
       m_op : uvm_accel_output_pipe_proxy of transfer_s is instance; // Output
                                                                   // Port
       keep m_op.hdl_path() == "outbox0";
       m_in :  interface_port of tlm_put of transfer_s is instance; // Input
                                                                    // Port
       m_out : interface_port of tlm_get of transfer_s is instance; // Output
                                                                   // Port
       connect_ports() is also{                                    // Port
                                                                   // Binding
       m_in.connect(m_ip.m_in);
       m_out.connect(m_op.m_out);
       };
       drive_transfer (cur_transfer : transfer_s)                 // Drive
                                                                  // transfer
                                                                  // method
};
```

The `master_bfm` extends the generic BFM and is extended further when the abstraction level is set to `UVM_ACCEL`. This example shows two `uvm_accel` pipe proxy interfaces, `m_ip` and `m_op` that are used for hardware acceleration.

Two `uvm_accel` pipe proxy interfaces are required for the `yamp` example since bidirectional communication is required. Each `uvm_accel` pipe proxy interface is unidirectional; hence, the need for one input interface and one output interface. For most protocols, bidirectional communication is required. So, it is typical for two or more interfaces to be instantiated. Each `uvm_accel` pipe proxy interface takes a data item type as a parameter.

Standard UVM *methods* must be defined for each driver. These methods are customized using extensions depending on the abstraction level. In the `yamp` example the `drive_transfer()` method implements the driver functionality. Separating the code into distinct abstraction levels makes the code easier to understand and debug.

5.6.2.1 drive_transfer Method

When the `master_bfm` is extended to operate in `UVM_ACCEL` mode, acceleratable interfaces are used to send and receive data items as transactions from the HVL partition into the HDL partition where a HDL BFM drives the DUT signals. The `drive_transfer()` method does not implement any signal level protocol

functionality; it operates purely at the transaction level in this mode. The HDL BFM is implemented as a separate module and is instantiated in the HDL partition that will be described in the next section.

The `uvm_accel` ports use standard transaction-level modeling (TLM) semantics to send and receive transactions via SCE-MI communication channels. The blocking `put()` and blocking `get()` functions are shown in the code snippet from the ***e*** `yamp` example below.

```
drive_transfer (cur_transfer : transfer_s)  @p_sys_smp.clk is only {
                cur_transfer.start_transfer();           // Get item from
                                                         // sequencer
          if (cur_transfer.direction == WRITE) {
                m_in$.put(cur_transfer);                 // Drive write
                                                         // transaction
          }
          else if(cur_transfer.direction == READ) {
                var ref_data : list of uint (bits : YAMP_DATA_WIDTH) =
                cur_transfer.get_data().copy();
                cur_transfer.data.resize(0);             // reset the
                                                         // data
                m_in$.put(cur_transfer);                 // Drive read
                                                         // transaction
                m_out$.get(cur_transfer);                // Get read data
          };
          cur_transfer.end_transfer();                  // End current
                                                        // sequence
};
```

Once a data item has been taken from the sequencer it can be put into an input channel using the blocking `put()` function associated with the port that is bound to that channel. The blocking `put()` function blocks until the transaction has been taken from the channel at the opposite end. This means that the `drive_transfer()` method does not need to implement any sort of `wait` before informing the sequencer that the current sequence item has been done. This is simpler than in the non-accelerated case where you must implement any code required to allow one sequence to be completed before the next one is started.

5.6.2.2 Acceleratable Driver BFM (*e*)

The acceleratable driver BFM resides in the HDL partition and implements the signal level protocol functionality required to drive the DUT. The acceleratable driver BFM contains SCE-MI pipes interfaces which are bound to ports within the driver component which resides in the HVL partition. The `drive_transfer()` method passes transactions through a SCE-MI pipe to the driver BFM that must extract the transaction and apply it to the DUT signal level interface.

The acceleratable driver BFM must be written in acceleratable SystemVerilog or Verilog in order for it to be accelerated by a hardware accelerator. The driver code should be partitioned into separate files to distinguish between the code that is to be simulated and the code that is to be accelerated. This simplifies the overall compilation process and makes the code easier to maintain.

The UVM Acceleration package provides ***e*** and SystemVerilog interfaces to allow access to the pipes on the HVL side. The same acceleratable driver BFM can be used in both environments.

For more information about the acceleratable driver BFM, see Section 5.5.2.5, "Acceleratable Driver BFM (SystemVerilog)," on page 213.

5.7 Collector and Monitor

The collector and monitor components have a similar implementation to the driver and sequencer components described in the previous sections, except that the collector and monitor observe and track activity on the DUT interface rather than drive it.

The collector component is responsible for making the physical connection to the DUT and should use `abstraction_level` to determine the kind of interface that should be built during the UVM `build_phase` simulation phase in a similar fashion as previously described for the driver.

The main difference between a collector and a driver is that a collector is a passive component. It does not drive values onto the DUT interface. Therefore, it does not need to be impacted by the tri-stating of any of the signals. Apart from this, a collector should be architected and partitioned in a similar fashion to a driver.

5.8 Summary

Simulation performance can slow down to unacceptable levels when scaling the verification run to the chip or system level. Yet, the demand keeps rising to run such simulations to establish a higher level of confidence in the quality of the product being verified. The acceleratable Universal Verification Methodology (UVM) allows portions of a standard UVM environment to be accelerated using a hardware accelerator. In fact, the methodology does not restrict its usage to hardware acceleration alone. UVM acceleration is truly an extension of the standard simulation-only UVM, and is fully backwards compatible with it. This means that Universal Verification Components (UVCs) architected to be acceleratable can be used in either a simulation-only environment or a hardware-accelerated environment.

This chapter shows how UVM users can build acceleratable UVCs in either SystemVerilog or ***e***. It describes how the UVC agent can be architected to operate in simulation as well as hardware acceleration. The underlying technology is compliant with the Accellera SCE-MI (Standard Co-Emulation API: Modeling Interface) standard providing additional vendor neutrality to the UVM community. In addition, the methodology is compliant with advanced verification techniques such as metric-driven verification, allowing the user community to further build additional verification intelligence into their verification arsenal.