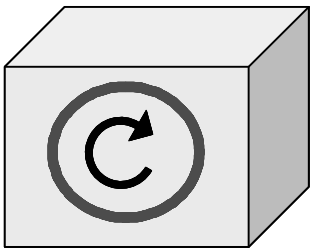
3 TRANSACTION GENERATOR, COMMAND TRANSACTOR, AND MONITOR



A transactor is a generic term used to identify components of the verification environment (i.e., classes) that interface between two levels of abstractions for a particular system/design (e.g., a particular protocol or the generation of transactions). A transactor can be defined as something that executes or observes transactions over time. Transactors can be stopped, started and reset. Several components can be identified as "transactors" including transaction generators to generate transactions; command transactors to drive a hardware interface; and monitors to collect information off a hardware interface.[1] The lifetime of transactors is static in the verification environment: They are created at the beginning of the simulation and stay in existence for the entire duration. They are structural components of the verification environment (implemented with SystemVerilog classes). It is important to note that the implementation of a transactor with a class offers many advantages over an implementation with a module or an interface. Specifically, unlike a module/interface, a class enables the definition of constraint blocks, allows inheritance, can turn ON/OFF the constraints, and can select the randomization modes of individual properties. In addition, as addressed in Chapter 5 and 6, a class allows the use the factory and callback patterns.

This chapter addresses various types of transactors in a VMM framework with focus on transaction generators, primarily atomic. This chapter introduces the concept of scenario, and custom generators, but delays the discussions of those generators to chapters 7 and 8. Other topics addressed in this chapter include the command transactor (a.k.a.BFM) and monitor

[1] Note: Scoreboards perform the verification between what is observed and what is expected. Scoreboards are not considered transactors because they typically perform their tasks in zero-time, and are not extended from any VMM base class.

Typically, only a handful of transactors get created. In contrast, transactions have a dynamic lifetime: Thousands of transactions get created during a simulation run by generators (which are themselves transactors), flow through other transactors (e.g., command transactors or BFMs), get recorded with monitors (also transactors), get compared in scoreboards, and then freed.

Transactors can be classified as:

1. **Proactive transactors:** Those are transactors that generate stimuli (i.e., initiate transactions) under control of the timing of the stimuli. An AMBA bus master is an example of such transactors. Another example is a segmenter that breaks down large packets into smaller ones, and passes those smaller packets to other transactors.
2. **Reactive transactors:** Those are transactors that react to the environment, thus they also generate stimuli, but in response to some event of interface activity. A slave BFM is an example of a reactive transactor: it provides ACK/NACK/DATA to transaction initiated by the DUT.
3. **Passive transactor:** This transactor could be a monitor that collects information from the DUT interfaces and passes that information to a scoreboard for verification. The monitor is addressed in Section 3.3.

3.1 TRANSACTION GENERATOR

In its simplest form, a transaction generator is a transactor that creates transactions as per the definition and constraints of the transaction class, and puts the created transactions into an output channel. Figure 3.1 demonstrates the placement of the generator in the context of the testbench. Generators can be classified as: atomic, scenario, and custom.

To better understand an atomic generator, it is important to understand the structure of a generator. To achieve this goal, we'll describe a simple custom generator without the bells and whistles. Chapter 7 extends that design to one with greater functionality and flags. We'll then address the generator supported by VMM macros, which greatly simplify and speedup the design of transaction generators that meet most of your needs. However, we elaborate the discussion of custom generators in the event your needs are so unique that you may want to build your own.

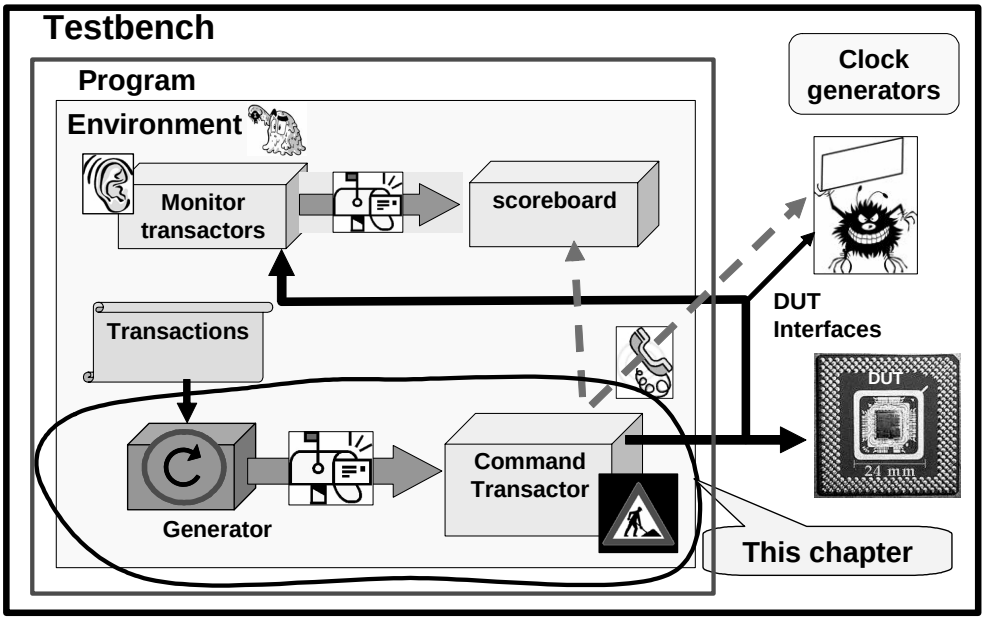


Figure 3.1 Generator in the context of the testbench

3.1.1 Atomic generator

3.1.1.1 Simple atomic generator

Conceptually, an atomic transaction generator generates individual instances of a specified transaction class, and puts those instances into a channel for consumption by another transactor. Specifically, an atomic generator first randomizes an object of the transaction class, and then puts a copy (i.e., a newly allocated copy version) of that transaction into the output channel. Figure 3.1.1.1-1 provides a UML view of a user defined FIFO Transaction Generator.

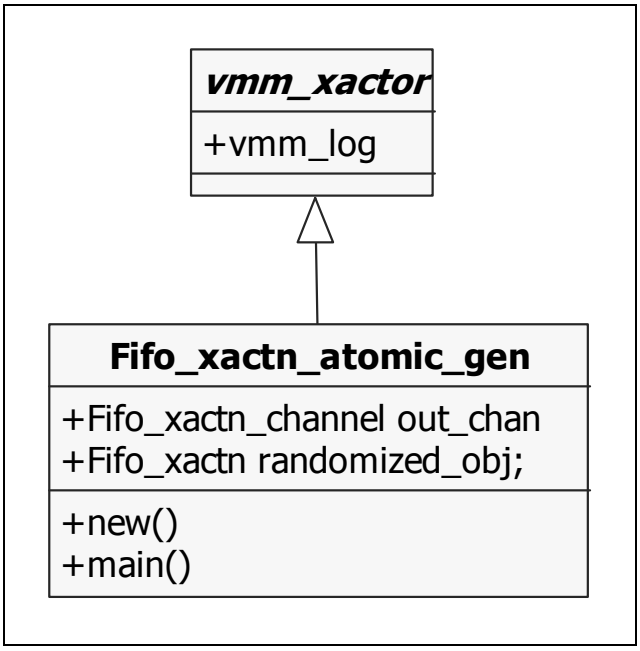


Figure 3.1.1.1-1 UML View of a Simple FIFO Transaction Generator

Figure 3.1.1.1-2 shows a functional flowchart of an atomic generator. The code shown in Figure 3.1.1.1-1 represents the core of the code that emphasizes the randomization of the local transaction (*randomized_obj*), and then the insertion of a copy of that randomized transaction into the local channel (*out_chan*).

```
class simple_gen extends vmm_xactor;
 //..
  task main();
  //Looping construct // defined by the requirements
      randomized_obj.randomize();
      $cast(inst, randomized_obj.copy(randomized_obj));
      this.out_chan.put(inst); // randomized_obj and  inst are of type Fifo_xactn
    //end of looping
  endtask : main
endclass : simple_gen
```

Figure 3.1.1.1-1 Core Code for Randomization of the Local Transaction (*randomized_obj*)

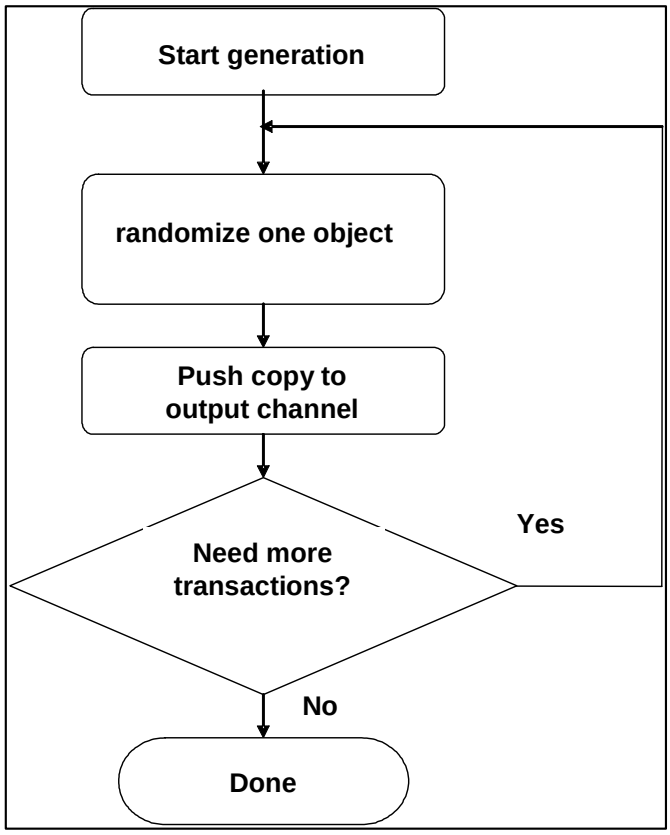


Figure 3.1.1.1-2 Flowchart of an Atomic Generator

3.1.1.2 `vmm_atomic_gen

The fastest, and recommended method to implement an atomic generator is to use the VMM macro: **`vmm_atomic_gen(class_name, Class_Description)**
That macro defines an atomic generator class named <*class_name*>*_atomic_gen* to generate instances of the specified class. In our FIFO example, we have in file *fifo_gen_xactor.sv* the *`vmm_atomic_gen* statement for the generation of the class *Fifo_xactn_atomic_gen* as shown in Figure 3.1.1.2-1.

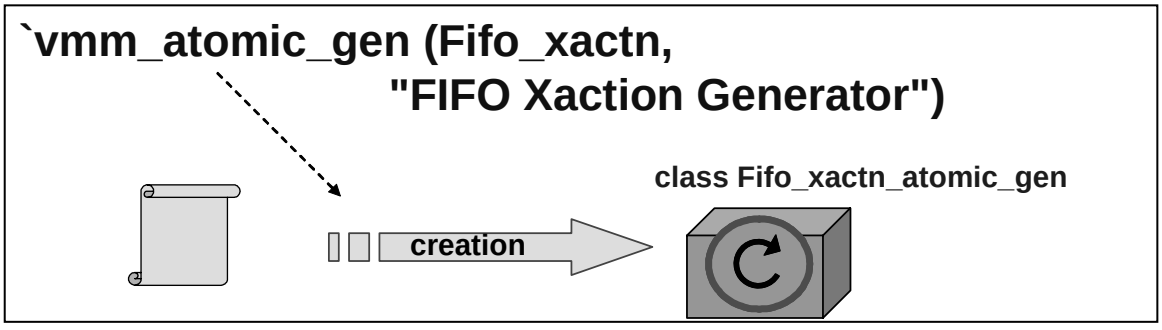


Figure 3.1.1.2-1 Application of the macro `vmm_atomic_gen

Figure 3.1.1.2-2 is a UML view of the VMM atomic transaction generator.

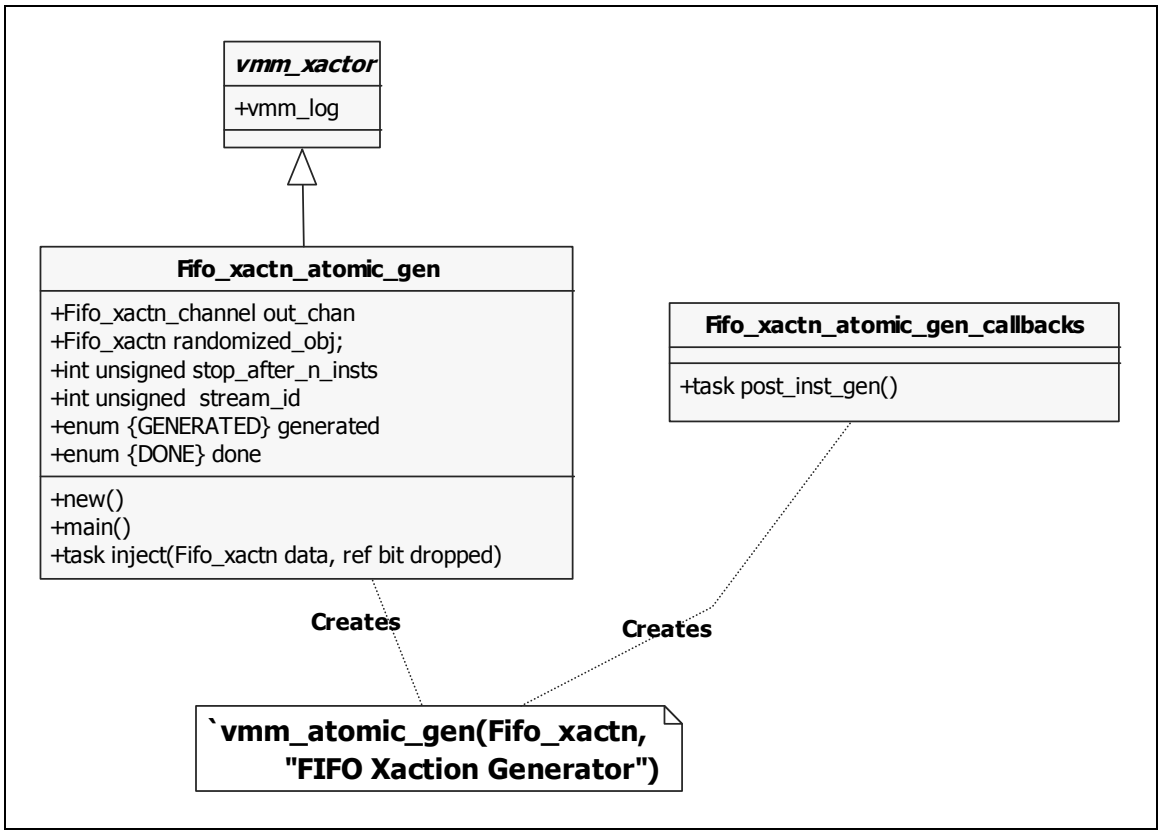


Figure 3.1.1.2-2 UML Elaborate View of a VMM Atomic Transaction Generator

The generator provides the constructor shown below:

```
function new(string inst,
             int stream_id = -1,
             Fifo_xactn_channel out_chan = null);
```

This function is used in the environment to create a new instance of the *Fifo_xactn_atomic_gen* class with the specified instance name and optional stream identifier.

Each atomic generator can be thought of generating a stream of transactions and can be associated with a unique *stream_id*. The generator can be optionally connected to the specified output channel. If no output channel instance is specified, one will be created internally in the *Fifo_xactn_atomic_gen::out_chan* property. In our FIFO environment (file *ch3/fifo_env.sv*) the allocation of the atomic generator is as shown in Figure 3.1.1.2-3. In addition, the association between the generator *out_chan* and the channel instance is performed in the *build* function via the constructor argument. This discussion of the environment is expanded in Chapter 4.

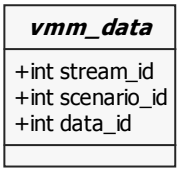
```
class Fifo_env extnds vmm_env;
 Fifo_xactn_atomic_gen fifo_xactn_gen_0;
 Fifo_xactn_channel fifo_channel_0; // variable declaration
 ..
 function void Fifo_env::build();
    super.build();
     // instantiation of channel
```

```
    this.fifo_channel_0 = new("fifo_chan","channel");
    // Allocation of command-layer transactor
    this.fifo_cmd_xactor_0 = new("cmd_xactor",
                         0,
                         `TOP.f_if,
                         fifo_channel_0
                         );
  // allocation of atomic generator
  this.fifo_xactn_gen_0 = new ("fifo_gen", 0,
                                fifo_channel_0);
…
 endfunction : build
endclass : Fifo_env
```

Figure 3.1.1.2-3 Environment for linkage of Channel to Generator and Command-Layer Transactor (*Ch4_fifo/fifo_env.sv)*

In addition to creating the random transactions as per the properties of the transaction class (i.e., the *rand* data members and constraints), the generator puts additional information into the class properties of the transactions that it generates. However, to better understand these actions, you need to understand the properties of *vmm_data*, from which the transaction class extends.

The base class *vmm_data* has three properties used for identification. These properties are defined within the environment for different purposes.



UML for key properties of *vmm_data*

1. The *data_id* is a simple counter that is incremented by the atomic generator at every generation of a transaction. It will be reset to 0 when the generator is reset and after the specified maximum number of instances has been generated. This *data_id* count allows the testbench to read the number of generated transactions and make decisions or reports based on that count.
2. There is one *stream_id* per instance of a generator. The *vmm_data::stream_id* property of the transaction instance is set to the generator's *stream_id* before each randomization. Basically, this mechanism allows the tagging or association of every transaction with its generator. Thus, if a system has multiple generators a transactor receiving (i.e., getting) those transactions can make callback decisions based on the source of the generator (Callbacks are addressed in Chapter 6). An example of such a decision can be an error injection.
3. The *scenario_id* is addressed in Chapter 8.

Because these IDs uniquely identify each transaction instance, like a serial number, they are very useful as a debugging aid during the analysis of a simulation run with hundreds of transactions because they can help track the transaction flow across the system.

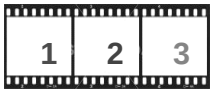
The macro for the atomic generator provides more useful features than what is already addressed, including the termination and notification of the end of the generation of transaction. As shown in the generic generator flow chart in Figure 3.1.1.1c, one of the fundamental controls any user expects in a generator is "how many transactions to generate". The VMM generation macro declares a class property named *stop_after_n_insts* (meaning: STOP after *n* number of transaction instances). When the user-specified number of transactions is reached, the generator indicates the

DONE notification used by the environment to know when the generator has completed the creation of transactions onto the channel. The environment (addressed in Chapter 4) can wait for this end of transaction generation with the following statement:

```
this.fifo_xactn_gen_0.notify.wait_for
                    (Fifo_xactn_atomic_gen::DONE);
```

The generator provides the DONE notification using the *vmm_notify::indicate* function. Notification is explained in Chapter 7. Another feature of this atomic generator macro is the automatic declaration and implementation of a callback class (*fifo_xatcn_atomic_gen_callbacks*) that implements a façade for the atomic generator callback. This callback includes a method *(virtual task post_inst_gen)* that is invoked by the generator after a new transaction or data descriptor has been created and randomized, but before it is added to the output channel. This could be used to hook up a functional coverage model to qualify the generator's randomness, for instance. We address the discussion of callbacks in Chapter 6.

3.1.1.3 Scenario generator



A scenario is "a sequence of random or directed stimulus that is particularly interesting to the device under test. A scenario is unlikely to be spontaneously generated with individually constrained-random stimulus. Multiple scenarios are applied to the device under test during a single simulation". We can consider a scenario as a sequence of atomic operations that are organized in a specific order. Scenario examples for a microprocessor bus could be:

1. READ IDLE IDLE (RD_I_I)
2. READ IDLE WRITE WRITE (RD_I_WR_WR)
3. LOAD IDLE IDLE (LD_I_I)
4. READ READ READ (RD_RD_RD)

Thus, unlike an atomic generator that generates random operations, a scenario generator creates random sequences made up of atomic operations organized in a specific sequence. VMM defines a scenario generator as one that generates scenarios in random order, and produces a stream of transactions that correspond to the generated scenarios. How can random scenario generators be designed? There are several methodologies including:

1. **Atomic generator that use an atomic choice of possible streams or sequences**. This is similar to the atomic generator (with the *`vmm_atomic_gen* macro), but instead of selecting an atomic operation (e.g., READ), the selections are streams (e.g., RD_I_I, RD_I_WR_WR, LD_I_I, RD_RD_RD). Those streams are then decomposed into individual atomic operations by a functional transactor
2. **Custom generator that uses the *randsequence***. This is addressed in Chapter 7. However, an example of the *randsequence* is shown below:

```
 randsequence (stream)
stream : first second third; // := 10 | second := 20 | third :=1;
  first  : idle | load_data;
  second : idle :=2 | enable_count := 7 | reset:= 1;
  third  : idle:=8 | enable_count :=2;
  idle         : {idle_task();}; // kind=IDLE;
  load_data    : {load_data_task();};
  enable_count : {enable_count_task();};
  reset        : {$display("reset, display only");};
endsequence
```

3. **Use of the VMM macro `*vmm_scenario_gen*** along with iterative constraints on the transactions to create scenarios. This approach with the use of constraints is explained in Chapter 8.

3.2 COMMAND-LAYER TRANSACTOR

A transactor uses transactions to do something. For example, the simplest command-layer transactor gets a transaction descriptor from a channel, decodes it, and converts the transaction job to appropriate signal level activity as per the design specification (in our FIFO model, this would mean asserting the correct signals such as *push, pop, data* etc.). VMM recommends that the command layer transactor is actually independent of the actual DUT signal names to foster reuse and recommends usage of a SystemVerilog *virtual interface* instead.[2] This virtual interface is hooked up to the actual interface at the environment level, as shown in Figure 3.1.1.2-3 (connecting to the DUT is discussed in more detail in Chapter 4). Command-layer transactors are often known as BFMs.

In the FIFO example, the command-layer transactor is responsible to get the generated transactions off the transaction channel, and translate them to stimuli on the FIFO interface, such as PUSH and POP control signals. We're also demonstrating the use of transaction completion information back to the generator via a response channel.[3] The generator could use inline code or callbacks to handle the completion information, such as do a retry in the event of a failure (Callbacks are addressed in Chapter 6). Figure 3.2-1 demonstrates a UML architecture of the command-layer transactor in the environment.

.

[2] VMM Rule 4-108 Physical interfaces shall be specified using a virtual *modport* interface as an argument to the transactor constructor.

[3] VMM Suggestion 4-139 Consumer transactors may use a different descriptor to return transaction completion information.

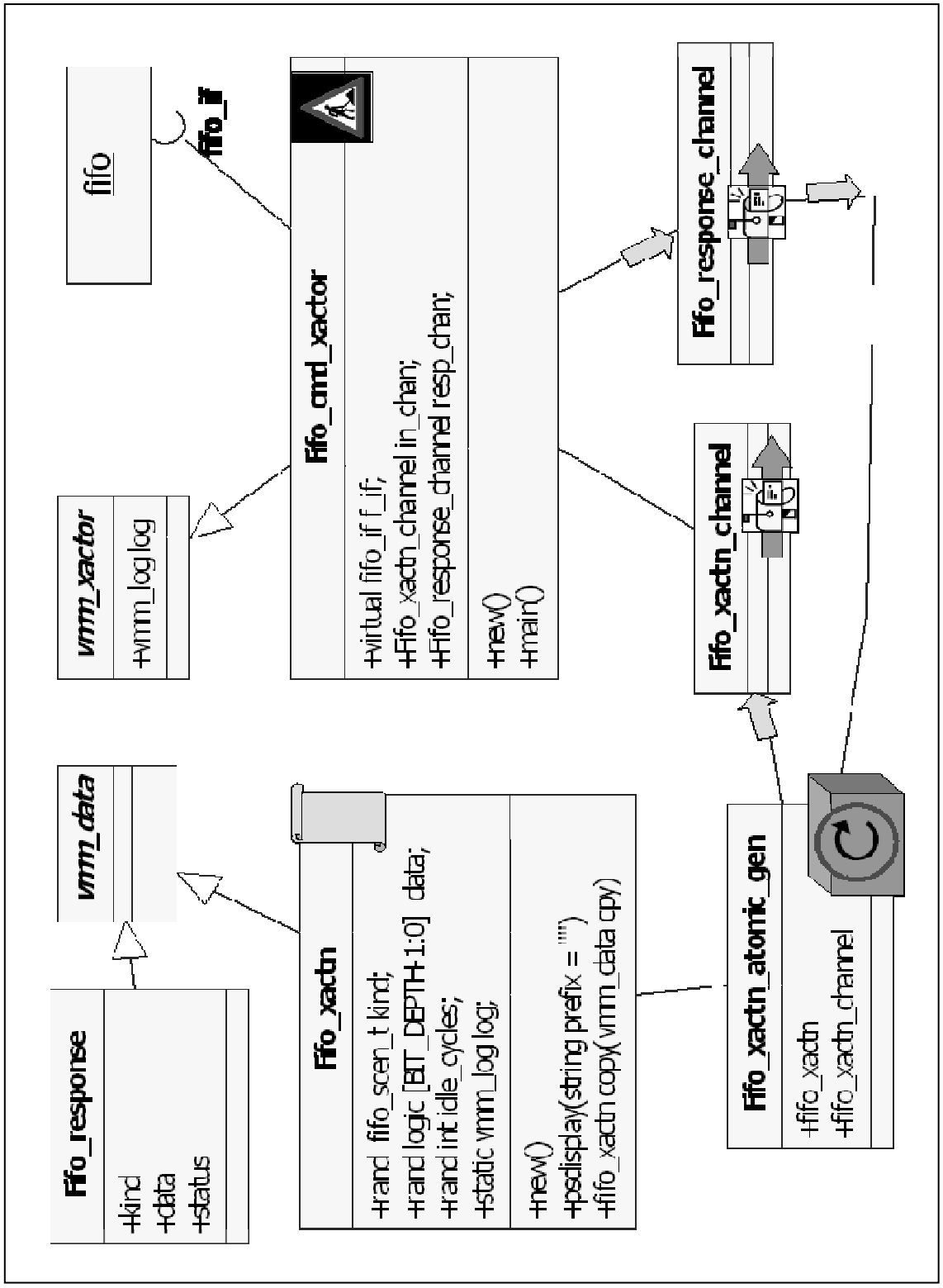


Figure 3.2-1 UML Architecture of FIFO Transactor in the Verification Environment

The key elements of this command-layer transactor architecture include the following:

1. **Class properties**: The properties declared at the top level of the transactor include:
   a. **Virtual interface:** The virtual interface is used instead of direct hierarchical signal references to support reuse and adaptability to a testbench with multiple instances of the interface. An example of this application may be a change to the requirements where the FIFO subsystem must be redundant, thus needing two command transactors (each with possible variations) and two instances of the actual FIFO interface. The declaration of that class property is shown below and makes use of the driver *modport* .

```
virtual fifo_if.fdrvr_if_mp f_if;
```

   b. **Log:** This is needed for the reporting of information. This *log* originates from the *vmm_xactor* base class and is allocated in its constructor.

   c. **Transaction channel:** The channel is the data interface mechanism used by command-layer transactor for fetching (i.e., *get*) the transactions created by the transaction generator. The variable declaration of the channel is: "`Fifo_xactn_channel in_chan;`". Figure 3.1.1.2-3 demonstrates the linkage of the channel instance to the Generator and Command Layer Transactor at instantiation of the transactor.

   d. **Fifo_response_channel:** The response channel, a.k.a. completion channel, is responsible for collecting information off the DUT interface, determining the success or failure of the transaction, and reporting this acquired knowledge back to the generator.[4]

2. **Methods:** The transactor methods include the following:
   a. **new():** This method is called in the environment when the transactor is constructed. The *new* function in our FIFO transactor looks as shown in Figure 3.2-2:

```
class Fifo_cmd_xactor extends vmm_xactor;
  import fifo_pkg::*;
  virtual fifo_if.fdrvr_if_mp f_if;
  Fifo_xactn_channel     in_chan; // input channnel for transactions
  Fifo_response_channel resp_chan;  // output responsse channel

  function new(string inst,
         int unsigned stream_id = -1,
         virtual fifo_if.fdrvr_if_mp  new_vir_if,
         Fifo_xactn_channel      new_in_chan=null,
         Fifo_response_channel   fifo_response_channel=null);
      super.new("cmd_xactor", inst, stream_id);
      this.f_if = new_vir_if;
      if (new_in_chan==null)
         this.in_chan=new("fifo_chan","channel");
      else this.in_chan = new_in_chan;
      if (fifo_response_channel==null)
        this.resp_chan=new("fifo_response_chan", "channel");
      else this.resp_chan=fifo_response_channel;
  endfunction : new
```

[4] VMM Rule 4-137 —Consumer transactors shall use the vmm_channel::sneak() method to add completed transaction descriptors to the completion channel.

```
  extern task main();
  extern task reset_task (int num);
  extern task push_task (word_t data);
  extern task pop_task();
  extern task push_pop_task (word_t data);
  extern task idle_task (int num_idle_cycles);
endclass: Fifo_cmd_xactor
```

Figure 3.2-2 Methods in the Command-Layer Transactor *(ch4_fifo/fifo_cmd_xactor)*

The *new* function of the transactor must call the *vmm_xactor*::*new* (or *super*.new). The *new* function also makes the association between the arguments of the interface and channels to the local interface declaration and channel declarations. If a channel is *null* then a new channel is instantiated.

b. **main():** Figure 3.2-3 demonstrates the code for the FIFO *main* task. The *main()* task is managed by the *vmm_xactor* base class and is called when the environment invokes the *start_xactor()* method. In our example, this is performed in the environment as

```
this.fifo_cmd_xactor_0.start_xactor();   // in environment
```

This model of *main* follows the rules described under "completion and response models" using the "in-order atomic execution" described in VMM. This completion and response modeling allows the BFM to inform the higher layer (in this case the generator) that the transaction has been executed and with what results.

Key features of this code are:

1. **Call to *vmm_xactor main()*** task (with *super.main()*), as required by the VMM. [5]
2. **Declaration of the transaction variable**, in our case the *Fifo_xactn fifo_xactn_0.* Note that we're only declaring the variable, but not allocating it. This variable is used to store the transaction information off the transaction channel.
3. **Transaction descriptors peeked from the input channel**.[6] The *peek* task gets a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset without actually retrieving it. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved. Otherwise, the *peek* is non-blocking. The *peek* method keeps the producer blocked from generating transactions into the channel (of size 1) while the current transaction is being executed.
   ```
   this.in_chan.peek(fifo_xactn_0);
   ```
4. ***vmm_data*::STARTED and *vmm_data*::ENDED notifications**. Those indications inform the generator of the status of the command transactor.[7] In Chapter 7, we use this notification to inform the generator of the standing of the BFM transactor. This information may be used by the generator to take action.
   ```
   fifo_xactn_0.notify.indicate(vmm_data::STARTED);
   ```

[5] VMM Rule 4-95 . Extensions of the *vmm_xactor::main()* task shall call *super.main()*.

[6] VMM Rule 4-121 Transaction descriptors shall be peeked from the input channel.

[7] Recommendation 4-123 The vmm_data::STARTED and vmm_data::ENDED notifications should be indicated.

5. **Information to the *vmm_xactor::log*** is provided for debugging and documentation purposes, as shown below: *vmm_trace* is used to avoid getting the message by default.

```
`vmm_trace(log,
$psprintf("Got a new fifo xaction from in_channel %s ",
                          fifo_xacn_0.psdisplay()));
```

6. **Actions based on transaction descriptor.** Once the transaction descriptor is obtained, the command-layer transactor can determine what actions to take based on the properties (i.e., variables) of the peeked transaction. The transaction can easily be decoded and acted upon to assert signals onto the interface. For example,

```
case (fifo_xactn_0.kind)
    PUSH : this.push_task(fifo_xactn_0.data);
    POP  : this.pop_task();
      …
```

The called tasks perform the actual interface bit wiggling. For example, the *task push_task (word_t data);* performs as follows:

```
task push_task (int data);
     begin
        `vmm_debug(this.log,$psprintf("%m Push data %0h ", data));
              f_if.driver_cb.data_in <= data;
              f_if.driver_cb.push <= 1'b1;
              f_if.driver_cb.pop  <= 1'b0;
              @ ( f_if.driver_cb);
            f_if.driver_cb.push <= 1'b0;
           end
        endtask : push_task
```

Note: The clocking block of the driver interface is used in the assignment and reading of the interface signals, as this ensures that the proper setup and hold times are used.[8]

7. **Checks on the success of the execution**. Following the bit-wiggling tasks (e.g., *push, pop*) to the DUT, we can check on the success of the execution of the transaction (e.g., a DUT may have responded with a RETRY). We can also notify the producer of this status. In our simple model, we assumed success with an indication of ENDED.

```
fifo_xactn_0.notify.indicate(vmm_data::ENDED,
                                fifo_response);
// Send the response to generator thru the response channel
// in nonblocking manner.
  this.resp_chan.sneak(fifo_response)
```

8. **Flush the transactor descriptor.** Use the *get* task to flush the already used transactor descriptor from the channel.

```
this.in_chan.get(fifo_xactn_0);
```

[8] VMM Rule 4-12 The clocking block shall be included in *modports* port list instead of individual clock and synchronous signals.

```
task Fifo_cmd_xactor::main();
    Fifo_xactn  fifo_xactn_0;   // transaction to get
    Fifo_response  fifo_response; // response to generator
    fork
      super.main();
    join_none
    forever
    begin : main_loop
      // Rule 4-121
      this.in_chan.peek(fifo_xactn_0);
      // Rule 4-123
      fifo_xactn_0.notify.indicate(vmm_data::STARTED);
      `vmm_trace(log,
         $psprintf("Got a new fifo xaction from in_channel %s ",
               fifo_xactn_0.psdisplay()));
      case (fifo_xactn_0.kind)
        PUSH :
          this.push_task(fifo_xactn_0.data);
        POP  :
         this.pop_task();
        PUSH_POP :
         this.push_pop_task(fifo_xactn_0.data);
        IDLE :
          this.idle_task(fifo_xactn_0.idle_cycles);
        RESET :
          this.reset_task(fifo_xactn_0.reset_cycles);
      endcase
      // Can do checks here if needed
      fifo_response=new();
      fifo_response.kind = fifo_xactn_0.kind;
       fifo_response.status= PASSED;
      // ..
      // Now do a get() to unblock producer
      // Rule 4-123
      fifo_xactn_0.notify.indicate(
            vmm_data::ENDED, fifo_response);
      // Send the response to generator thru the response channel
      // in nonblocking manner.
      this.resp_chan.sneak(fifo_response);
      // Rule 4-121
      this.in_chan.get(fifo_xactn_0);
    end : main_loop
  endtask : main
```

Figure 3.2-3 main Task in FIFO transactor *(ch4_fifo/fifo_cmd_xactor)*

The simulation of this model is addressed in Chapter 4 and 5.

3.3 MONITOR

A monitor is a transactor very similar to a transaction generator with the exception that it extracts a transaction from the observed interface, and puts that newly assembled transaction onto a local channel. Thus, in essence, the difference between a transaction generator and a monitor transactor is in the creation of the transactions to be put into the channel: the transaction generator first randomizes the transactions while the monitor assembles the observed transactions off the interface. Figure 3.3-1 demonstrates this difference.

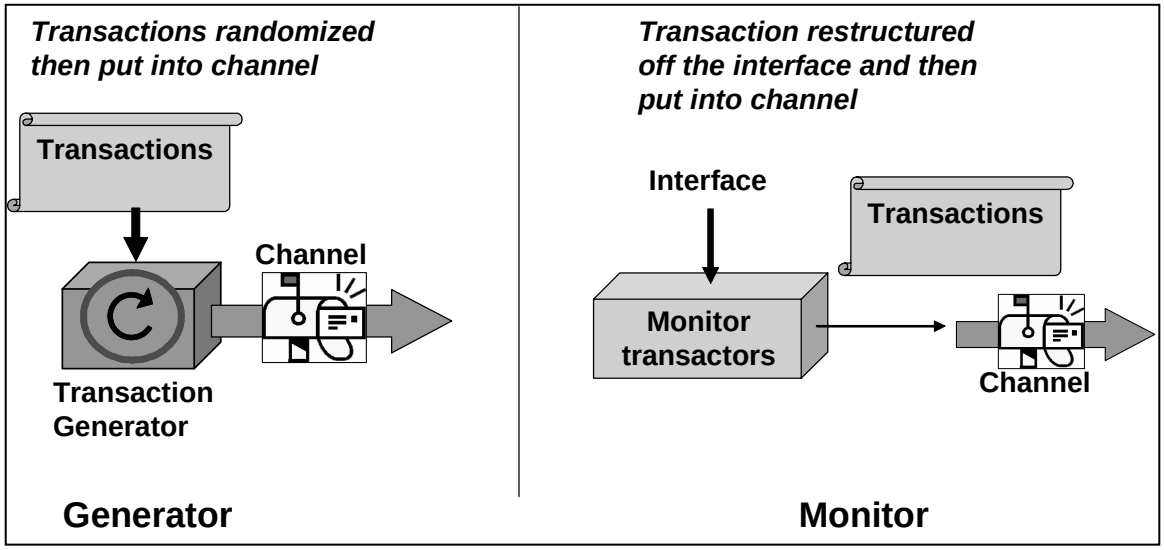


Figure 3.3-1 Difference between a Generator and a Monitor in the Use of Transactions

Figure 3.3-2 represents the UML of a monitor. Figure 3.3-3 represents the monitor code for the FIFO interface.

Notes about our monitor model:

1. We kept a count of the number of observed PUSH transactions. This can be used to detect an end of simulation after a desired number of transactions were observed.
2. In Chapter 5, under the notification section, we expand this model to add notification that a PUSH transaction was detected. This can be used by another model (e.g., scoreboard) to perform additional verification checks.

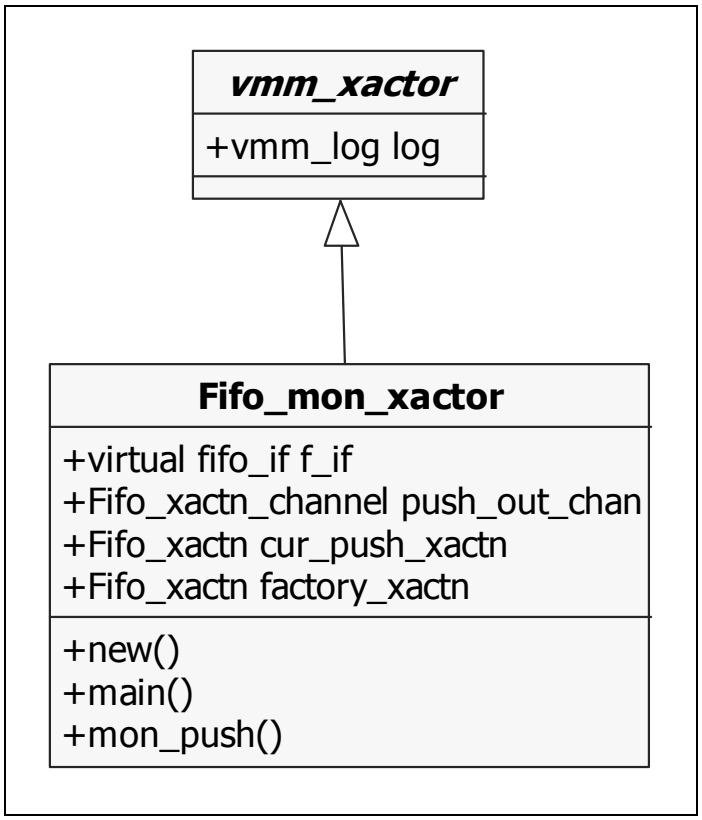


Figure 3.3-2 UML of a Monitor Transactor

```
 class Fifo_mon_xactor extends vmm_xactor;
  virtual fifo_if f_if;
  Fifo_xactn_channel push_out_chan;
  Fifo_xactn cur_push_xactn;
  Fifo_xactn factory_xactn;

  function new(string              inst,
               int unsigned        stream_id = -1,
               virtual fifo_if     new_vir_if,
               Fifo_xactn_channel  new_push_out_chan=null);
      super.new("Fifo Monitor Xactor", inst, stream_id);
      this.f_if = new_vir_if;
      if (new_push_out_chan==null)
          this.push_out_chan=new("Fifo_mon_chan_0","channel");
       else this.push_out_chan = new_push_out_chan;
  endfunction : new

  extern task main();
  extern task mon_push();
endclass : Fifo_mon_xactor
```

```
task Fifo_mon_xactor::main();
    `vmm_trace(log, "Inside Monitor");
    fork
      super.main();
      this.mon_push();
   join_none
  endtask : main

task Fifo_mon_xactor::mon_push();
  string msg;
  forever begin : mon_push_loop
    @(this.f_if.mon_cb);
    if (this.f_if.mon_cb.push === 1'b1) begin
      $cast(this.cur_push_xactn, factory_xactn.allocate());
      this.cur_push_xactn.data = this.f_if.mon_cb.data_in;
      this.cur_push_xactn.xactn_time = $time;
      $sformat(msg,"Found a PUSH Xactn data %0d ",
               this.cur_push_xactn.data);
      `vmm_trace(log, msg);

      this.push_out_chan.sneak(this.cur_push_xactn);
    end // if
  end : mon_push_loop
endtask : mon_push
```

Figure 3.3-3 Simplified Model of a Monitor (Ch4_fifo/fifo_mon_xactor.sv)

Simulation results yielded a log report as shown in Figure 3.3-4.

```
2250.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.3 Fifo Xaction PUSH
2350.00 ns Fifo Monitor Xactor [Trace:DEBUG] | Found a PUSH data 599
2350.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.4 Fifo Xaction
PUSH_POP
2450.00 ns Fifo Monitor Xactor [Trace:DEBUG] | Found a PUSH Xactn data 550
2450.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.5 Fifo Xaction PUSH
2550.00 ns Fifo Monitor Xactor [Trace:DEBUG] | Found a PUSH Xactn data 474
2550.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.6 Fifo Xaction POP
2650.00 ns cmd_xactor [Trace:DEBUG] | Got a new fifo xaction from in_channel  #0.0.7 Fifo Xaction POP
```

Figure 3.3-4 Simulation Results with Monitor Log

File Structure

Table 3.4 demonstrates the file Structure and the purpose of each file. Figure 3.5 is a graphical representation of the relationship between the files for this chapter.

Table 3.4. File Structure and Functions

File	Function	Used by
fifo_pkg.sv	Defines types and parameters	ALL
fifo_if.sv	Defines the FIFO interface	RTL, property models, and by program, testbench, transaction and transactors
fifo_csr_if.sv	Defines the FIFO configuration interface	RTL, property models, and by environment, and possibly transactors
fifo_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Fifo_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface.
fifo_rtl.sv	Represents the FIFO RTL DUT	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind
fifo_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model,
fifo_cmd_xactor.sv	Provides the transactor definition to drive the FIFO model.	FIFO environment
fifo_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel	Scoreboard, top level
Fifo_response.sv	Class for command-layer transactor to compose a response	Command-layer transactor

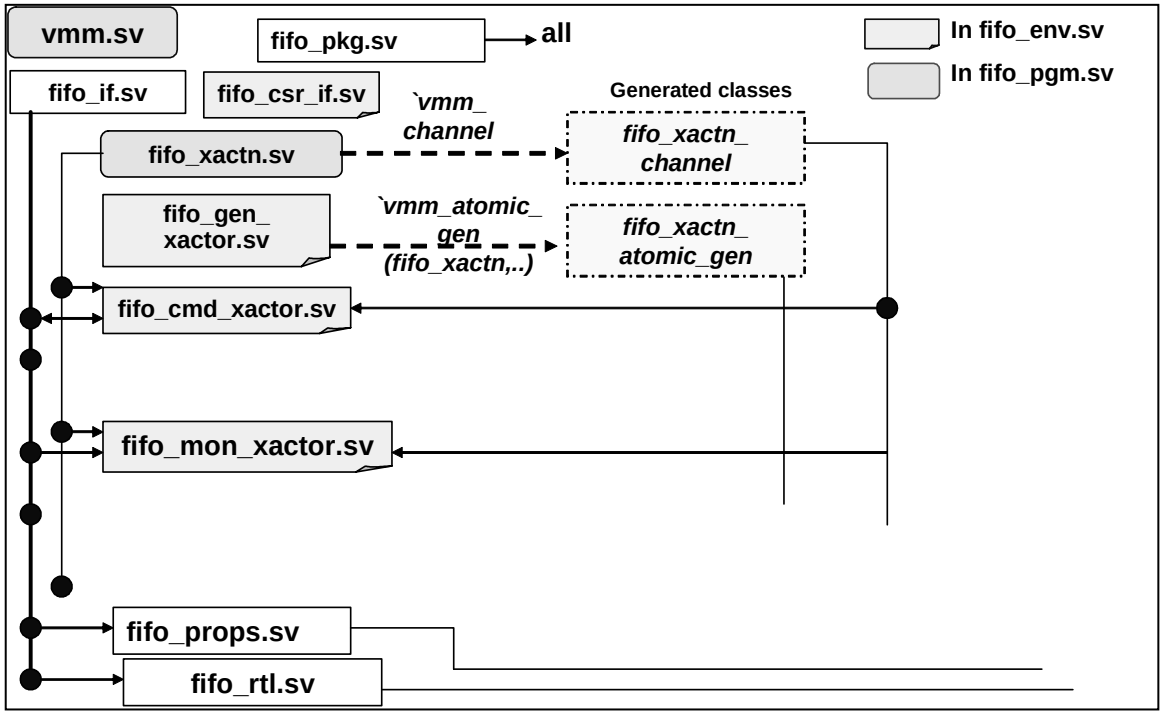


Figure 3.4 File Structure and Relationships

Chapter 3 Questions and LAB

Q1. Why does the transaction generator send the transactions to the channel instead of directly to the transactor?

Q2. Why is a transactor extended off the base class *vmm_xactor*?

Q3. What role does a monitor play?

LAB03
Build a transaction generator using the macro and a command transactor for the counter model. See instructions in subdirectory lab/lab03/todo/readme.txt.