4 BUILDING THE ENVIRONMENT AND TESTBENCH

A verification environment is an encapsulation of various verification components such as drivers, generators, channels, monitors, scoreboards, etc. While the various verification components are independent of each other, it is the environment that brings all of them together to accomplish a given verification task. This chapter introduces the anatomy of a VMM environment and then describes the operation of the environment in detail. Figure 4.0 demonstrates the relationship of the environment with respect to the testbench.

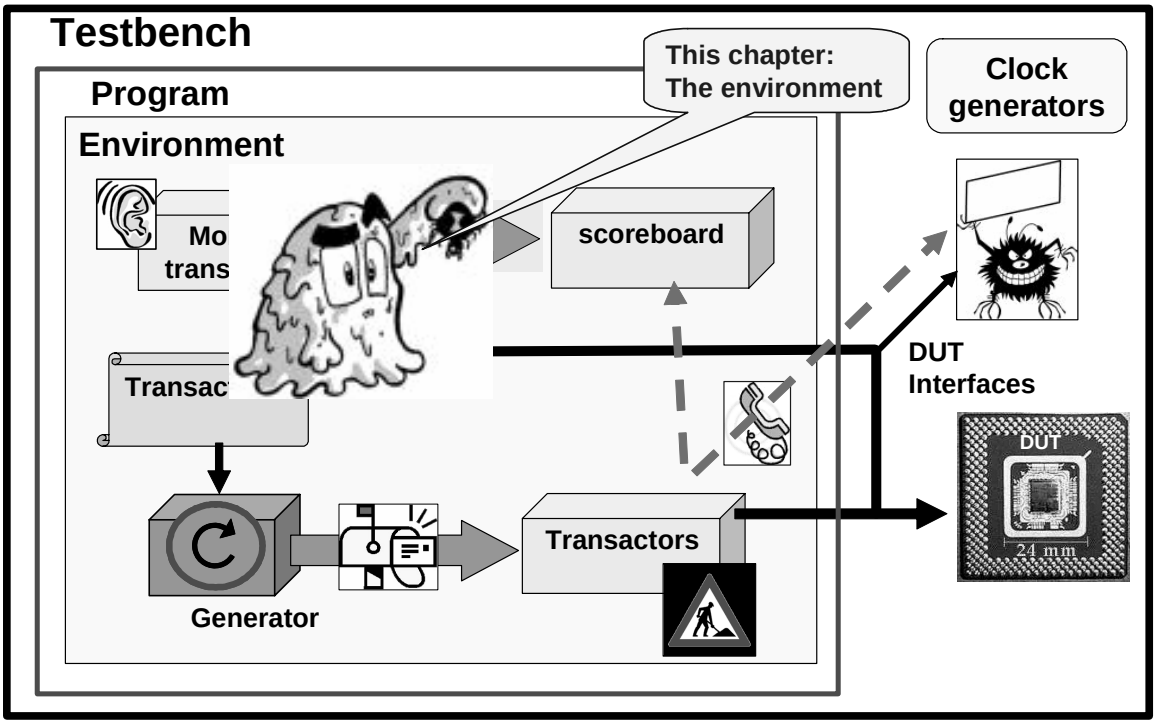

Figure 4.0 Relationship of the Environment with Respect to Testbench

4.1 ANATOMY OF A VMM ENVIRONMENT

Broadly speaking, a VMM compliant verification environment consists of two major segments:

- A **structural segment** that hooks up different verification components,
- A generalized **test flow segment** that controls how all of these different components interact with each other in both spatial and temporal contexts.

4.1.1 Structural Segment

This segment of the environment contains instantiation of various verification components such as:

- Transaction generators (Atomic or Scenario)
- Functional transactors, if needed for the system to emulate hardware resources not yet developed (e.g., mid-level protocol translator, image reformatter, etc).
- Command level transactors - drivers as well as monitor (a.k.a BFMs)
- Channels to hook up different transactors
- Signal layer connections - via SystemVerilog *virtual interfaces*
- Any reactive response generators (slave models)
- Scoreboards
- Functional coverage unit
- Logger

Figure 4.1.1-1 represents the structural segment of the environment that consists of transactors and channels to transmit transactions from the generator to the BFMs; responses form the BFMs to the generator for transfer status information, e.g., success/failure/retry; and monitors for the gathering of data off the DUT interfaces for transfer to a scoreboard for verification.

```
class fifo_env extends vmm_env;
// Fifo transaction class declaration
  Fifo_xactn fifo_xactn;
// command-layer declaration
  Fifo_cmd_xactor fifo_cmd_xactor_0;
// channel declaration
  Fifo_xactn_channel fifo_channel_0, fifo_mon_chan_
// response channel cmd transactor -> generator
  Fifo_response_channel fifo_response_chan0;
// Configuration declaration
  Test_cfg test_cfg_0;
// atomic generator declaration
  Fifo_xactn_atomic_gen fifo_xactn_gen_0;
// monitor declaration
  Fifo_mon_xactor mon_0;
// format control declaration
  Fifo_log_fmt log_fmt_cntl;

   ..
endclass : fifo_env
```

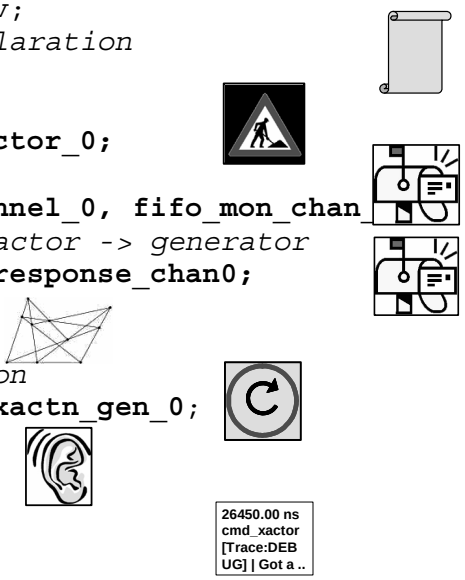

Figure 4.1.1-1 Structural Segment of the Environment (*ch4_fifo/ fifo_env.sv)*

The command transactor needs virtual interfaces to communicate (e.g., read and write) to the signals of DUT. The virtual interfaces enable the reuse of the transactor for interconnection to multiple instances of the actual interfaces. The *vmm_env* hookups such virtual interfaces to actual, design interface. This is done in the test flow segment during the *build()* phase, addressed in section 4.1.2.2.

The constructor (the function *new()*) of the *fifo_env* is shown in Figure 4.1.1-2. The constructor is a good place to perform user preference settings such as:

- Log format control
- Time format control etc.

The construction of the structural components is delayed until the build() phase rather than within the constructor because it facilitates maintenance of the environment. The Log format is done via a VMM class named *vmm_log_fmt*, more on this in Chapter 7. The time formatting can be easily done using Verilog's *$timeformat* system task.

```
  function new();   // for environment
     super.new();
    $timeformat(-9,2, " ns");
    this.test_cfg_0 = new;[1]
    log_fmt_cntl = new();
    log = new("fifo_env", "");
    log.set_format(log_fmt_cntl);
  endfunction : new
```

Figure 4.1.1-2 Constructor of the FIFO Environment (*ch4_fifo/ fifo_env.sv)*

Figure 4.1.1-3 provides a UML view of the main elements of the environment.

[1] VMM Example 4-18. Randomization of Testcase Configuration Descriptor

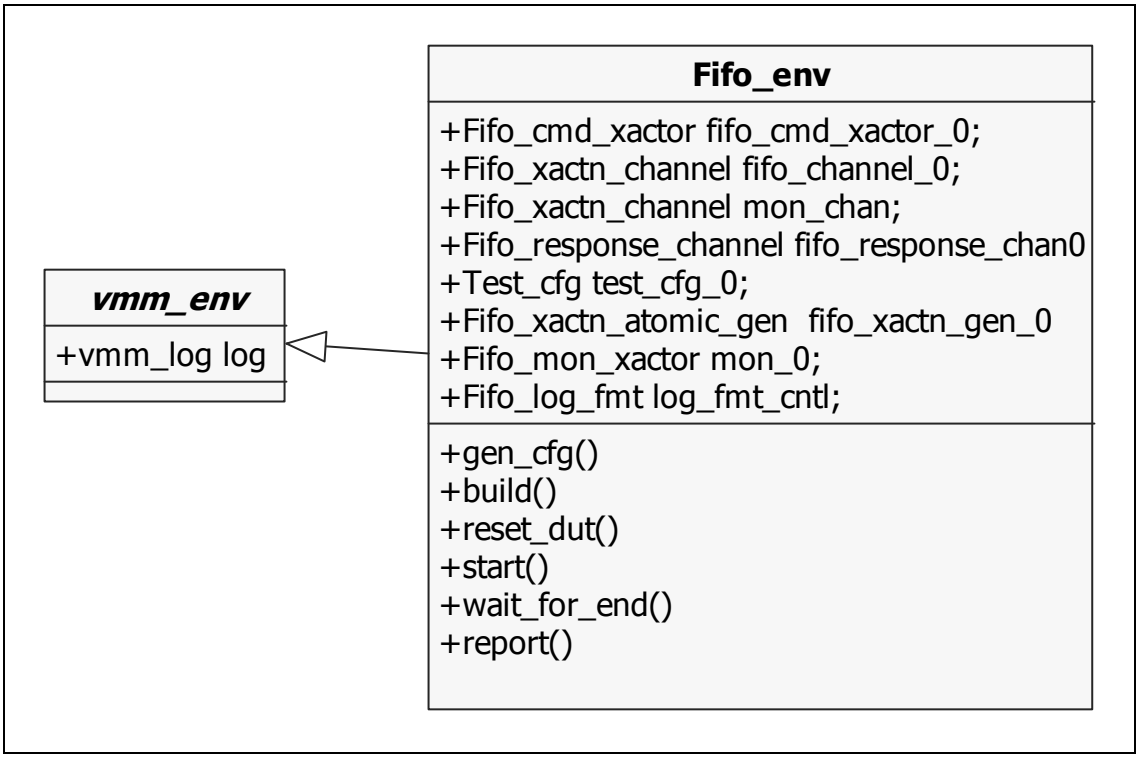

Figure 4.1.1-3 UML of Main Elements of the Environment

4.1.2 Test Flow Section

As with every other component in VMM, the base class *vmm_env* captures the best known practices formulated from a variety of verification environments. One of the most interesting aspects of VMM is its generalized test flow mechanism. Almost every functional test that is run on a design has several distinct sequences of steps, but often that goes unnoticed. For example, a novice engineer trying to verify the FIFO design may develop a testcase as follows:

- Generate the clock
- Reset the DUT
- Configure the registers (optional step)
- Start the transactions - PUSH, POP etc.
- Wait for certain number of clocks
- Finish the simulation

To appreciate the need for the generalized test flow, let's consider a hypothetical networking design.[2] A simple test will look as follows:

- Generate the clock
- Reset the DUT for few clocks
- Configure the DUT registers (if any)
- Simulate few transaction/packets/frames
- Wait for certain number of clocks
- Finish the simulation

[2] We present a sample of such design and a verification environment in Chapter 8.

While the above two flows look very similar, we list both examples to demonstrate the fundamental fact that across a variety of designs, there is an underlying test flow common in skeleton.

In a simple, non-VMM based testbench, the implementation of the above test flow may likely be implemented inside a single (or few) Verilog *initial* block, where the concept of a "sequence of steps" is often implicitly present. VMM standardizes this test flow and recommends a 9-step flow. This flow is not duplicated in every test. It is captured once. These steps are implemented using SystemVerilog virtual methods (tasks or functions inside a *class*). The use of virtual methods allows you to easily annotate and customize a given step for your design. Such a standard flow provides several advantages, including:

1. **Consistency in structure across all designs**
   This is important because in the lifecycle of a design, the verification task undergoes several iterations, and may often be handled by different verification engineers. Thus, as new engineers inherit or review a verification testbench, they carry with them a common understanding of the verification flow standardized by the VMM framework.

2. **Customizable flow plan**
   The steps involved in the verification flow are well outlined and customizable by you, if that customization is necessary. These well thought-out steps do serve as a reminder to you, the verification engineer, and to the code reviewers as to the needed and possibly missed steps in this flow. In essence, "it makes you think". For example, in a conventional, non-VMM flow, a testcase could have forgotten to configure the PCI properly and have instead started the transactions. This could lead to all transactions being aborted by the PCI interface, and a debug engineer might spend several hours looking at the simulation results only to realize that the configuration of the DUT was not done before sending the transactions

3. **Reusability and maintainability**
   With a standard flow, building the environment is relatively easy. Maintaining it and upgrading it is even easier. For example, in our FIFO model we demonstrated various concepts in the different chapters, including the use of different transactions, generators, factories, and callbacks. However, making the necessary changes to build a new environment out of those various components and patterns was very easy

4. **Enhanced debugability**
   With a standard test flow the debugging becomes simpler. For example, if there is a problem in configuring a register, you can quickly look at the *cfg_dut()* method extension and isolate the problem.

5. **Flow control**
   There are two aspects to flow control: User-control and framework control. As a user, you can control and modify the configuration of the environment and DUT. You can also control the start/stop/restart of various transactors. The VMM framework automatically controls the sequencing of the various steps needed to create the verification of the DUT. That framework relieves you from the task to create this flow control.

The generalized test flow is illustrated through a flowchart in Figure 4.1.2 along with the actual method names being used in VMM. The individual steps are further elaborated in the subsequent sections.

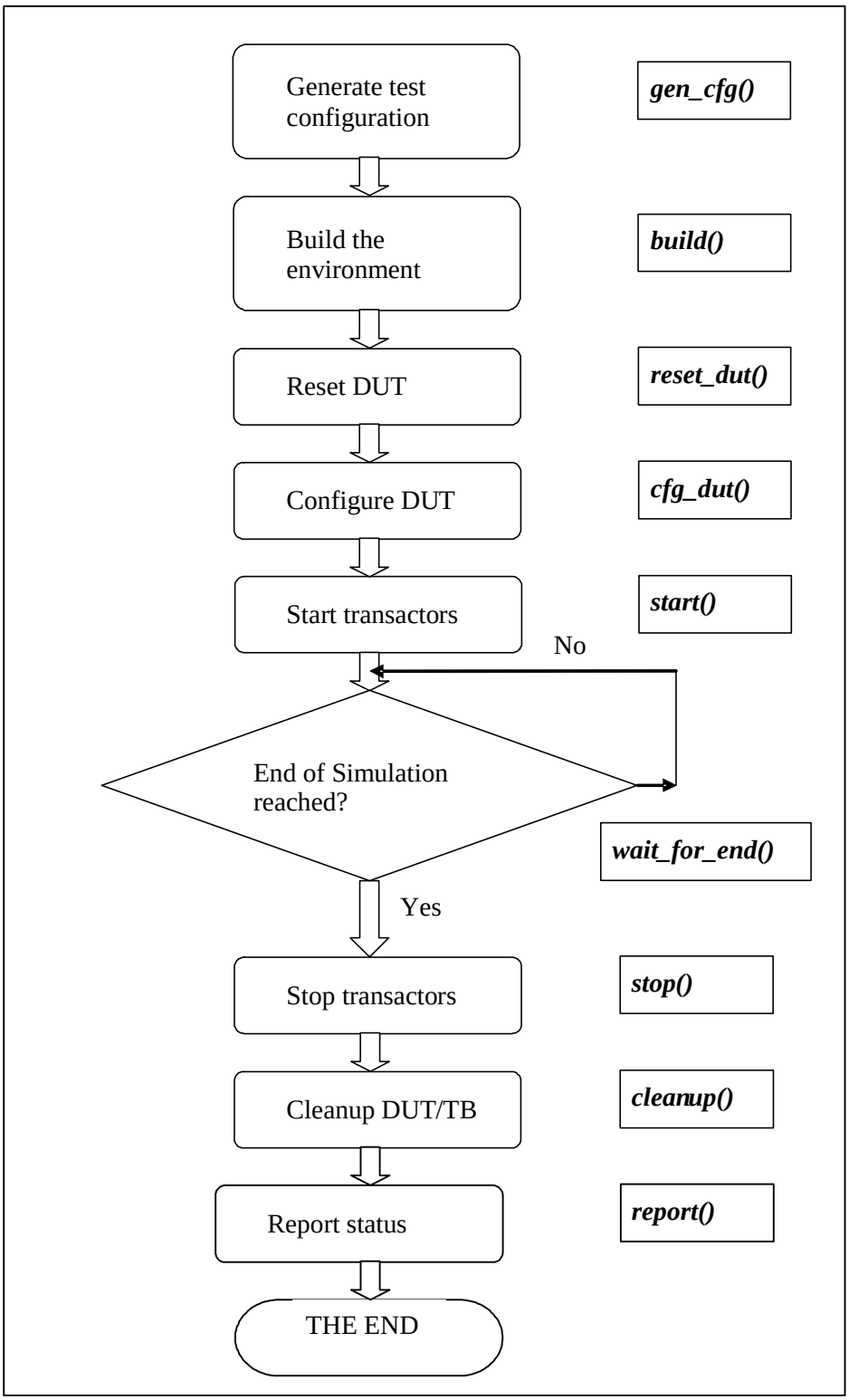

Figure 4.1.2 Test Flow of the Environment

4.1.2.1. gen_cfg()

This generate configuration method creates a configuration of the test environment used by the DUT. The random configuration is encouraged; however, the configuration could also be determined from command-line arguments or an external file or be hard-coded. Typical items that get randomized at this step are:

- Number of input/output ports, and their speed, mode etc. in a configurable IP.
- Number of masters and slaves in a shared bus env. such as OCP.
- Number of transactions, percentage errors etc.

For example, in our FIFO model, we can determine the random values for the various CSRs (Configuration and Status Registers). The goal of this step is that over many random runs, one will test every possible configuration, instead of the limited number chosen by directed test writers. The actual configurations are placed under a custom SystemVerilog *class* known as a "test configuration descriptor". In our FIFO example we used the *Test_cfg* class. An example of a *gen_cfg* that randomizes properties used by the environment and the DUT is shown in Figure 4.1.2.1.

```
class Test_cfg;
  rand bit [7:0] no_of_xactions;
endclass : Test_cfg

class Fifo_env extends vmm_env;
  Test_cfg test_cfg_0; // Test Configuration Descriptor
  …
  extern function void gen_cfg();
endclass : Fifo_env

function void Fifo_env::gen_cfg();
    super.gen_cfg();
    this.test_cfg_0.randomize();
endfunction : gen_cfg
```

Figure 4.1.2.1 Sample configuration generation in *vmm_env* (*ch4_fifo/ fifo_env.sv)*

Essentially this step provides one central place for all random configurations for the environment of the DUT. This centralized location helps in the control and maintenance of that environment. The created configuration of the environment is set in the *build(),* while the DUT detailed configuration is later downloaded during *cfg_dut()* step,

4.1.2.2. build()

This method builds the verification environment per the configuration generated in the previous step. This includes generators, checkers, drivers, monitors etc. In SystemVerilog, this method essentially calls the constructors of various component classes. One question that often arises is "Why not construct these elements as part of the constructor of the *env* itself (i.e., the *new* of the environment)?" - Remember that the verification environment is also a SystemVerilog *class*. There are two main reasons as to why you want to construct the instantiated components in the *build* method:

1. The random configurations are not yet available during the construction
2. This architecture allows for dynamic re-configurability, thus after a specific number of transactions, you may choose to re-configure the environment and re-build it.

Figure 4.1.2.2a represents our build for the FIFO.

```
  function void Fifo_env::build();
  string msg;
    super.build();
    // Instantiation of channels
    this.fifo_channel_0 = new("fifo_chan","channel");
    this.fifo_response_chan0=
         new("fifo_response_channel", "channel");[3]
    // Instantiation of command-layer transactor
    this.fifo_cmd_xactor_0 = new("cmd_xactor",
                         0,
                         `TOP.f_if,
                         fifo_channel_0,
                         fifo_response_chan0
                         );
    // Instantiation of transaction generator
    this.fifo_xactn_gen_0 =
           new ("fifo_gen", 0, fifo_channel_0);

  // Setting up the number of transactions
    this.fifo_xactn_gen_0.stop_after_n_insts =
                     this.test_cfg_0.no_of_xactions;
  // Setting up a message, then issue it
    $sformat(msg, "Sim shall run for no_of_xactions %0d ",
        this.fifo_xactn_gen_0.stop_after_n_insts);
    `vmm_note(log, msg);
    // Instantiation of monitor channel
    this.fifo_mon_chan_0 =
              new("Fifo_mon_chan_0","channel");
    // Instantiation of monitor
    this.mon_0 = new("FIFO Mon", 0, `TOP.f_if,
                   this.fifo_mon_chan_0);

endfunction : build
```

Figure 4.1.2.2a FIFO Build (*ch4_fifo/ fifo_env.sv)*

An important aspect of *build()* is the hookup of the command-level transactor *virtual interface* to corresponding DUT interface. The *virtual interface* is used by command transactors in VMM to drive the transactions to the DUT. By maintaining this virtual interface the transactor can be connected to different actual interfaces of the same type. At the top level testbench, each of these interfaces will be instantiated along with DUT (Refer to Section 4.2 later in this chapter), and that complete hierarchical path to the interface instance is provided to the command level transactor as shown in Figure 4.1.2.2b (e.g., *`TOP.fifo_if_0*). The macro *TOP* is defined in the FIFO package with a *`define* construct that refers to the top level testbench.[4]

[3] The *Fifo_response* represents a transaction for the transfer of status or data back to the generator. This information is useful because a RETRY feedback to the generator should cause a retry of the last issued transaction. This response model is used here for demonstration purposes and could be ignored for simple testbench environments.

[4] A direct path can be used instead of the macro TOP since there is only one top-level.

4.1.2.3. reset_dut()

This step takes care of resetting the DUT. In simple designs this simply means toggling the reset signal and waiting for few clocks. However, complex ASICs may have a well defined reset sequence and may take several hundred clocks to get the proper reset. Some designs get reset when specific values written to configuration registers and/or memory blocks. The *reset_dut* method has the following functionalities:

- Initiate the reset condition
- Wait for the reset sequence to be completed.
- Verify the status of DUT after reset.

Complex SoCs have one or more processors and their boot sequence can also be considered as part of this *reset_dut()* method. On-chip PLLs is another candidate where reset holds the key for the entire chip's operation in normal mode. In PLLs, the time it takes to get to a locked state is usually long in terms of simulation cycles. As seen from the previous examples, the reset process can consume many clock cycles. To speed this process, one could use a FAST_MODE configuration technique to initialize the design via backdoor access, such as the direct load of the memory elements via the direct path.

Figure 4.1.2.3 represents the simple reset for our FIFO model

```
task Fifo_env:: reset_dut();
    super.reset_dut();
    `TOP.reset_n <= 1'b0; // in fifo_pkg.sv:  `define TOP fifo_tb
    `TOP.f_if.pop   <= 1'b0;
    `TOP.f_if.push  <= 1'b0;
    repeat (10) @(`TOP.f_if.driver_cb);
    `TOP.reset_n  <= 1'b1;
    repeat (10) @(`TOP.f_if.driver_cb);
  endtask : reset_dut
```

Figure 4.1.2.3 FIFO Reset Task (*ch4_fifo/ fifo_env.sv)*

4.1.2.4. cfg_dut ()

Once the DUT is reset, the next step is to configure the DUT as per the random configuration generated in *gen_cfg()* step. This is a very important step in real design. Some verification teams tend to combine this step with the *reset_dut()* step. This is not recommended because keeping the two steps separate allows a simulation to run in one configuration for some transactions and then in a different configuration for other transactions.

For example, consider a networking chip with a large address table. The address table is architected to be configured as a linked list for faster search operations. The configuration process is considered done only after the entire table is initialized properly to form a linked list, a process that can take several hundred clock cycles. Once this is done, a specific value is written by the ASIC to a status register. In this case, the address table should be verified for link list consistency (such as all entries being linked, no loops, etc.).

In complex systems such as an Ethernet device, the number of registers to be configured can be large and can take a long simulation time. It is recommended that you provide backdoor accesses for such designs to speed up simulations, using *$readmemh/b* or a hierarchical reference to configuration registers (e.g., top.chip.pci_blk.cfg_0 = 10). Some design teams also prefer using C-code for this step as the C-code can be reused across software and hardware validation. SystemVerilog allows direct C-function calls across C and SystemVerilog language. This is a greatly simplified method over the PLI route of integrating C with Verilog.

4.1.2.5. *start()*

This method starts the test components. Note that some components may need to be started in cfg_dut() to be able to do read/write cycles. At this stage in the test flow, both the design and testbench are configured with the chosen configurations and the design is ready to be simulated with traffic. In a TLM based environment such as VMM, this is usually done by starting the transactors such as generators, driver, monitor etc. VMM provides such a task in the *vmm_xactor* base class named *vmm_xactor::start_xactor()*. It is very important that transactors do not start on their own immediately after construction. Starting may need to be phased in a particular order, depending on the relationships of components. For example, a transactor that emulates a pyrotechnic controller may need to be started after a BFM transactor that drives it. It is very important not to forget to start of a transactor because without a start, it will not be active and will have no effect in the verification process. For example, if you leave out the driver transactor (command transactor, or BFM) from starting, the generator will simply generate transactions, but there will be no consumer or drivers for those transactions. Moreover, since the channel is blocking after the channel level is reached, the generator will wait and the simulation will be running with no useful traffic being simulated. In our FIFO example, the *start()* task in class *fifo_env* calls the *start_xactor()* of individual components/transactors. The *start_xactor* method internally starts the *main* method of its corresponding transactor. Figure 4.1.2.5 demonstrates a snippet of code for *start* task in FIFO model.

```
task Fifo_env:: start();
   super.start();
   this.fifo_xactn_gen_0.start_xactor();
   this.fifo_cmd_xactor_0.start_xactor();
   this.mon_0.start_xactor();
   this.fifo_mon_chan_0.sink(); // flush content of channel [5]
 endtask : start
```

Figure 4.1.2.5 Snippet of start Task for FIFO model (*ch4_fifo/ fifo_env.sv)*

4.1.2.6. wait_for_end()

This step is where the "core" of verification occurs until the end of test. Thus, this is where transactions are generated and driven to the DUT, and where responses are monitored, covered, and verified for correctness. This could also mean that when a simulation fails, the actual analysis of the failure can generally be traced from this step. However, this is only a guideline as errors can be traced back to the configuration or initialization process.

This method waits for the end of the test, and is usually done by waiting for a certain number of transactions or a maximum time limit, or a predefined number of errors to occur. A sample code from our FIFO example is shown in Figure 4.2.2.6-1. In our example, the environment waits for a *DONE* notification generated by the atomic generator.[6]

[5] May also want to flush response channel with "`this.fifo_response_chan0.sink();`"

[6] See Chapter 7 for more information on notification.

```
task Fifo_env:: wait_for_end();
    super.wait_for_end();
    this.fifo_xactn_gen_0.notify.wait_for(
                          Fifo_xactn_atomic_gen::DONE);
   // this. mon_0.notify.wait_for(fifo_pkg.MON_DONE);
   // Not yet implemented, but shown here for documentation
  endtask : wait_for_end
```

Figure 4.1.2.6-1 wait_for_end Task (*ch4_fifo/fifo_env*)

Determining the end of test is sometimes an overlooked step, but as per the authors' experience, this is a very important step - more so with constrained random tests because you cannot simply "wait for 1000 clocks and exit". Depending on the random number of transactions being generated, that number 1000 maybe insufficient to run all the transactions. EOT (End Of Test) detection is quite system/design dependent, and it is recommended that the verification architects bring this process upfront, and provide the necessary hooks in the environment to determine the EOT (e.g., notification, watchdog, number of transactions, coverage value, etc). A simplified flow for EOT detection is provided in Figure 4.1.2.6.

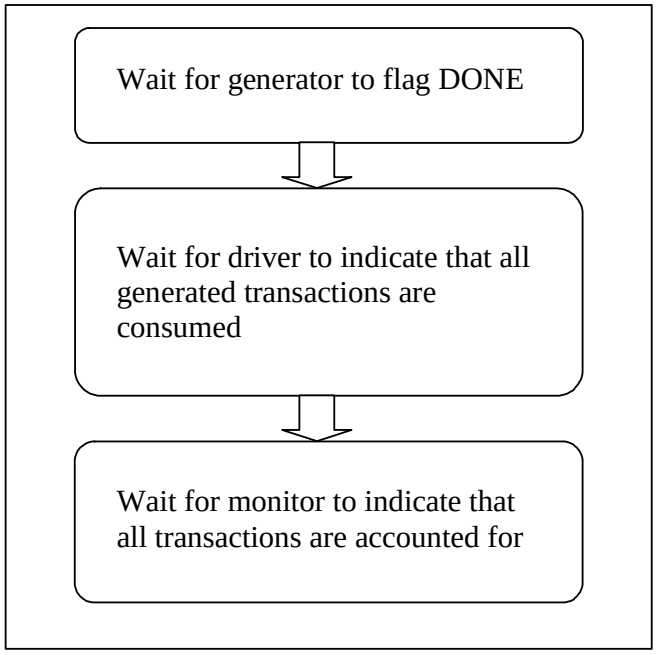

Figure 4.1.2.6 End Of Test Recommended Flow

Note that the above flow chart does not cover many other issues, such as watchdog for inactivity in the DUT, dropped transactions, internal FSM states etc. These are highly system/design dependent. We provide another example that addresses those issues as part of advanced topics in Chapter 7.

Once the EOT condition is detected, you could abruptly end the simulation. However, such an abrupt end might lead to false failures as some of the transactions might still be in progress, and the data checking has not yet completed etc. To be VMM compliant, the recommended approach is to stop all the active elements in the verification environment, clean up the design/environment and then finish. The next three steps in the VMM test flow address the termination of simulation.

4.1.2.7. stop()

This method is used to stop all the components of the verification environment to terminate the simulation cleanly in preparation for the next step. If this method has not been explicitly invoked in the test program, it will be by default invoked by the *vmm_env::cleanup()* method.

4.1.2.8. cleanup()

This method is intended to be a place where the design and testbench is cleaned up before finish. Cleanup of testbench may involve emptying the VMM channels, SystemVerilog Qs (declared in a scoreboard for example) etc. On the design side, this could involve reading interrupt status registers, statistics counters etc.

For example in a networking ASIC with 16 ports, some of the output ports may be configured to be in blocking state during the simulation. The intended test functionality should be tested in such a mode untill *wait_for_end* is reached. However, once the intended functionality has been tested, you may want to re-enable the previously blocked output ports so that all packets inside the DUT can be emptied out cleanly. This step will also be beneficial if you want to run several tests in chain without a HARD RESET in between. Usually this is not required, but once a design is mature enough, some design teams like to run several tests in a chain without applying reset in between. In a sophisticated environment with self-checkers, such a run might raise false alarms if the previous test did not clean up the expected queues. VMM channel provides a method to empty the channel as *vmm_channel::flush()*, as shown in Figure 4.1.2.8.

```
task Fifo_env::cleanup();
    super. cleanup ();
    this.mon_0.out_chan.flush();
    ...
 endtask : cleanup
```

Figure 4.1.2.8 Application of *vmm_channel::flush()*

vmm_env::cleanup() is also an ideal place to do a memory profiling of a simulation run. Typical memory profile statistics indicate areas prone to memory leaks, excessive storage in the system etc. Usually the memory consumption is largest towards the end of simulation[7]. Synopsys' *vcs* provides techniques to analyze/profile memory consumption during a simulation run. For example, *vcs* provides a system task *$vcsmemprof()* that can be called from testbench code at an appropriate point in simulation time to dump out memory profile data. Adding that to *vmm_env::cleanup()* is recommended to check for unusually large memory consumption. For example, in an environment with several SystemVerilog Qs and VMM Channels (with some of them not being consumed because higher layers do not need those transactions), the number of transactions stored in these Queues and channels might be a primary contributor to excessive memory consumption. A memory profile will reveal such code level bottlenecks.

[7] This is however not always true, some times the peak consumption can be in the early simulation cycles.

4.1.2.9. report()

This is the last step in VMM test flow. This method is responsible to declare the test run as a PASS or a FAIL. You may also want to include a final statistics of what the test has achieved to provide a quick summary of the achieved goals. For example, in our FIFO model we provided two text messages, as shown in Figure 4.1.2.9-1..

```
task Fifo_env:: report();
    super.report();
    `vmm_trace(log,
               "This is where additional model info is displayed");
     `vmm_note(log, "**** REPORT ***");
 endtask : report
```

Figure 4.1.2.9-1 Report Task

4.2 TOP LEVEL TESTBENCH/SYSTEM WITH VMM

Once an environment is created using *vmm_env*, the last phase is to instantiate that environment in a SystemVerilog *program.* Hence the top level testbench has essentially three components, as shown in Figure 4.2-1:

1. The SystemVerilog *program* that wraps the VMM environment
2. The DUT instantiation
3. The Clock generators

Figure 4.2 demonstrates an overview of the testbench.

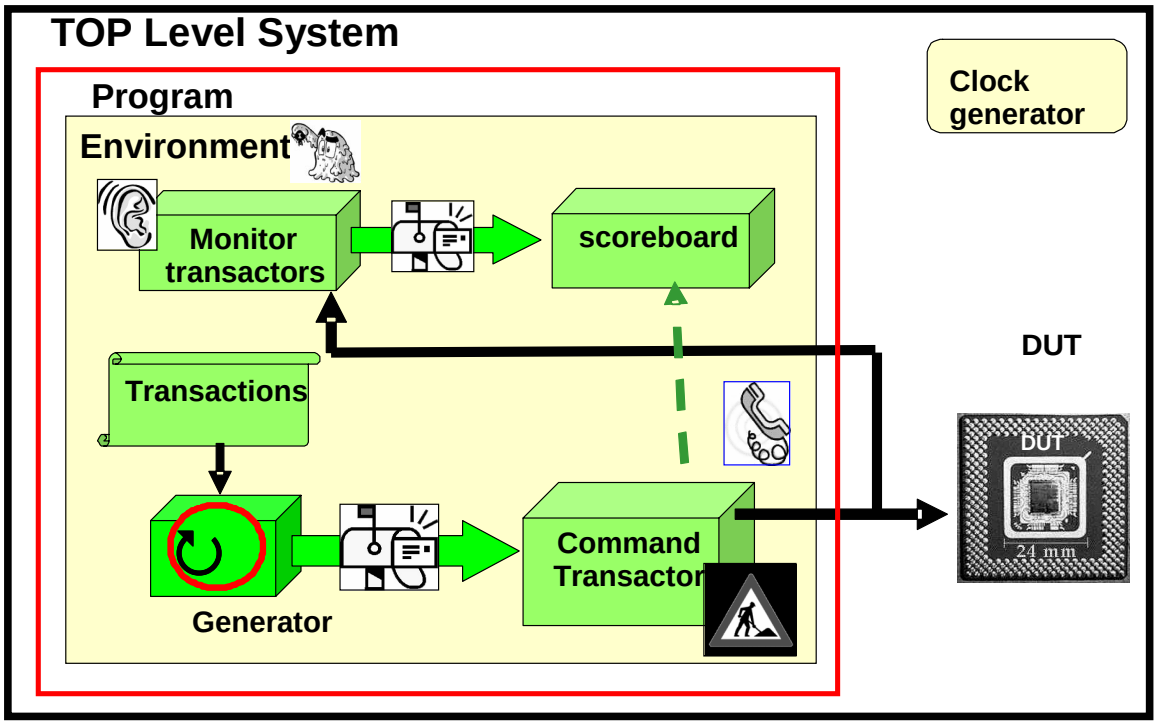

Figure 4.2 Testbench Overview

4.2.1 The Test *program*

As mentioned briefly in Chapter 1, SystemVerilog provides a new construct for testbench entry point - the *program*. For over a decade, design teams have been using Verilog's *module* construct to model both designs and testbenches. While it certainly works, a clear separation from the two, from a language semantic viewpoint, makes a lot of sense - especially because of the infamous race conditions that are inherent in Verilog (and SystemVerilog as it inherits basic Verilog).

The *reset_n* signal can be viewed as either part of the interface or part of a signal at the top level. If the reset signal is handled as part of the interface, then a system with multiple instantiations of that interface will have several reset signals, one for each of the separate interfaces. Since our FIFO model has a single reset signal defined at the top level, we access the reset signal through hierarchical cross module reference. Depending upon your requirements, you need to decide how your reset signal will be handled.

The *program* for our FIFO model is shown in Figure 4.2.1-1. A common mistake is to forget to include the *vmm.sv* file in the program file. This is necessary to have access to VMM. Having the include file within the program instead of in the compilation script also provides documentation for the need of the base VMM classes.

```
program fifo_test_pgm ();
  timeunit 1ns;
  timeprecision 100ps;
  //the include files + log + fifo_env_0 instantiation
  `include "test.svh"
  initial
  begin
     // Build all components of an environment - testbench
     `vmm_note(log,"Start of Test");
      fifo_env_0.build();
      begin
         Fifo_xactn fifo_xactn; // fifo transaction class declaration
         // Instantiation of transaction class
       fifo_xactn = new();
      // Setting the factory for the transaction on the generator (see Ch 5 for factory)
  // ** NOTE: If next line is commented out, then transaction generator
  //       will use its internally instantiated copy of randomized_obj
       fifo_env_0.fifo_xactn_gen_0.randomized_obj =
                                                fifo_xactn;
      // Setting up the factory fifo_xactn for the monitor
      fifo_env_0.mon_0.factory_xactn=fifo_xactn;
       end
     fifo_env_0.run();
     `vmm_note(log, "End of Test");
  end
endprogram : fifo_test_pgm
```

Figure 4.2.1-1 *program* Structure for FIFO Model *(ch4_fifo/fifo_pgm.sv)*

Figure 4.2.1-2 shows the contents of *test.svh* file.

```
`include "vmm.sv"
`include "fifo_xactn.sv"
`include "fifo_response.sv"
`include "fifo_log_fmt.sv"
`include "fifo_cmd_xactor.sv"
`include "fifo_gen_xactor.sv"
`include "fifo_mon_xactor.sv"
`include "fifo_env.sv"
 vmm_log log = new("test", "main");
 Fifo_env fifo_env_0=new();
```

Figure 4.2.1-2 Contents of *test.svh* File (*(ch4_fifo/test.svh*).

As per SystemVerilog LRM, a *program* can contain everything that a *module* can, except the *always* block and module instantiations. Thus, an *initial* block is used to construct the environment and start the test flow.

In SystemVerilog, a simulation is considered finished once the *program* block's execution is completed. This is quite different and new to Verilog users who are accustomed to adding a *$finish* or *$stop* to stop the simulation at the desired point. So in the above example, if the *vmm_env::wait_for_end()* is not properly implemented, you may experience an early simulation termination even without an explicit call to *$stop* or *$finish.* If an unexpected early or late termination occurs, take a close look at the conditions that caused the *wait_for_end* to occur (or never occur).

4.1.3 Clock generation

As a good methodology, clock generation should be done in a *module* and not inside a *program.* To appreciate this guideline, you need to understand the SystemVerilog's event scheduling mechanism. Though there are many regions within the same time step, the ones that are relevant to this discussion are the *active, nonblocking assignment (NBA),* and *reactive* regions. While the classical Verilog *module* assignments get evaluated in the *active/NBA* regions, the newly added *program* block gets executed in the *reactive* region. This scheduling mechanism was designed in SystemVerilog to prevent race conditions originating between the design and testbench. Hence any assignment within the *program* will be visible to the design (*module*) only after the *reactive* region execution. This would mean that the design will go wrong on the combinatorial signal computation. This is obviously not desired. Also, fundamentally a clock is more closely associated with the hardware design than to a high level software-oriented testbench model. Usually the testbench's view of clock is more an abstract view - it is only concerned with the clock edges. Hence it makes lot of sense to retain the clock generation in a *module*. In our FIFO model, we have added that as code at the top testbench.

A question may arise as to how to randomize some of the clock generation parameters, such as clock period, duty cycle etc. This can be achieved at the top level with randomized variables used in the clock generation model. Those variables can be defined in classes.

4.1.4 DUT Instantiation and hook up

The top level module instantiates and connects the following blocks:

- *program* block
- Needed *interfaces*
- DUT
- Clock generator

This is very much similar to any classical Verilog structural model, and hence is not elaborated further here. Please refer to file *ch4_fifo/ top_tb.sv* for details.

4.3 DEVELOPING TESTCASES WITH VMM

In a VMM framework, a testcase is implemented inside a SystemVerilog *program* block. In this section, we will show how you can progressively develop testcases starting from a simple, constrained random test to an advanced test case. One of the benefits of using the *vmm_env* base class to build a verification environment is that the sequencing of these individual steps is maintained under the hood. The user needs to only call the *vmm_env::run()* method and the test sequencing will be automatically taken care of.

4.3.1 Simplest Testcase in VMM framework

The simplest test for our FIFO model looks as shown in Figure 4.3.1-2 It basically instantiates the *fifo_env*, constructs the instance of *fifo_env* as *fifo_env_0*, and simply calls the *run()* method. The *run* method then performs the steps explained in section 4.1.2. The simplest testcase does not include any customization, which is explained in Chapter 5.

4.3.2 Trivial Testcase with just one transaction in VMM framework

While the simplest test looks good for a stable DUT, when the very first test is being run, you may prefer to have a fairly simple, directed-like test case with only one transaction. This is easily achievable in the VMM framework by redefining the number of transactions from within the program, as shown in Figure 4.3.2.

```
program first_test();
  //the include files + log fifo_env_0 instantiations
   `include "test.svh"

  initial begin : b1
    fifo_env_0.gen_cfg();    // from vmm_env::gen_cfg
    // Override the number of xactions field
    fifo_env_0.test_cfg_0.no_of_xactions = 1;
    // Do the rest of the flow as usual
    fifo_env_0.run();
  end : b1
endprogram : first_test
```

Figure 4.3.2 Redefining the number of transactions for simulation

Here, the *fifo_env::gen_cfg()* is called first to generate random test configurations. But since we wanted to override the number of transactions property inside the test configuration descriptor, the value is simply assigned a new value after generation. Once the number of transactions is changed, we want to follow the rest of the test flow as usual. This is done by calling *fifo_env::run()*. Note that since the test case has explicitly called the *fifo_env::gen_cfg()*, the *fifo_env::run()* continues the flow from where it left off (i.e., it shall not re-invoke that *fifo_env::gen_cfg()*) and proceed from the step that follows it - i.e., *fifo_env::build()*.[8] This is very important because otherwise the number of transactions will again get randomly generated.

[8] This flow is also addressed in section 5.2.

4.4 FILE STRUCTURE AND COMPILATION

Table 4.4.demonstrates the file structure and the purpose of each file. Figure 4.4-1 is a graphical representation of the relationship between the files.

The compilation and simulation of the model with Synopsys VCS simulator can make use of the *Makefile* in the *vcs* subdirectory, as shown in Figure 4.4-2. The file *flist* is shown in Figure 4.4-3.

```
env:
        vcs -debug_all -sverilog -f flist +incdir+../ -ntb_opts rvm -R -l ch4_fifo.log
// with trace:
        vcs -debug_all -sverilog -f flist +incdir+../ -ntb_opts rvm -R -l ch4_fifo.log
+rvm_log_default=trace +plusargs_save

run:
        ./simv -gui &
clean:
        \rm -fr csrc* simv* scsim* *vpd ag* session* work/* WORK/* *.so *.log test* cm* ucli*
worklib/* DVE* *.h

pp:
        dve -vpd vcdplus.vpd &
```

Figure 4.4-1. *Makefile* for Compilation with Synopsys VCS Simulator (ch4_fifo/vcs/Makefile)

```
../fifo_pkg.sv
../fifo_props.sv
../fifo_if.sv
../fifo_csr_if.sv
../fifo_rtl.sv
../fifo_pgm.sv
../top_tb.sv
```

Figure 4.4-2. File list used for Compilation (file vsc/flist)

Note that the compilation list does not include all the files used by the testbench. This is because the program file (*test.svh*) included in the program file includes the following:

```
`include "vmm.sv"
`include "fifo_xactn.sv"
`include "fifo_response.sv"
`include "fifo_log_fmt.sv"
`include "fifo_cmd_xactor.sv"
`include "fifo_gen_xactor.sv"
`include "fifo_mon_xactor.sv"
`include "fifo_env.sv"
 vmm_log log = new("test", "main");
 Fifo_env fifo_env_0=new();
```

Table 4.4. 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 program, testbench, transaction and 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_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_log_fmt.sv	Defines formatting information for display.	FIFO environment
fifo_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel.	Scoreboard, top level
fifo_env.sv	Creates the build and start for simulation	program
fifo_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind
fifo_rtl.sv	Represents the FIFO RTL DUT	Top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none
test.svh	Include files needed for compilation	In program
fifo_response.sv	The response transaction from the command transactor to another transactor, such as a generator	Environment

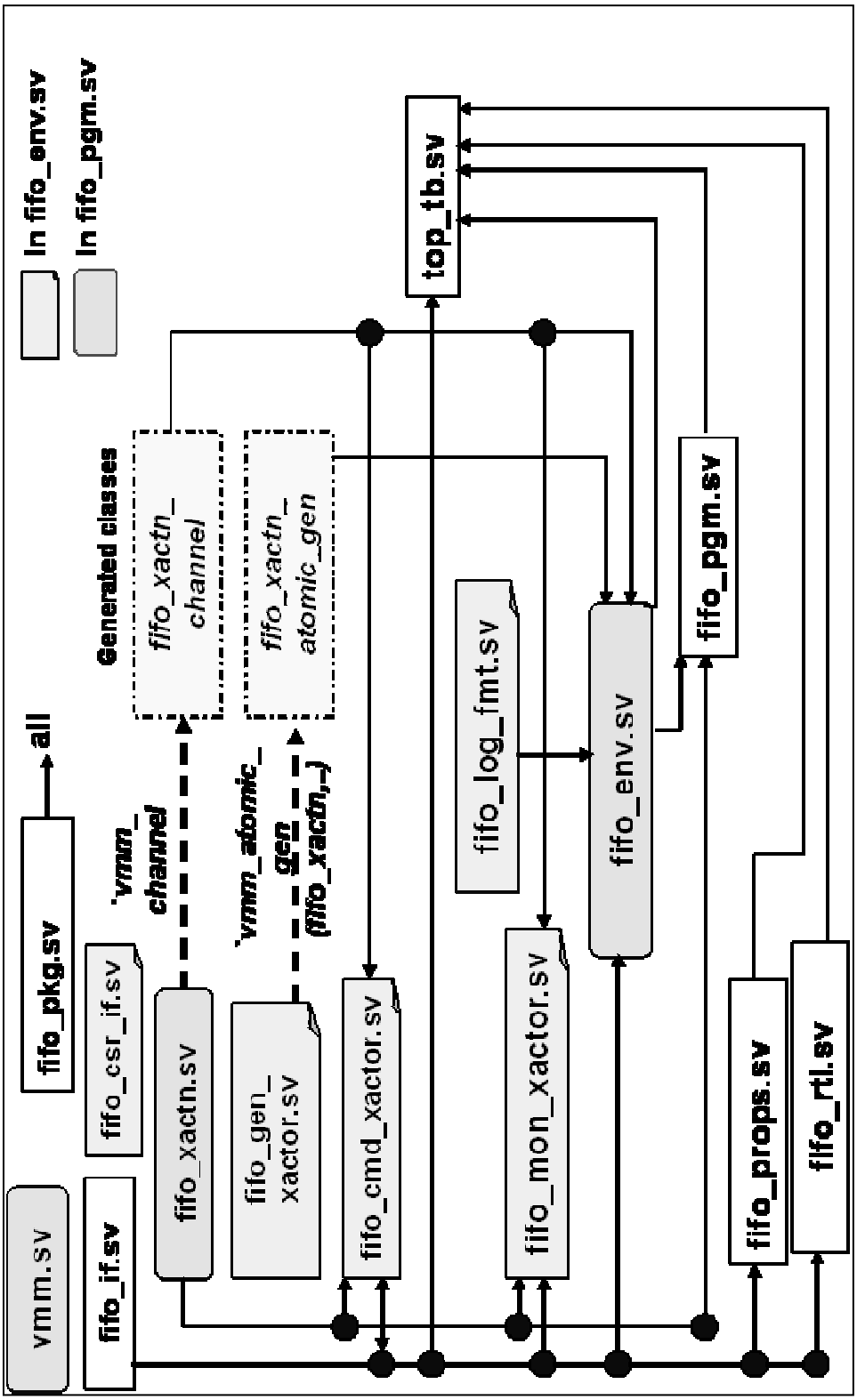

Figure 4.4-3. File Structure and Relationships

Chapter 4 Questions and LAB

Q1. Why is it necessary to build the verification environment in a separate class?

Q2. Why is the environment in a program rather than in a module?

Q3. How is the testflow of the environment initiated? What is the importance of this testflow?

LAB04, Build a verification environment for the counter.
Follow instructions in subdirectory lab/la04/todo/readme.txt