

VMMing a SystemVerilog Testbench by Example

Ben Cohen
Srinivasan Venkataramanan
Ajeetha Kumari

VhdlCohen Publishing / Consulting

ben@abv-sva.org

ABSTRACT

This paper describes a SystemVerilog transaction-based testbench compliant to the Verification Methodology Manual (VMM). It explains by example the VMM methodology in the creation of a comprehensive constrained-random verification environment using a transaction-based approach. This includes generation of transactions and consumption of them via transactors. The paper also addresses through graphical explanations how VMM macros and classes are used in the makeup of a transaction-based verification testbench. The DUT used for this purpose is a synchronous FIFO model with assertions. The testbench models and results are demonstrated. The complete verification model is available for download.

Table of Contents

1.0	Why SystemVerilog for Verification	3
1.1	SystemVerilog Constructs Supporting Verification.....	3
	Table 1.1 SystemVerilog Constructs for Verification.....	3
2.0	Why VMM?	5
3.0	Transaction-Based Verification of a FIFO.....	6
3.1	The DUT	7
3.2	The Testbench	8
3.2.1	Conceptual View.....	8
3.2.2	Testbench Outline	11
3.2.3	The program	11
3.2.4	A generalized test flow mechanism	13
	gen_cfg()	14
	build().....	14
	cfg_dut ().....	14
	start()	14
	wait_for_end().....	14
	stop()	14
	cleanup().....	14
	report().....	15
3.2.5	Transaction class	15
3.2.6	Transactor class.....	16
3.2.7	Creation of Channels.....	16
3.2.8	Generation of Transactions	17
3.2.9	Consumption of transactions from the channels	18
3.2.10	Monitoring of Transactions.....	21
3.2.11	Class Relationships in UML	22
3.2.12	Message Service.....	23
3.3	Simulation Results	24
3.4	File Structure and Compilation	27
4.0	Conclusions and Recommendations.....	29
5.0	Acknowledgements	29
6.0	References	30

1.0 Why SystemVerilog for Verification

SystemVerilog is a rich language that provides constructs needed to support advanced methodologies for verification of today's complex designs. These methodologies include transaction-based verification (TBV), coverage-driven verification (CDV), constrained-random testing (CRT), and assertion-based verification (ABV). Functional coverage can be further divided into temporal coverage (with SystemVerilog assertions (SVA)), and data coverage (with *covergroup*). A good transaction-based verification with CRT relies on constrained randomization of transactions and the channeling of those transactions to transactors for execution (i.e., driving the device under test (DUT) signals for testing). These methodologies can use the collection and access of functional coverage so as to dynamically modify the test scenarios. An adaptation of these methodologies supported by reusable libraries is explained in the book *Verification Methodology Manual (VMM) for SystemVerilog* ^[1]. "VMM Standard Library object code is available today for VCS users. VMM Standard Library source code, which can be used with EDA tools compliant with IEEE P1800 SystemVerilog, is planned to be available for license at no additional charge by VCS users and SystemVerilog Catalyst members before the end of the year", (September 21, 2005).^[2]

1.1 SystemVerilog Constructs Supporting Verification

A summary of the SystemVerilog constructs supporting verification is shown in Table 1.1.

Table 1.1 SystemVerilog Constructs for Verification

SystemVerilog Construct	Verification Application
Interface and virtual interface	Provides grouping of signals needed to be driven and viewed by the verification model.
Class and virtual class	Builds reusable extendable classes for the definition of constrained-random variables and the collection of supporting tasks related to common objectives.
Mailbox / Queue	Provides channeling and synchronization of transactions and data. Is also used by scoreboard for verification
Clocking block	Identifies clock signals, and captures the timing and synchronization requirements of the blocks being modeled.
Program block	Provides an entry point to the execution of testbenches. Creates a scope that encapsulates program-wide data. Provides a syntactic context that specifies scheduling in the <i>Reactive</i> region. Creates a clear separation of testbench and design thereby eliminating race conditions in older Verilog.
covergroup	Provides coverage of variables and expressions, as well as cross coverage between them.
Assertions, cover (SystemVerilog Assertions)	Captures temporal behavior of the design as assumptions, checks those behaviors, provides functional coverage and the reporting of information upon error. Assertions can interact with the testbench. ^[3]
API	Supports Application Programming Interface (API) for assertions and coverage.

The SystemVerilog *class* construct deserves some explanation because classes are core to the VMM methodology. A class is a collection of data (class properties) and a set of subroutines (methods) that operate on that data.

- Classes can be inherited to extend functionality.
- Classes can be virtual (requiring a subclass or derived class)
- Classes can be used to build libraries for common functions, e.g., VMM.
- Classes must be instantiated **and constructed to be used**.
- The *randomize* function can be used to randomize class variables (that are qualified via an attribute, *rand*).
- Classes can be typed, parameterized.
- Classes can be passed as objects to methods in other classes and to mailboxes and queues.
- Classes can use virtual interfaces, with the actual interface passed to the constructors. This allows the reuse of classes with multiple interfaces.

Randomization is very key to CRT for the creation of tests targeted toward a coverage-driven verification methodology where the testplan is more focused on the coverage rather than directed tests. SystemVerilog supports the generation of constrained-random values with the use of the *randomize* function, the *rand* and *randc* type-modifier, *randcase* and *randsequence* statements, and the rich sets of constraints with the *constraint* construct.

Coverage is a very important ingredient in the verification process because it provides feedback as to the progress of the verification effort. SystemVerilog offers two types of coverage: temporal coverage with SVA's *cover*, and data coverage with *covergroup*. It also allows them to be used together - for instance a PCI abort condition can be detected via a SVA property and the slaves being addressed during such abort can be monitored (and the address space can be effectively binned/grouped) using *covergroup*. The results of the coverage information can be used to create a reactive testbench based on the coverage information extracted dynamically during simulation.

Assertions play a key role in the verification process as they provide a concise way to capture design behavior spread across multiple and possibly varying number of clock cycles. In addition, assertions can be tightly coupled to the verification environment through the action blocks or calls to tasks from within an assertion thread. They also can be used as SystemVerilog *events*. This interaction capability with the testbench can provide the following:

- a. Write to a variable, thus having the capacity to modify the flow of the testbench.
- b. Update user's implementation of coverage. For example, bits of an initialized static vector can be modified when an assertion (i.e., *assert* or *cover*) reaches a certain state (e.g., passes or is covered). When that vector is all ONEs, then the desired coverage is reached. In addition, SystemVerilog API can also extract coverage info.
- c. Upon a failure, one could write to a file information about the failure, along with a text message. That can include all the relevant variables of the design, the local variables of the assertion thread, simulation time, severity level, etc.

d. SystemVerilog *sequence* can create an *event* when the sequence is finished, and that is very useful to synchronize various testbench elements.

2.0 Why VMM?

SystemVerilog is a vast language with 550+ pages LRM (on top of IEEE Std 1364-2001 Verilog HDL). One is very likely to get trapped into its landscape and thereby using it in a sub-optimal way to achieve the end goal - i.e., finding all bugs. A good methodology is the best way to use the language to its optimum. Figure 2.0a shows the impact of such a methodology in capturing the power of SystemVerilog. VMM represents a methodology supported by a standard library that consists of a set of base and utility classes to implement a VMM-compliant verification environment and verification components. VMM provides several benefits in the construction of testbenches. These include unification in the style and construction of the testbench and in the reporting of information; the quick build of a layered and reusable testbench; and the access to high-level tests using constrained random stimulus and functional coverage to indicate which areas of the design have been checked.

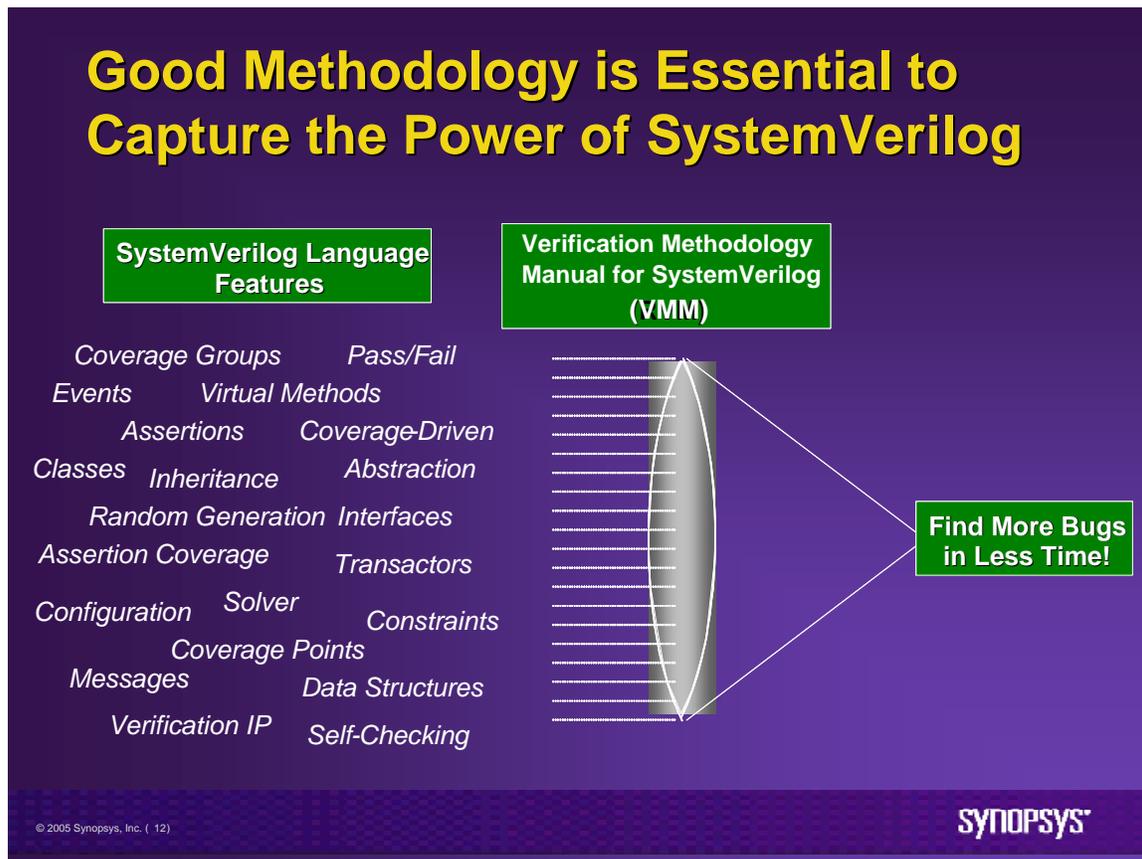


Figure 2.0a Impact of VMM Methodology in Capturing the Power of SystemVerilog

The VMM consists of several base classes as shown in Figure 2.0b, and described in the *VMM for SystemVerilog* book. This paper will demonstrate via an example the application of some of these services. However, the modeling used for this paper did not use all of the features of

VMM, and we are restricting our discussions to the VMM features that we felt were most useful for this simple example.

The major differences between a VMM compliant testbench and a conventional transaction-based testbench include the following aspects:

1. The formalization of the sequencing of steps taken during the verification cycle (See VMM appendix A, *vmm_env*).
2. The methodology used to generate and consume transactions, including the automation with the use of VMM macros.
3. The methodology and support used to adapt transactions to modifications through callbacks.
4. The level of support using the various base-class methods.
5. The methodology used to report logging and status information.

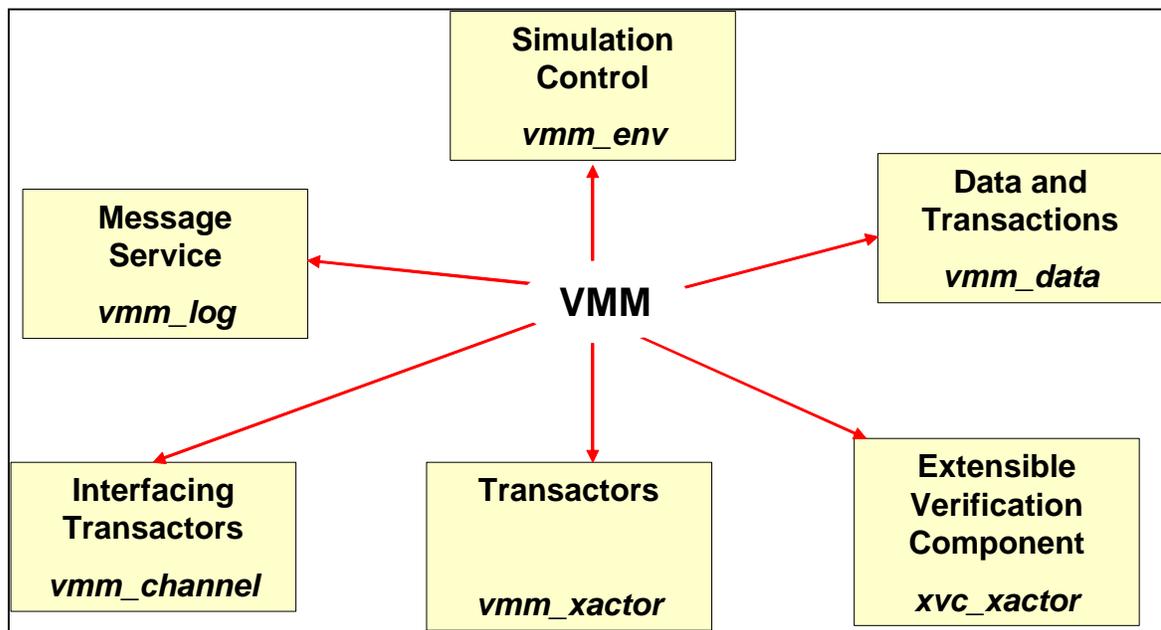


Figure 2.0b VMM Basic Base Classes

3.0 Transaction-Based Verification of a FIFO

This section presents a testbench of a FIFO using the VMM methodology. Fundamentally, VMM recommends a layered approach to building verification environments. While layered testbench concepts have been around for several years now, there has not been any common definition. The different interpretations of layered testbench caused the design of different verification environments even within the same organization. Experience has shown that such heterogeneous verification environments lead to too much redundancy. For example, a verification IP developed by one group doesn't fit easily well into another slightly different project/environment. A significant amount of effort can be easily saved when various teams follow a unified methodology in the architecture of testbenches. For this to become reality, a

reference verification architecture that is flexible to cater various domains needs to be developed. VMM is the industry's first non-proprietary, open, standard language-based verification methodology.

3.1 The DUT

The design under test (DUT) is a synchronous first-in first-out (FIFO) model with the following port connections:

```
module fifo
    (input clk, input reset_n, fifo_if.fslave_if_mp f_if);
```

Those port connections include the clock, the reset, and the FIFO interface using with the slave *modport*. Figure 3.1a. demonstrates the FIFO interface with the *modports* used throughout the design and the testbench.

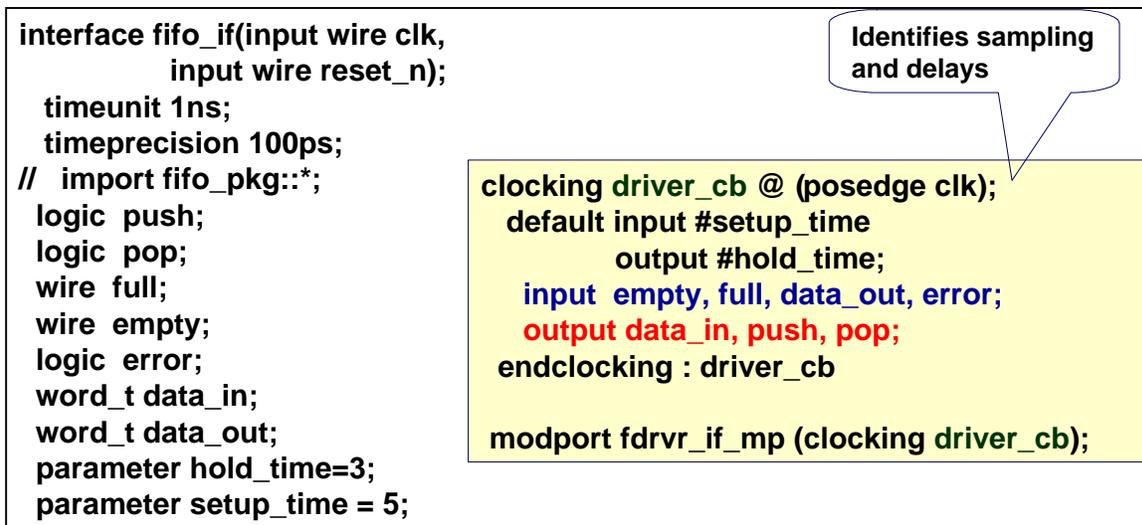


Figure 3.1a FIFO Interface (file *fifo_if.sv*)

The paper addresses the PUSH and POP interface with one transactor (file *fifo_cmd_xactor.sv*) to drive all the signals, and a monitor transactor (file *fifo_mon_xactor.sv*) to monitor the signals. The complete model is shown in the appendix and is available for download.^[4]

A DUT typically carries with it a set of requirement documents and a set of interfaces. SystemVerilog provides a useful construct, the *interface* to abstract the communication across several modules. Some designers use the SystemVerilog *interface* definition in the RTL design. Others restrict the design to the Verilog style with individual port signals, instead of grouping the signals with SystemVerilog interfaces. If an *interface* is not defined, it is necessary for the verification engineer to define such an interface model as this facilitates the connections to the verification environment defined in classes through the use of virtual interfaces.

An *interface* typically has tasks and assertions associated with the operation of the signals of the interface. Examples of tasks in a FIFO interface include a *push_task*, a *pop_task*. VMM (rule 4-9) recommends the definition of those tasks associated with the *interface* in *classes* and *subclasses* (a.k.a. derived classes), separate from those defined in the interface. Interface

assertions relate to the properties or timing relationships of those signals. Assertions can be defined in the interfaces or in a property module bound to the DUT instance. The specifications are used to build the *interfaces* and property module, as shown in Figure 3.1b.

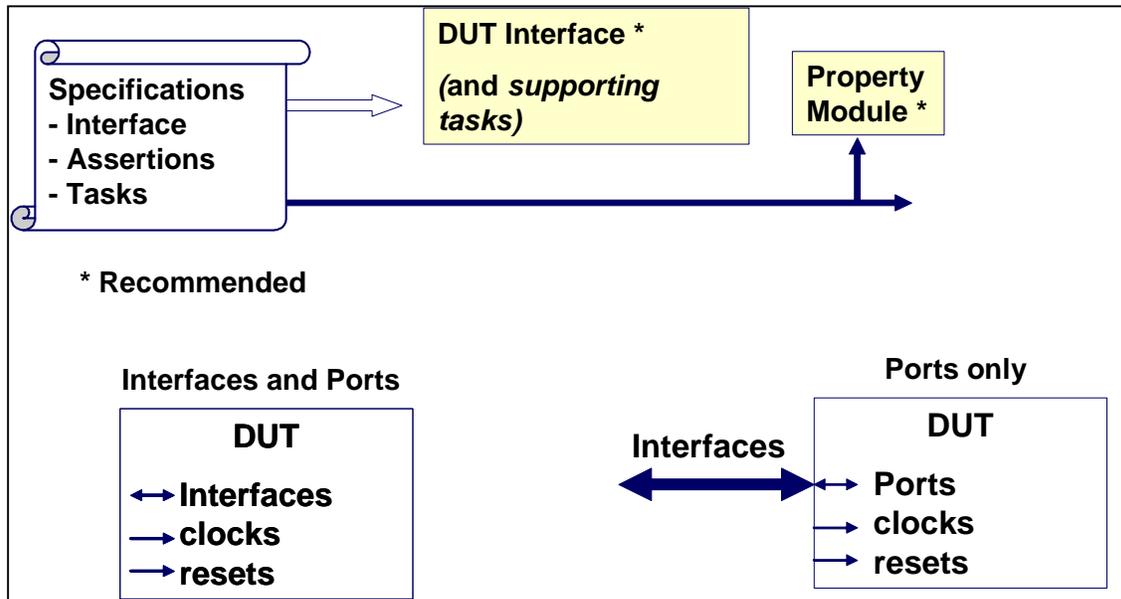


Figure 3.1b. Specifications Used for Definition of Interfaces and Verification Modules

3.2 The Testbench

3.2.1 Conceptual View

The basic idea of a transaction-based verification methodology, such as VMM, is to separate the *transaction* from the *transactor*. While there are few varying definitions of these terms, here is a simple definition we follow in this paper. A *transaction* is basically “what needs to be tested” and a *transactor* models “how to test”. Examples of a *transaction* are:

1. **Instruction.** This represents the high-level tasks to be executed, such as a READ, WRITE, NO-OP, LOAD, etc.
2. **Data.** This represents information such as address, data, number of cycles, etc.
3. **Parameters.** This can represent a mode, a size, etc.

In VMM, a *transactor* is a generic name, and there are several kinds of transactors such as generator, driver, monitor, scoreboard etc. A direct equivalent of a typical VMM *transactor* is what’s conventionally known as BFM (Bus Functional Model) at the lower level. On the driver side, a BFM takes a *transaction* as input and sends it to the DUT according to the underlying protocol.

It is best to use SystemVerilog *classes* to declare *transactions* and *transactors*. The rationale behind this guideline is as follows: by definition a *transaction* has a limited lifetime - from the time it gets generated to the time it is consumed by the DUT, checked for correctness etc. The number of such *transactions* in a system is variable - this would logically mean that a dynamic memory allocation of such *transactions* is a must have to make optimal use of simulation. By

definition, constructs such as plain Verilog *modules* and SystemVerilog *interfaces* etc. are “static” in nature - they exist throughout the simulation and hence are not suited for modeling dynamic *transactions*. Another key reason to consider using SystemVerilog *classes* to model *transactions* is the ability to easily derive and extend them to create variations of transactions and to mimic real-life data streams such as Ethernet Packets. An Ethernet packet, as defined by the standard, has several layers (such as L2, L3, L4 etc.), and each layer encapsulates another one. With a conventional “Hardware Design Language” one is limited to use only a simple modeling style that does not lend to good, maintainable and reusable code. The Software domain has been handling such complexity in the past with great success with Object-Oriented (OO) programming style. SystemVerilog brings in that OO style to Hardware Verification via the *class* data type.

VMM defines a base class named *vmm_data* to model *transactions*. It is used as the basis for all transaction descriptors and data models. A simple example of a FIFO *transaction* modeled using *vmm_data* is shown in Figure 3.2.1a.

```

class fifo_xactn extends vmm_data;
  rand fifo_scen_t          kind;    // see package for type
  rand logic [BIT_DEPTH-1:0] data;
  rand int                 idle_cycles;
      time                 xactn_time;
endclass : fifo_xactn

```

Figure 3.2.1a. Transaction Class Example (file *fifo_xactn.sv*)

The *vmm_data* base class defines several virtual functions and tasks. A complete list is beyond the scope of this paper. One such function is the *vmm_data::copy()* that should (VMM Rule 4-76) be extended to add relevant transaction fields. In our FIFO example, this function is shown in Figure 3.2.1b.

```

function fifo_xactn fifo_xactn::copy(vmm_data cpy);
  fifo_xactn local_xactn;
  if (cpy == null) begin
    local_xactn = new;
  end
  else if (!$cast(local_xactn, cpy)) begin
    `vmm_fatal(log, "Attempting to copy a non fifo_xactn instance");
    copy = null;
    return;
  end

  local_xactn.kind = this.kind;
  local_xactn.data = this.data;
  local_xactn.idle_cycles = this.idle_cycles;
  copy = local_xactn;
endfunction : copy

```

Figure 3.2.1b. Sample User-defined Copy Method (file *fifo_xactn.sv*)

Other useful *vmm_data* functions include the *compare()* for comparison of transactions useful with scoreboards to check for data integrity, and *psdisplay()* for the return of an image of the transaction. The macro ``vmm_channel` (discussed later on) defines a channel class derived from the *vmm_data* to transport instances of a specified class (e.g., a transaction).

Transactors are the workhorses of a transaction-based verification (TBV) environment; they perform the actual job of transferring the data (*transaction*) to other units to perform a task, such as driving the DUT pins or driving the verification scoreboard.

This concept is represented in Figure 3.2.1c where in constrained-random testing, the transactions defined in a *transaction* class are randomized with a generator and sent to a *transactor* via a *channel* for the execution of those transactions. For example, a transaction such as a PUSH / IDLE / POP is randomized with constraints, and then sent to a *channel* (constructed with a queue and characterized in the diagrams as a mailbox) via the *put* method. The *put* method blocks if there is no room in the *channel* to insert another transaction. When the *transactor* is ready to process another transaction, it extracts from the *channel* the next transaction via the *get* method. The *transactor* then proceeds on executing the begotten transaction.

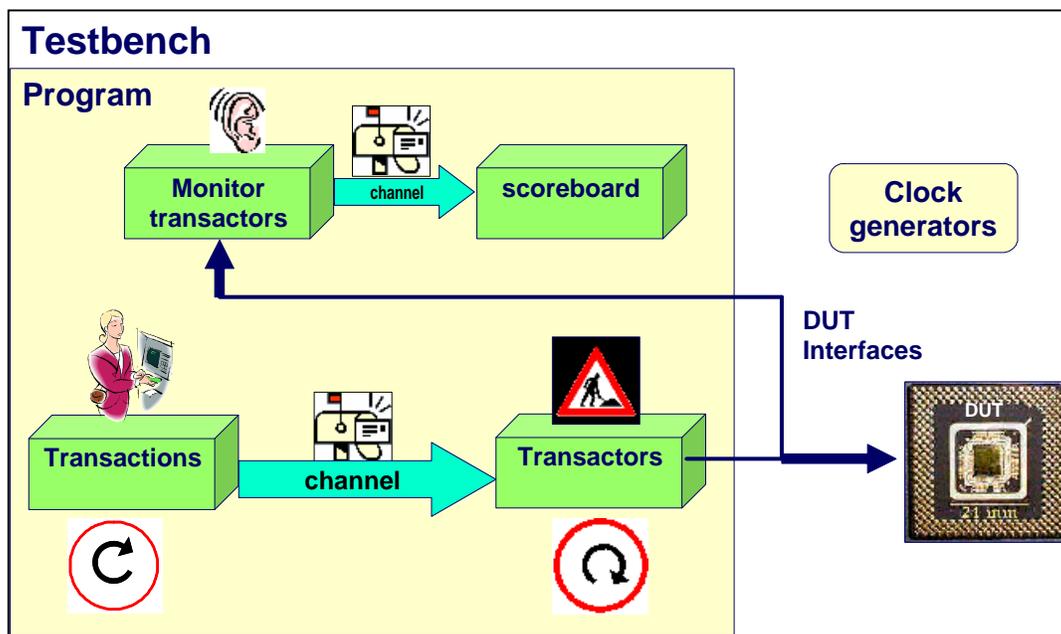


Figure 3.2.1c High-level View of the Testbench

Note that the use of a *channel* provides several advantages, including the buffering and separation between the generation and the consumption of the transactions. A second advantage is the simplicity in clock synchronization between the generation and consumption side of the transactions. Specifically, they do not need to be synchronous to a common clock because the insertion and extraction of transactions is separate. A third advantage is the capability to easily modify the transactions through *callbacks* to provide changes such as error injection. A fourth advantage is capability to have generation (and even consumption) of the transactions be performed by different agents/transactors.

3.2.2 Testbench Outline

Figure 3.2.2 represents a structural view of the testbench. The testbench includes the following objects:

1. **Variables declarations:** These are variables local to the testbench
2. **Interface instantiations:** These are the DUT interfaces to provide the connection between the stimulus drivers/monitors and the DUT.
3. **Program instantiations:** The *program* provides the control for testing the DUT. A testbench may contain more than one *program*.
4. **DUT instantiations:** These are the devices under test.
5. **Binding of property modules to DUT instances:** Property module typically includes assertions and coverage requirements.
6. **Clock generators:** These generators emulate the clocks in the system.

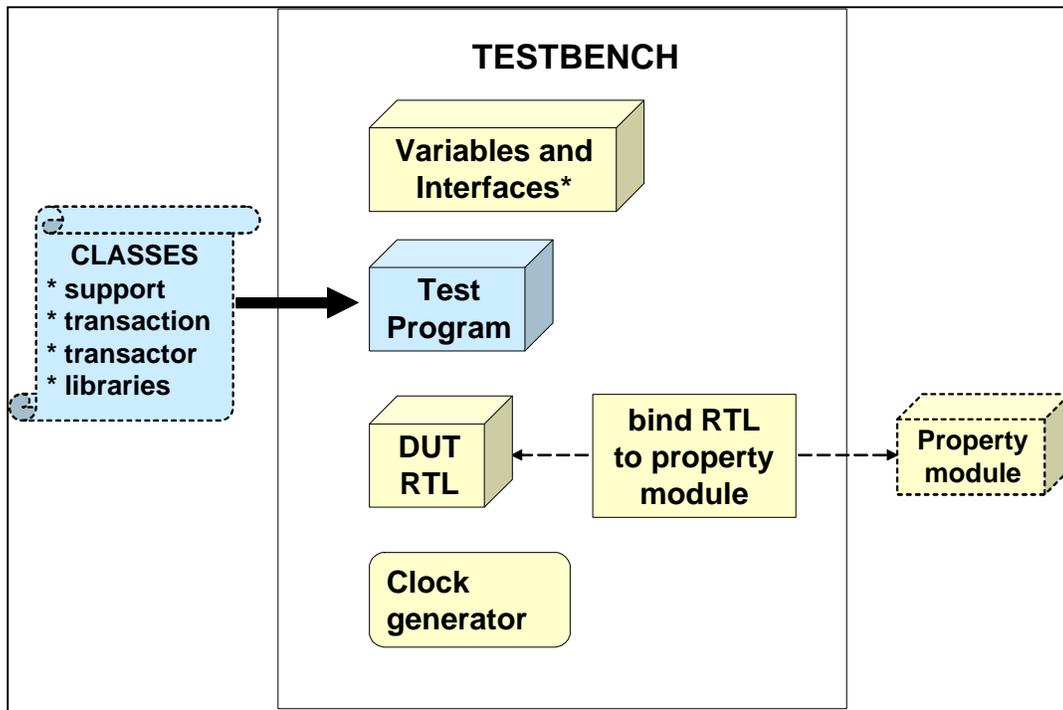


Figure 3.2.2 Testbench Structure

3.2.3 The program

SystemVerilog *program* provides an ideal encapsulation for all testbench related items. It acts as entry and exit points for the simulation. SystemVerilog LRM has well defined semantics for *program* that requires it to execute under reactive time-step, thereby eliminating any design-to-testbench race conditions (one of the most recurring problems with plain Verilog testbenches and many teams have spent unproductive debug cycles in detecting and resolving them in the past). The *program* is responsible for the generation and verification of the test vectors. Figure 3.2.3a represents the *program* for the verification of the FIFO model. The *program* makes use of the VMM library, a user-defined set of constants and type definitions, a *transaction* class, and an *environment* class. A *program* can have the following constructs: *initial*, continuous assignment, *final* construct, module or generate item declaration, concurrent assertion item,

timeunits declaration. It cannot contain *always* blocks, UDPs, modules, interfaces, or other programs. VMM Rule 4-7 is a worthy rule related to the *program* block that states: “synchronous interface signals shall be sampled and driven using a *clocking* block. This approach will avoid race conditions between the design and the verification environment, and it will allow the verification environment to work with RTL and gate-level models of the DUT without any modifications or timing violations.” See files *fifo_if.sv*, *fifo_cmd_xactor.sv* for an application of the *clocking* blocks, and section 3.3 for the resulting display of hold times specified in the *clocking* blocks.

Per VMM guidelines, the *initial* block constructs an *environment* object and starts the *run* method from the *environment* class. The *vmm_env::run()* execution sequence is described in the VMM book and is shown in Figure 3.2.3b. That *run* method essentially builds and starts the verification environment, including the *fifo_xactn*, the generator of transactions into the transaction channel with the *fifo_xactn_push_atomic_gen*, the transactor to drive the FIFO interface, and the monitor to extract the observed transactions on the FIFO interface.

```

program fifo_test_pgm (fifo_if fifo_if_0
    );
    timeunit 1ns; timeprecision 100ps;
`include "vmm.sv"
`include "fifo_pkg.sv"
`include "fifo_xactn.sv"
`include "fifo_env.sv"
    vmm_log log;
    fifo_env fifo_env_0;
    fifo_xactn my_push_xaction;

initial
    begin
        // Build all components of an environment - testbench
        log = new("Pgm_Logger",0);
        fifo_env_0 = new(fifo_if_0);
        `vmm_note(log,"Started");
        fork : f1
            fifo_env_0.run();
        join_none
        #100000;
    end
endprogram : fifo_test_pgm

```

Figure 3.2.3a. High-Level Structural View of the Program (file *fifo_pgm.sv*)

The formal arguments of the *program* include the *fifo* interface (and may include other interfaces and signals).

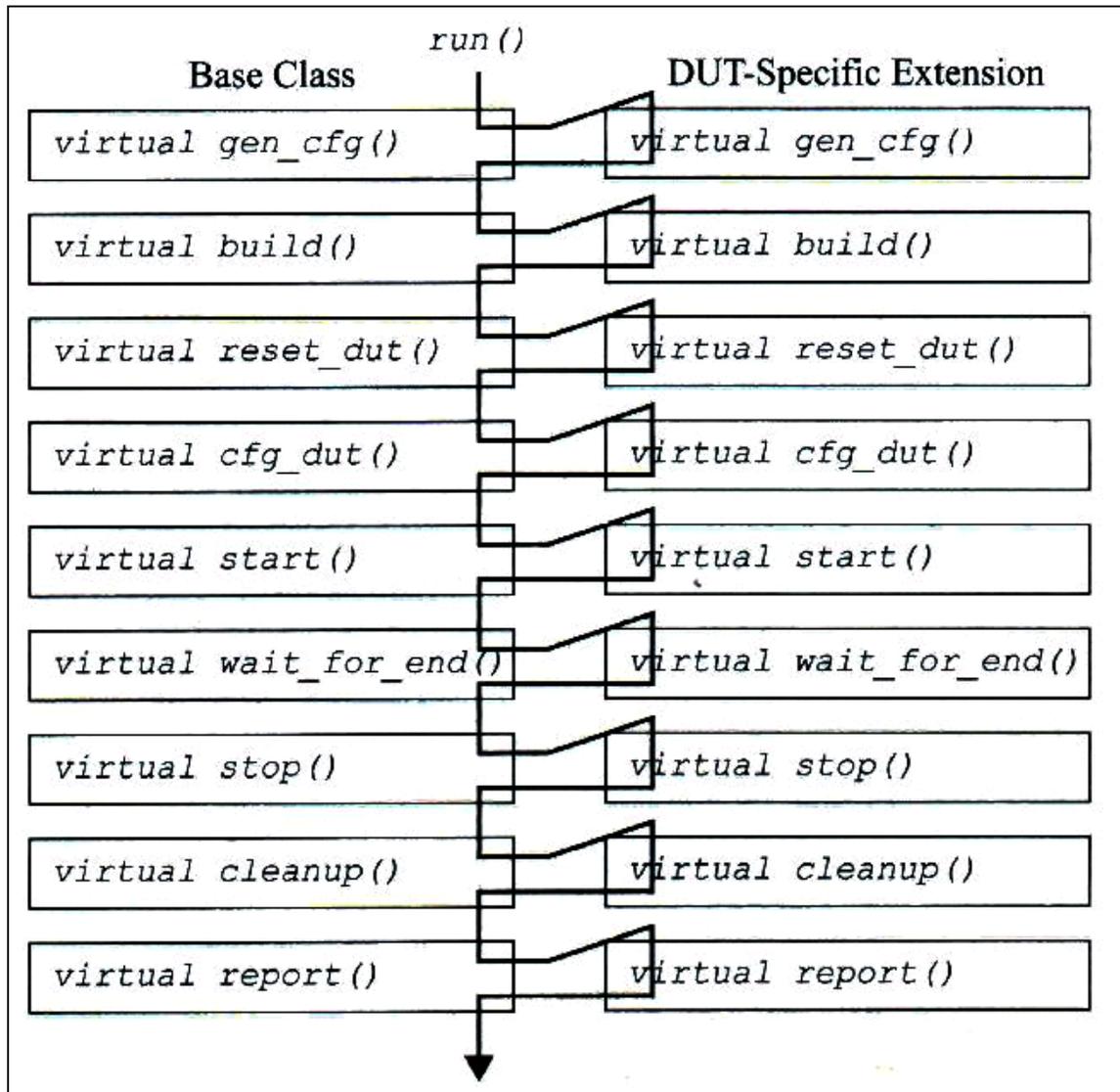


Figure 3.2.3b vmm_env::run() Execution Sequence
 (from Verification Manual Methodology Manual for SystemVerilog)

3.2.4 A generalized test flow mechanism

In general, every test that is being run in a simulation follows a test-flow such as *initialize – start – wait_for_end – finish*. In many environments this is not defined a priori leading to various difficulties such as ease of integration, wasted debug cycles (e.g., after hours of debug, problem might be root caused to premature start of packet transmission - before configuring the DUT registers). VMM defines a well thought out flow to avoid such problems, in addition to being very flexible to suit different environments. It also has built-in checks to make sure the steps are not by-passed accidentally.

The *run()* method consists of the calls to other methods, which are summarized below. Many of the specific extensions for these methods are described in this paper.

gen_cfg()

This method creates a random configuration of the test environment and DUT. It may choose the number of input and output ports in the design and their speed, or the number of drivers on a bus and their type (master or slave). One can also randomly select the number of transactions, percent errors, and other parameters. The goal is that over many random runs, one will test every possible configuration, instead of the limited number chosen by directed test writers.

build()

This method builds the testbench configuration generated in the previous method. This includes generators and checkers, drivers and monitors, and anything else not in the DUT. An example of the user-defined *build* method is shown in section 3.2.6.

cfg_dut ()

In this method one downloads the configuration information into the DUT. This might be done by loading registers using bus transactions, or backdoor loading them using *\$readmemh/b* or a hierarchical reference to configuration registers (e.g., *top.chip.pci_blk.cfg_0 = 10*), or C code.

start()

This method starts the test components. This is usually done by starting the transactor objects. For example, the *start()* task in class *fifo_env* (which extends *vmm_env*) call the *start_xactor()*, which in turn call the *main()* tasks in the transactors. This is a key step in the whole flow - this is where all components of the testbench are starting their intended operations. For all *transactors* that are built in the environment, their individual *start_xactor()* task should be called in this step.

```
task start(); // in fifo_env.sv file
    super.start();
    this.push_gen_0.start_xactor();
    this.fifo_cmd_xactor_0.start_xactor();
    this.mon_0.start_xactor();
endtask : start
```

wait_for_end()

This method waits for the end of the test, usually done by waiting for a certain number of transactions or a maximum time limit. Depending on the design under test and the nature of test, this might become complicated - for instance, one may need to monitor internal state machines to see when they return to their idle states.

stop()

This method stops the data generators and waits for the transactions in the DUT to drain out.

cleanup()

This method checks recorded statistics and sweeps for lost data.

report()

This method prints the final report. Note that *vmm_log* will automatically print its report at the end of simulation.

The following sections describe the *transaction* classes, the creation of *channels*, the generation of transactions into the channels, the consumption of transactions from the channels, the *build* of the environment, and the *start* of execution. Simulation results are then presented.

3.2.5 Transaction class

In a TBV methodology, a *transaction* defines the basic data model of the system, establishes a common currency for the system. The individual properties/members of a *transaction* may need to be randomized to support a CRT on top of TBV. The *transaction* class is defined as a derived class of *vmm_data* to take advantage of methods available from this base class. Also consistent use of *vmm_data* to derive transactions will ensure same look and feel and shall help in maintenance of the code over a period of time. Figure 3.2.4 represents the transaction class for the FIFO.

```
class fifo_xactn extends vmm_data;

    rand fifo_scen_t          kind;
    rand logic [BIT_DEPTH-1:0] data;
    rand int                  idle_cycles;
    time                      xactn_time;

    constraint cst_data {
        data < 1024;
    }
    constraint cst_idle {
        idle_cycles inside {[1:3]};
    }

    constraint cst_xact_kind {
        kind dist {
            PUSH := 400,
            POP := 300,
            PUSH_POP := 200,
            IDLE := 300,
            RESET := 1
        };
    }

    static vmm_log log = new("fifo_push", "");
    function new();
        super.new(this.log);
```



```

`vmm_note(this.log, "Message from constructor");
endfunction : new

virtual function string psdisplay(string prefix = "");
    $sformat(psdisplay,
            "%s Fifo push Xaction %s \n",
            prefix, this.kind.name());
endfunction : psdisplay
extern virtual function  fifo_xactn_push
                        copy(vmm_data cpy = null);

endclass:fifo_xactn_push

```

Figure 3.2.4. Transaction Base Class (file *fifo_xactn.sv*)

Some of the important methods inside *vmm_data* are *copy()*, *psdisplay()*, etc. Every *vmm_data* derivative should implement the actual definitions for these functions. Function *psdisplay()* defines a consistent way to display every transaction object in the system. It returns a string, and hence can be easily used in any *\$display* call etc. The *copy()* method provides a very important functionality of implementing what constitutes a true copy of the *transaction*.

3.2.6 Transactor class

Transactors represent the workhorses of the system. All the BFM, generators, scoreboards, monitors etc. are built as transactors. VMM defines a base class named *vmm_xactor* for a generic transactor. All transactors in a system should be derived from this *vmm_xactor*. A *vmm_xactor* has several hooks for allowing basic functionalities as well as advanced features such as flow control. A detailed look into how a transactor operates is provided later in the paper.

3.2.7 Creation of Channels

A *channel* provides the structures (e.g., queues) to store the transactions, and provides the support to process those transactions. One side of the *channel* is the *generator* putting *transactions* into the *channel*. The functional *transactor* (e.g., the BFM) gets the *transactions* out of the *channel*, and executes them. The VMM *channel* is constructed with a queue that has both high-water and low-water marks to fine-tune the interactions between the producer and consumer. Channels allow flow control, so the *put()* method will block if the channel is full. The *get()* method removes the transaction from the end of the *channel*, while *peek()* provides a handle to it without removal. Both the *get()* and the *peek()* block if the *channel* is empty. Note that a *channel* acts like a mailbox, and is symbolically represented as a mailbox in the diagrams.

To facilitate the implementation of *channels* VMM automatically creates a derived class from the user-defined transaction class using the ``vmm_channel` macro as shown in the example below (see file *fifo_xactn.sv*):

```

`vmm_channel (fifo_xactn_push)

```

That macro creates the class *fifo_xactn_push_channel* that produces a strongly typed queue to help prevent coding errors. Figure 3.2.5 represents a graphical view for the creation of the *channel class* and the creation of the *transaction generator* (discussed in the next subsection).

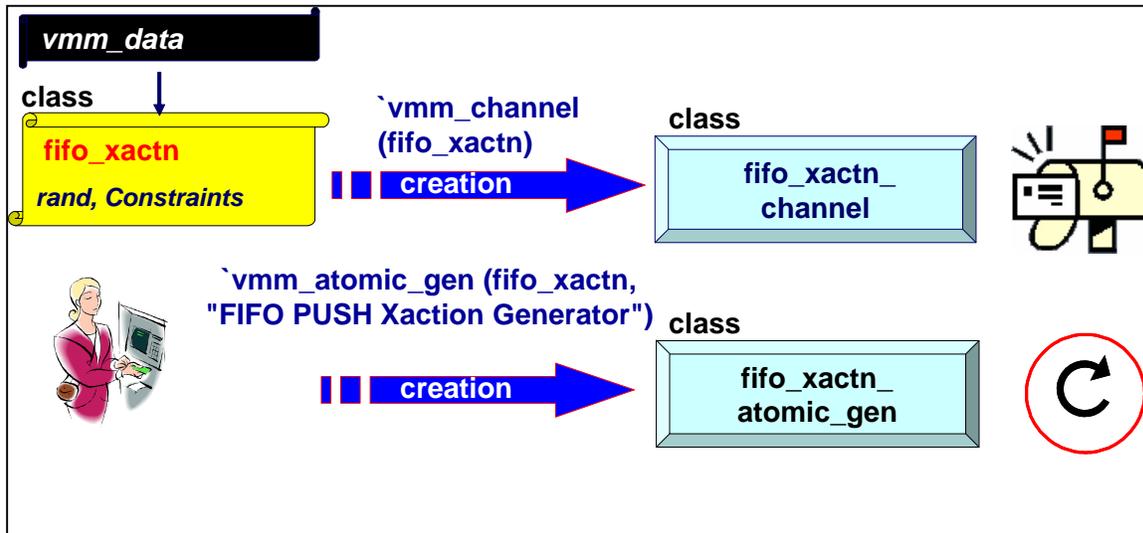


Figure 3.2.5 Creation of Channels and Generators from Transactions
(See files *fifo_xactn.sv* , *fifo_env.sv*)

3.2.8 Generation of Transactions

In a CRT methodology, the default transactions are randomly generated. This generation can be accomplished as follows:

- Instantiate a “blueprint” of the object to be generated (e.g., a *new* transaction).
- Construct it, randomize it.
- Push it to the output *channel* so that they can be extracted by the down-stream *transactors*. Note that the pushing can be blocked when the channel is to capacity.
- Loop process.

To support this feature in an automatic manner, VMM provides a macro `\vmm_atomic_gen` and `\vmm_scenario_gen` for the creation of generator classes for atomic (purely random with no sequence) and sequence generation of transactions. In this model we used a simple automatic generator with the pre-defined macro for simplicity:

```
\vmm_atomic_gen (fifo_xactn, "FIFO PUSH Xaction Generator")
```

This macro creates the class *fifo_xactn_atomic_gen*. When this class is instantiated, connected to the *channel*, and started, then the *transactions* are automatically generated, randomized and put into the *channel* for extraction by the consumer. The user does not have to create or call a method to explicitly do this randomization / generation function.

The application of this generated class is demonstrated in the *build* method of *fifo_env*, as shown in Figure 3.2.6a.

```

// In file fifo_env.sv
// Channel instance
fifo_xactn_channel      fifo_push_channel;
// Generator instance
fifo_xactn_atomic_gen  push_gen_0;

function void build();
    this.fifo_push_channel = // instantiate channel
                           new("push_chan", "0");
    this.push_gen_0 =       // instantiate generator
                           new ("Push Xaction generator", 0);

    this.push_gen_0.out_chan = // Connect channel to generator
                              this.fifo_push_channel;
    ...
endfunction : build

```

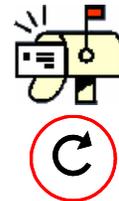


Figure 3.2.6a Generation of Transactions into Channels (file *fifo_env.sv*)

Figure 3.2.6b represents a graphical view of this generation of transactions into channels.

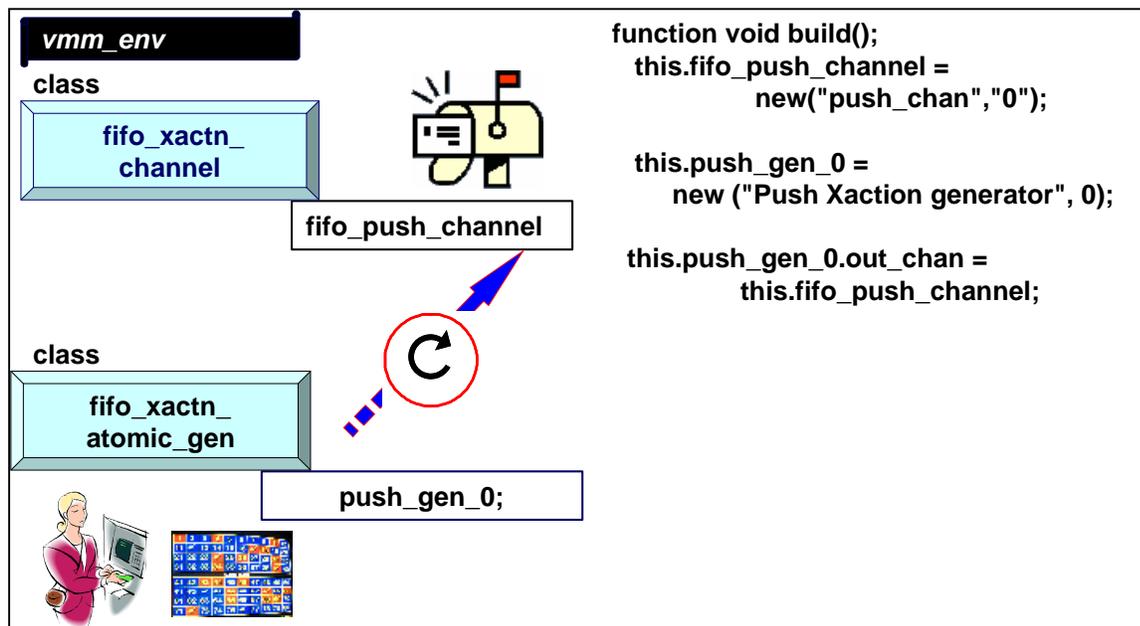


Figure 3.2.6b Generation of Transactions into Channels

3.2.9 Consumption of transactions from the channels

The *transactor* class is responsible for extracting (or getting) the *transaction* from the *channel*, and parsing the *transaction* into the vector sequences used by the DUT. The *transactor* needs to communicate the assertion of vectors onto signals. To facilitate reuse, those signals are defined into *virtual interfaces*. In this example, the FIFO driver interface is of type *fifo_if.fdrvr_if_mp*, Figure 3.2.7a represents the consumption of *transactions* from the *channel*.

The *build()* method in the user-defined environment (file *fifo_env.sv*) creates the simulation environment. Figure 3.2.7b demonstrates the generation and consumption of *transactions* through the use of *channels* for transaction transfers, the use of the *atomic generator* for the production of the *transactions* into the *channels*, and through instantiation of the *transactor* for the consumption of the *transactions*.

The *main* task in the transactor gets the *transaction* from the *channel*, analyzes its contents, and drives signals onto the virtual interface. This is demonstrated in Figure 3.2.7c. The *main* task of the *fifo_xactor* derived class (extended from *vmm_xactor*) is automatically started by the base class *vmm_xactor*.

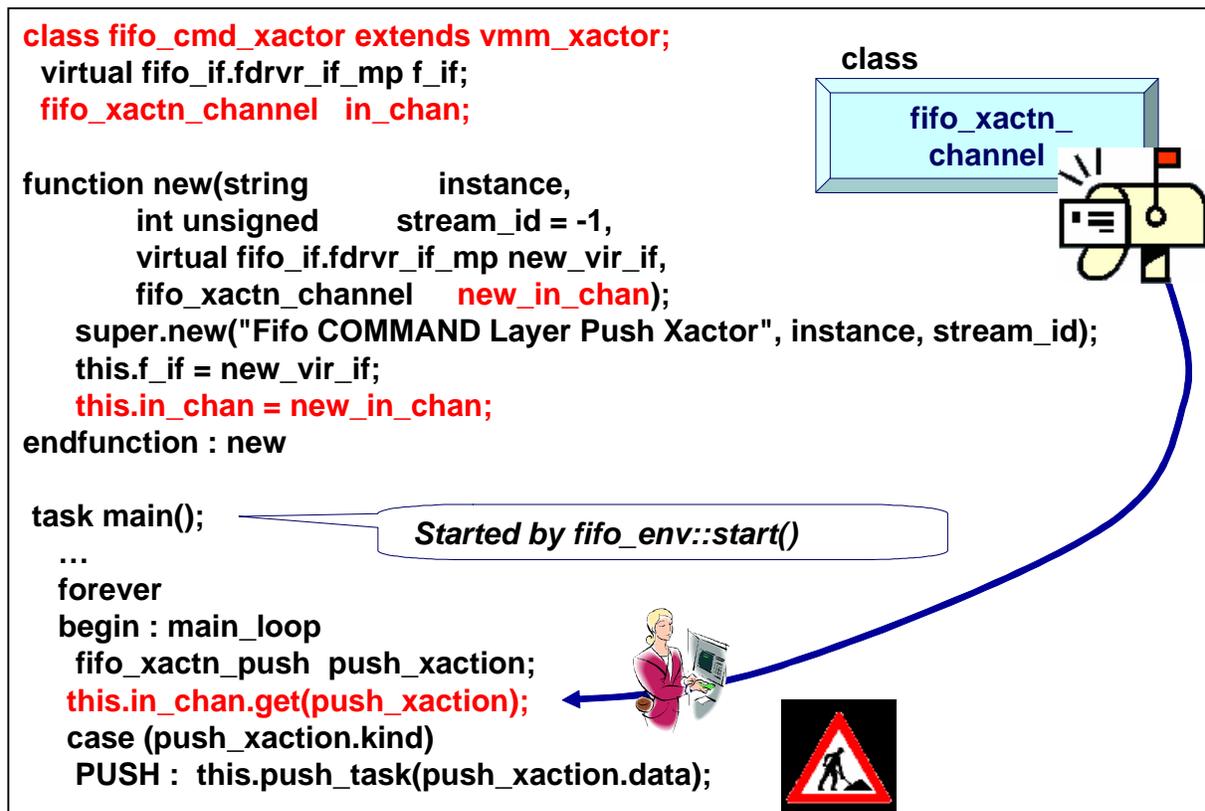


Figure 3.2.7a. Consumption of Transactions from the channel (see file *fifo_cmd_xactor.sv*)

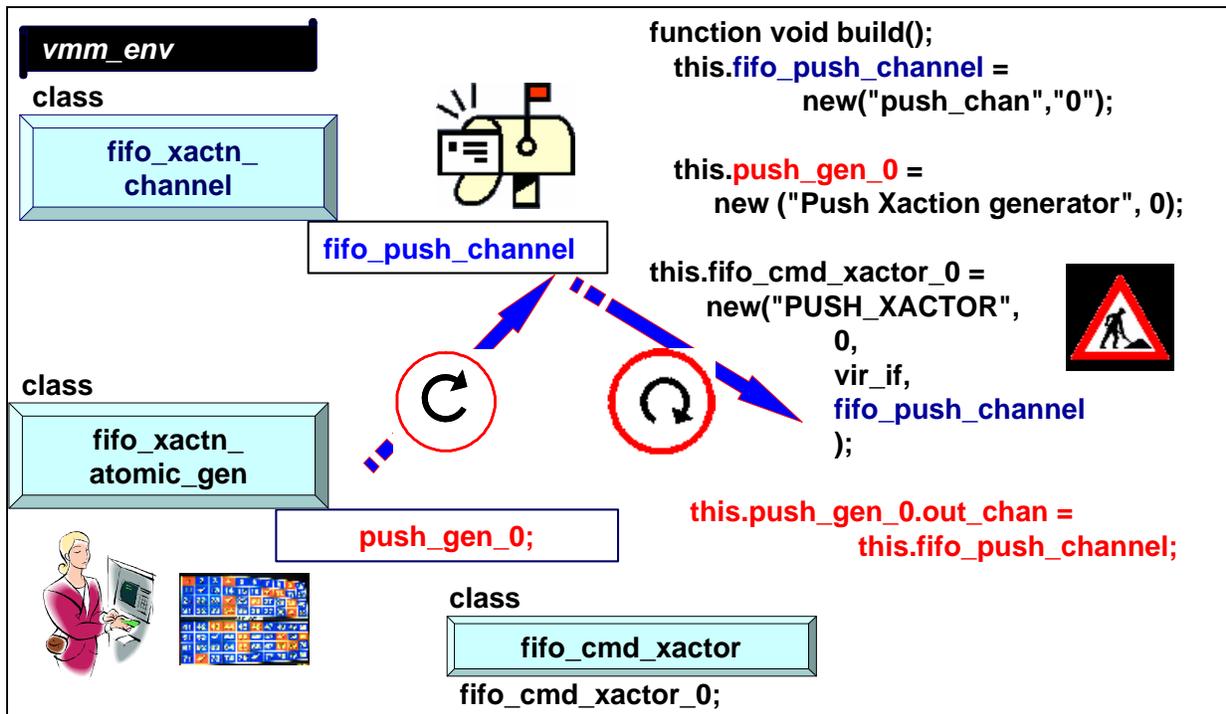


Figure 3.2.7b. Generation and Consumption of Transactions

```

class fifo_cmd_xactor extends vmm_xactor;
. .
task main();
  fork
    super.main();
  join_none
  forever
  begin : main_loop
    fifo_xactn push_xaction;
    this.in_chan.get(push_xaction);
    case (push_xaction.kind)
      PUSH : this.push_task(push_xaction.data);
      POP  : this.pop_task();
      PUSH_POP : this.push_pop_task(push_xaction.data);
      IDLE : this.idle_task(push_xaction.idle_cycles);
      RESET : this.reset_task(5);
    endcase
  end : main_loop
endtask : main

```

VMM rule 4-93 – All threads shall be started in the extension of the *vmm_xactor::main()* task. *main()* is started by *fifo_env::start_xactor()*

Extracting the transaction from the channel

```

task push_task (logic [BIT_DEPTH-1:0] data);
  begin
    $display ("%0t %m Push data %0h ", $time, data);
    f_if.driver_cb.data_in <= data; // using clocking block
    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
..

```

Figure 3.2.7c. Execution of Transaction by Transactor (file *fifo_cmd_xactor.sv*)

3.2.10 Monitoring of Transactions

“A monitor is *passive transactor* that autonomously reports observed data or transactions. It may include a *checker* or equivalent checking functionality for the observed protocol, but not the data or transactions transported by the protocol”. Thus, the *monitor* examines the interface, creates a *transaction* based on what is observed on the interface, and *puts* that observed *transaction* onto a monitor *channel*. That process is demonstrated graphically in Figure 3.2.8a.

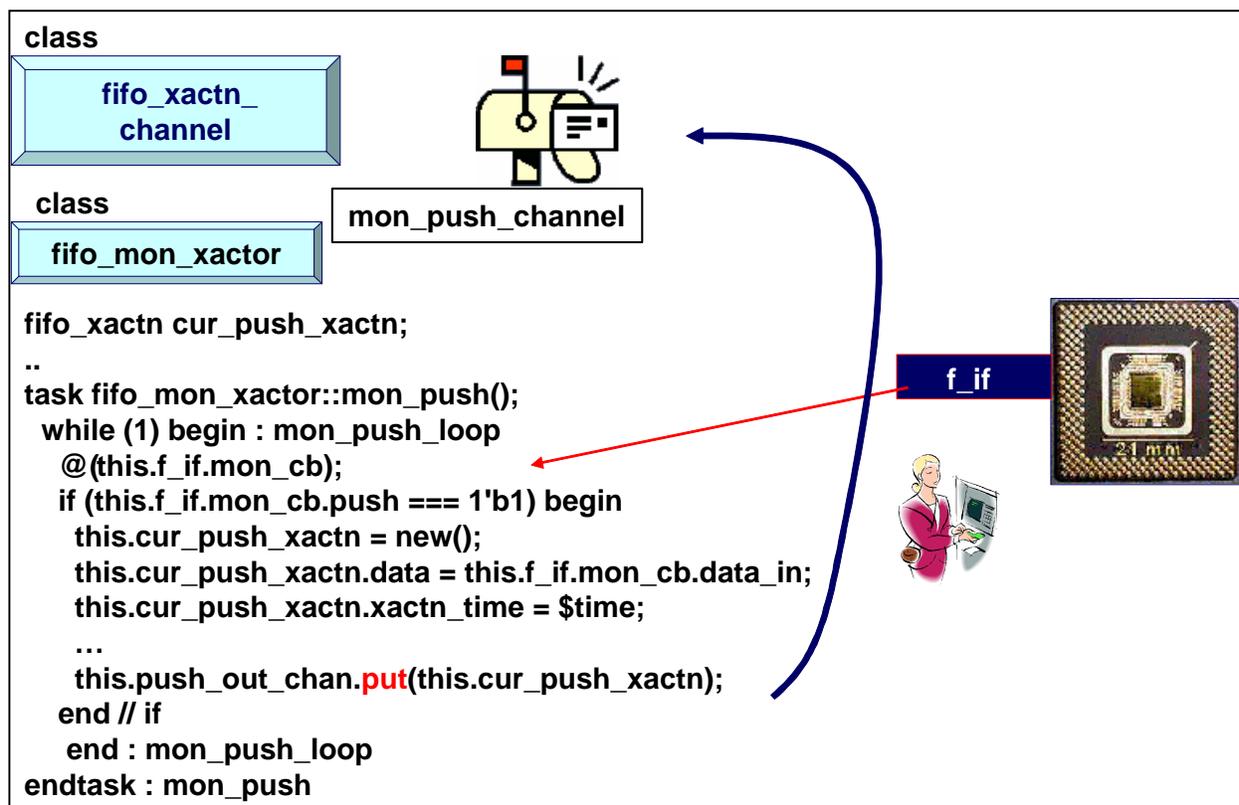


Figure 3.2.8a Putting Transactions into Monitor Channel (file *fifo_mon_xactor.sv*)

3.2.11 Class Relationships in UML

To express the class relationships of the testbench design model, a UML diagram was created using StartUML™ [5]. These are shown in Figure 3.2.11a and Figure 3.2.11b. The diagram demonstrates the class relationships between the classes, and the class objects, operators. Design patterns are addressed in the book *Design Patterns* [6] and is a recommended reading.

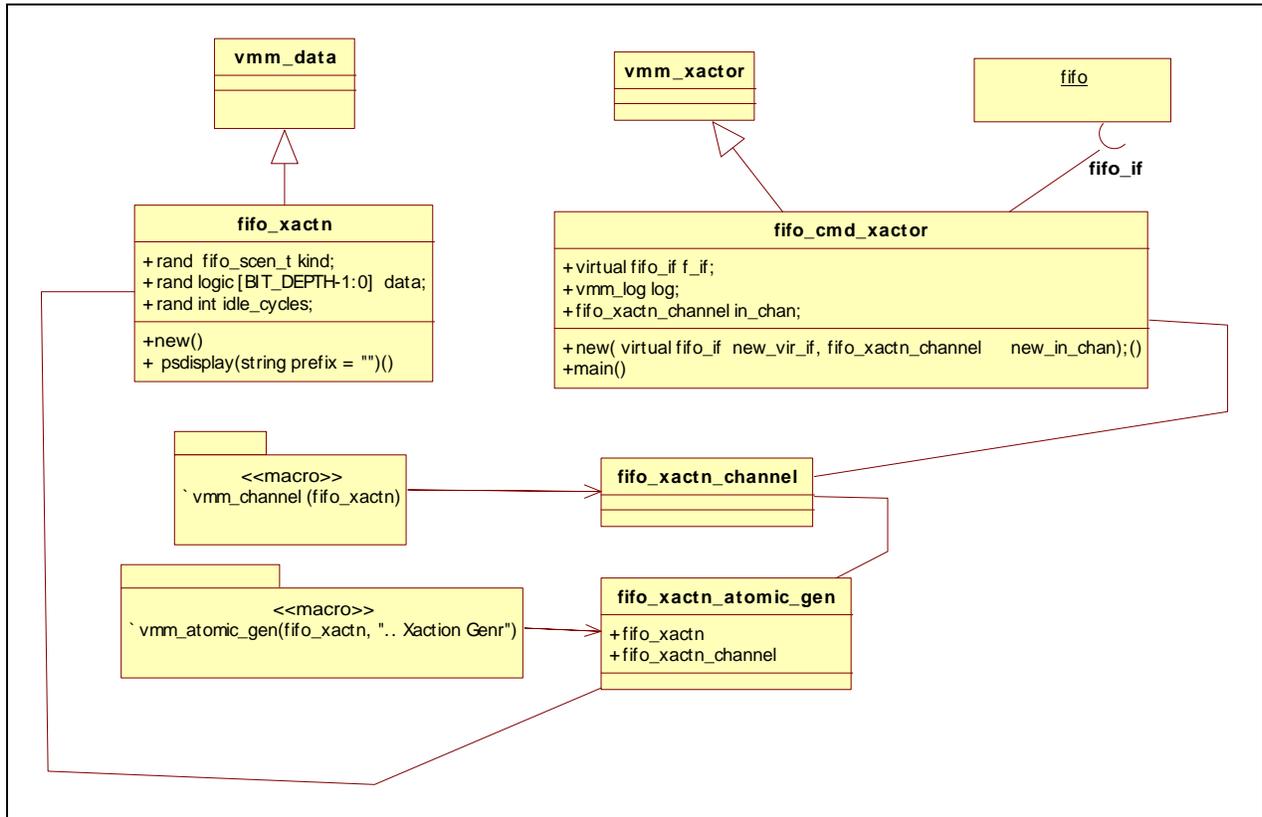


Figure 3.2.11a Class Relationships of the Testbench Design Model

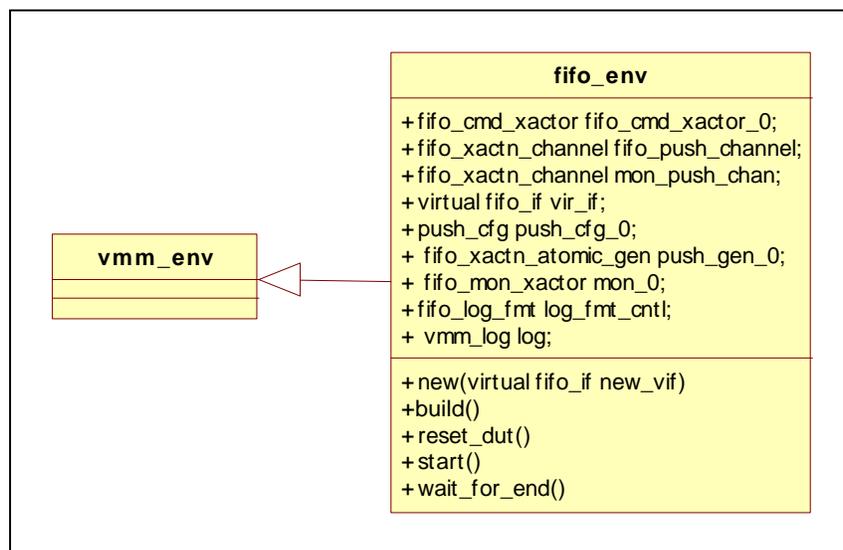


Figure 3.2.11b Class Relationships of the Testbench Design Model

3.2.12 Message Service

The message service uses the *vmm_log* class. This class and its supporting functions and macros help to ensure a consistent look and feel to the messages issued from different sources. This section demonstrates an example in the use of the *vmm_log* class and macros.

The class *fifo_xactn_push* demonstrates the use of *vmm_log* and the definition of the Synopsys VCS function *psdisplay*. Figure 3.2.12a provides a snippet of the code.

```
static vmm_log log = new("fifo_push", "");
function new();
    super.new(this.log);
    `vmm_note(this.log, "Message from constructor");
endfunction : new

virtual function string psdisplay(string prefix = "");
    $sformat(psdisplay, "%s Fifo push Xaction %s \n",
            prefix, this.kind.name());
endfunction : psdisplay
```

Figure 3.2.12a. Application of *vmm_log* (file *fifo_xaction.sv*)

The class *fifo_cmd_xactor* demonstrates the application of *`vmm_note* macro and the *\$psprintf*. Figure 3.2.12b provides a snippet of that code.

```
vmm_log log;
function new(string instance,
            int unsigned stream_id = -1,
            virtual fifo_if.fdrvr_if_mp new_vir_if,
            fifo_xactn_push new_in_chan);
    super.new("Fifo COMMAND Layer Push Xactor",
            instance, stream_id);
    this.f_if = new_vir_if;
    this.in_chan = new_in_chan;
    this.log = new("Fifo COMMAND Layer Xactor", "Logger0");
    `vmm_note(this.log, "Push CMD_Xactor new");
endfunction : new

task main();
    ..
    begin : main_loop
        fifo_xactn push_xaction;
        `vmm_note(this.log,
            "About to Get a new fifo xaction from in_channel ");
        this.in_chan.get(push_xaction);
        `vmm_note(this.log,
            $psprintf("Got a new fifo xaction from in_channel %s ",
                push_xaction.psdisplay()));
    end
endtask
```

Figure 3.2.12b. Application of *`vmm_note* and the *\$psprintf* (file *fifo_cmd_xactor.sv*)

During simulation, Figure 3.2.12c demonstrates a sample text of what was displayed for these functions and macros.

```

0.00 ns Pgm_Logger [Normal:NOTE] | Started
0.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | Push CMD_Xactor new
0.00 ns fifo_push [Normal:NOTE] | Message from constructor
0.00 ns FIFO Env Logger [Normal:NOTE] | Sim shall run for no_of_xactions 238
..
1950.00 ns fifo_push [Normal:NOTE] | Message from constructor

1950.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | About to Get a new fifo
xaction from in_channel

1950.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | Got a new fifo xaction
from in_channel Fifo push Xaction PUSH_POP

1950.00 ns fifo_tb.utest_pgm.\fifo_cmd_xactor::push_pop_task Push data e

```

Figure 3.2.12c Sample Display of Simulation Messages

3.3 Simulation Results

All simulations were performed with Synopsys VCS Version X-2005.SP1. Figure 3.3a shows the assertion failure summary for the property module bound to the DUT. Figure 3.3b shows a Push Error (Push on FULL), while Figure 3.3c demonstrates a Pop Error (POP on EMPTY

Filter: Time Range: (0 - 1000000)						
First Fail Ended	First Fail Started	Delta	Instance	Assertion	Offending	
20500	20500	0	fifo_tb.fifo_rtl_1	ap_pop_error_1		
20500	20500	0	fifo_tb.fifo_rtl_1.fifo_props_1	ap_pop_error		
127500	127500	0	fifo_tb.fifo_rtl_1	ap_push_error_1		
127500	127500	0	fifo_tb.fifo_rtl_1.fifo_props_1	ap_push_error		
241530	241530	0	fifo_tb.utest_pgm	'fifo_cmd_xactor:idle_task.a_0 (num_idle_cycles < 4)		

Assertion Failure Summary: / Assertions /

Figure 3.3a Assertion Failure Summary

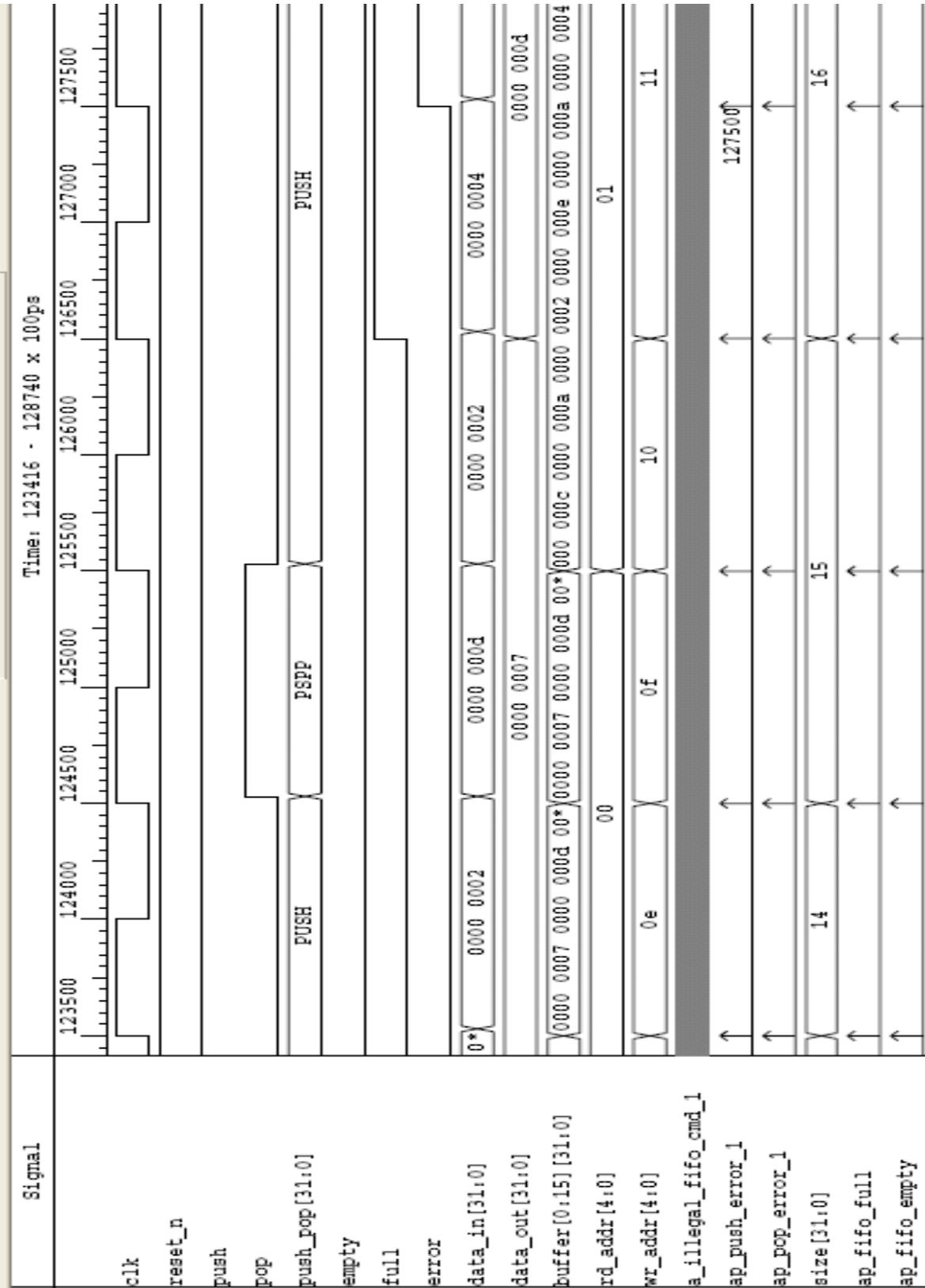


Figure 3.3b Simulation with Push Error (Push on FULL)

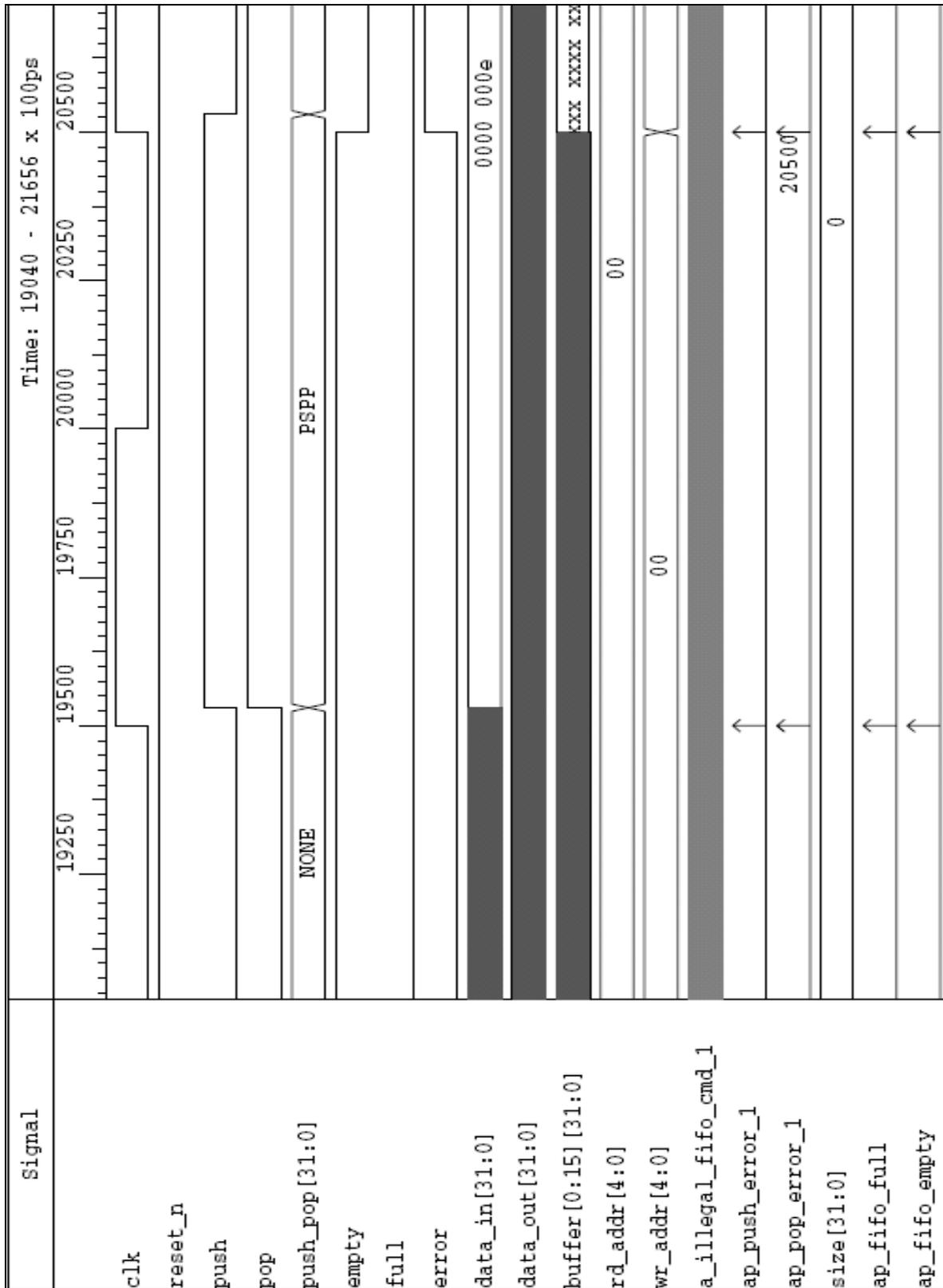


Figure 3.3c Simulation with Pop Error (POP on EMPTY)

3.4 File Structure and Compilation

Table 3.4. demonstrates the file Structure and the purpose of each file. Figure 3.4a 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 3.4b. The file list is shown in Figure 3.4c.

```
all:
    vcs -debug_all -sverilog -f flist +incdir+../ -ntb_opts rvm
run:
    ./simv -gui &
clean:
./rm -fr csrc* simv* scsim* *vpd ag* session* work/* WORK/*
    *.so *.log test* cm* ucli* worklib/*
```

Figure 3.4b. Makefile for Compilation with Synopsys VCS Simulator (file vcs/Makefile)

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

Figure 3.4c. 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 (*fifo_pgm.sv*) had include statements:

```
\include the "vmm.sv"
\include "fifo_pkg.sv"
\include "fifo_xactn.sv"
\include "fifo_env.sv"
```

In addition, the *fifo_env.sv* file has include statements:

```
\include "fifo_log_fmt.sv"
\include "fifo_cmd_xactor.sv"
\include "fifo_gen_xactor.sv"
\include "fifo_mon_xactor.sv"
```

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_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: <code>`vmm_channel (fifo_xactn)</code>	<code>`vmm_channel</code> macro for generation of channel, <code>`vmm_atomic_gen</code> 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 <code>`vmm_atomic_gen</code> 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 <i>run</i> 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

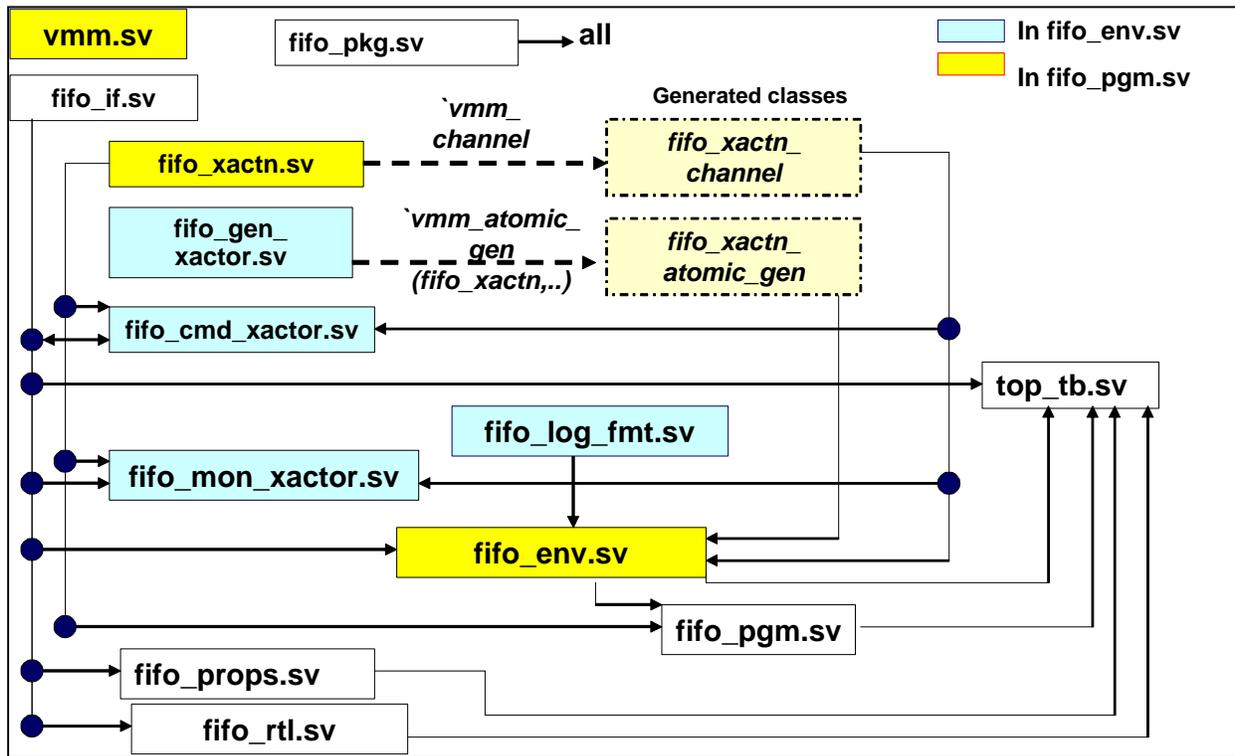


Figure 3.4. File Structure and Relationships

4.0 Conclusions and Recommendations

Our experience with VMM and assertions was very positive because VMM represents a methodology that addresses the important phases of the verification process in a structured manner, along with the potential for easy expansion and reuse. The assertions along with the random transactions did detect an error in the RTL model for the FULL flag. The VMM library and macros do help in the building of the model. However, the application of VMM requires a good understanding of the use of the library elements and macros. This knowledge can be acquired through training, examples, and the use of the *VMM for SystemVerilog* book. We hope that this paper provided a better understanding of the generation and consumption of transactions written in a VMM.

5.0 Acknowledgements

We thank Tim L Wilson from Intel for reviewing this paper and for providing valuable comments.

6.0 References

[1] *Verification Methodology Manual for SystemVerilog*, Bergeron, J., Cerny, E., Hunter, A., Nightingale, A. 2005, ISBN: 0-387-25538-9

[2] http://www.synopsys.com/news/announce/press2005/snps_sourcode_licsvpr.html

Reference Verification Methodology Tutorial, Synopsys documentation 2005

[3] *SystemVerilog Assertions Handbook*, Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, 2005 ISBN 0-9705394-7-9

[4] http://www.abv-sva.org/vmm/snug06_cohen_sri_aji.tar

[5] Diagrams created with StarUML™ - The Open Source UML/MDA Platform available at: <http://www.staruml.com/>

StarUML™ is a software modeling platform that supports UML (Unified Modeling Language). It is based on UML version 1.4 and provides eleven different types of diagram, and it accepts UML 2.0 notation. It actively supports the MDA (Model Driven Architecture) approach by supporting the UML profile concept.

[6] *Design Patterns: Elements of Reusable Object-Oriented Software*

(Addison-Wesley Professional Computing Series), [Erich Gamma](#), [Richard Helm](#), [Ralph Johnson](#), [John Vlissides](#) ISBN 0-201-63361-2

