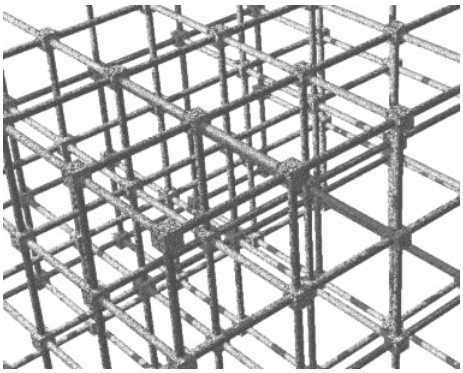
1 VMM FRAMEWORK



New technologies are needed to verify today's ever larger and more complex designs. These technologies came in the form of new languages such as Open Vera, *e* and recently SystemVerilog. However, having a powerful language alone was not sufficient to handle these designs as every design team had to find a way to make use of the extensive list of features in the language to address the problems at hand. New methodologies and frameworks appeared to handle the verification tasks, including Transaction-Level Modeling (TLM)[1], Reference Verification Methodology (RVM)[2], Advanced Verification Methodology (AVM)[3], and Universal Reuse Methodology (uRM).[4] In mid 2005, Synopsys and ARM collaborated and published an open verification methodology built on the strong concepts of RVM, and the result was a document – The *Verification Methodology Manual (*VMM*) for SystemVerilog* – supplemented with a set of SystemVerilog libraries (classes and macros).[5] VMM supports transaction-based verification (TBV), directed verification, coverage-driven verification (CDV), constrained-random testing (CRT), and assertion-based verification (ABV), and proposes an optimal usage of each of these advanced techniques. This chapter addresses the features that SystemVerilog provides in the field of verification, and why VMM represents a viable framework for verification.[6] In this chapter, we present a short review of the SystemVerilog constructs used in testbenches along with an overview of a typical VMM compliant testbench architecture.

---

[1] Transaction Level Modeling: An Overview http://www.ics.uci.edu/~gajski/presentation/Transaction.ppt

[2] Synopsys, http://www.synopsys.com/products/simulation/pioneer/pioneer_ntb.html

[3] Mentor Graphics, http://www.mentor.com/products/fv/news/questa_avm.cfm

[4] Cadence, Incisive Plan-to-closure Methodology http://www.cadence.com/

[5] VMM is an adaptation of RVM for SystemVerilog documented in the book *Verification Methodology Manual for SystemVerilog*, Springer 2006. Additional information on VMM is available at http://vmm-sv.org/

[6] This book addresses SystemVerilog and VMM only and makes no attempt to compare languages or other frameworks.

1.1 FRAMEWORK

What is a framework? The dictionary defines it as:[7]

1. *A structure for supporting or enclosing something else, especially a skeletal support used as the basis for something being constructed.*
2. *An external work platform; a scaffold.*
3. *A fundamental structure, as for a written work.*
4. *A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality.*

*In software development, a **framework** is a defined support structure in which another software project can be organized and developed. A framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project. The word "framework" has become a buzzword due to recent continuous and unfettered use of the term for any generic type of libraries.*

*A **software framework** is a reusable design for a software system (or subsystem). This is expressed as a set of abstract classes and the way their instances collaborate for a specific type of software (Johnson and Foote 1988; Deutsch 1989). Most software frameworks are object-oriented designs. Although designs don't have to be implemented in an object-oriented language, they usually are.*

VMM is a framework for the verification of hardware FPGA and ASIC designs, and is built with SystemVerilog as its supporting language.[8] VMM provides the support environment to create a reusable and extensible testbench in a transaction-level style. As such, you need to understand how to use this framework. In this book we will demonstrate how the various concepts and libraries are used to build such a testbench. These concepts are introduced in this chapter and are explained in the subsequent chapters.

1.2 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). Directed testcases is another methodology that, despite its low productivity rate, continues to be required to exercise deep corner cases or initialization of DUT. A proper methodology must be able to support directed tests – but directed tests should not be the primary verification approach.

Functional coverage can be further divided into temporal coverage (with SystemVerilog assertions (SVA)), and data coverage (with *covergroup*). A good transaction-based verification methodology 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*.

---

[7] From http://www.answers.com/framework

[8] Note: This framework can be implemented in another language. For example, RVM is the same framework implemented using *OpenVera*. VMM could be implemented in *SystemC*.

1.2.1 SystemVerilog Constructs Supporting Verification

A summary of the SystemVerilog constructs supporting verification is shown in Table 1.2.1

Table 1.2.1 SystemVerilog Constructs for Verification

SystemVerilog Construct	Verification Application
Interface and virtual interface	Encapsulates the communication between different components of the design by grouping the signals used for communication, capturing legal and illegal behavior of these signals via assertions, *covergroup* etc.
Class and virtual class	Fundamental building block for an Object-Oriented Verification environment. Builds reusable extensible classes for the definition of constrained-random variables and the collection of supporting tasks related to common objectives.
Constraint	Provides a way to constrain random generation. Pure random generation is almost never useful for practical designs. However, constraints and their associated methods lay the foundation for CRT in SystemVerilog.
Mailbox / Queue	Provides channeling and synchronization of transactions and data. May also be used by scoreboard for verification
Clocking block	Identifies clock signals, and captures the timing and synchronization requirements of the blocks being modeled. Provides a synchronous communication between testbench and design.
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 *Reactive* region. Creates a clear separation of testbench and design, thereby eliminating potential race conditions when the same design constructs are used to model testbenches.
covergroup	Provides a way to measure verification effectiveness by capturing parts of traditional test plan in an executable fashion. 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, and provides functional coverage and reporting of information upon error. Assertions can interact with the testbench.
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 (i.e., create an object) and allocated (i.e., create storage) to be used.
- The *randomize* function can be used to randomize class variables (that are qualified via *rand*; individual randomization of scalar variables is also possible).
- Classes can be typed and parameterized.
- Classes can be passed as objects (instance of class) to methods in other classes and to mailboxes and queues.

- Classes that need to interconnect to physical interfaces can use virtual interfaces that are referenced to the appropriate SystemVerilog interfaces instances (e.g., DUT interface)

With the number of directed test cases exponentially increasing, it is becoming a huge task to scale up that methodology to modern day designs. CRT provides a viable alternative as it puts the burden on the machine rather than the user – the same test run with a different seed creates a different set of scenarios. CDV works hand-in-hand with CRT to monitor the verification progress.

SystemVerilog supports the generation of constrained-random values with the use of the *randomize* function, the *rand* and *randc* type-modifiers, *randcase and randsequence* statements, and the rich sets of constraints with the *constraint* construct.

Coverage is another important ingredient in the verification process because it provides feedback on the progress of the verification effort. SystemVerilog supports 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 sequence, and the slaves being addressed during such an abort condition can be monitored using *covergroup*, which bins or groups the address space. The results of the coverage information may also 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 numbers 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 *event*s. This interaction capability with the testbench can provide the following:

- Write to a variable, thus having the capacity to modify the flow of the testbench.
- 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.
- Upon a failure, write 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.
- SystemVerilog *sequence* can create an *event* when the sequence is finished, and that is very useful to synchronize various testbench elements.

Note that assertions can be written in modules, programs, or interfaces. Assertions are not allowed in classes. However, Chapter 6 demonstrates the capability, via callbacks, to copy class properties (i.e., variables) to a debug SystemVerilog interface. This allows you to write within the debug interface assertions derived from both the DUT interface and copies of the variables defined in the class object.

1.3 WHY VMM?

SystemVerilog is a vast language with a 550+ page LRM (on top of IEEE Standard 1364-2001 Verilog HDL). It is easy to get trapped in its landscape and use it in a sub-optimal way to achieve the end goal - i.e., finding all bugs as efficiently as possible. A good methodology is the best way to use the language to its optimum. Figure 1.3-1 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; speedy creation of a layered and reusable testbench; and access to high-level tests using constrained random stimulus and functional coverage to indicate which areas of the design have been checked.

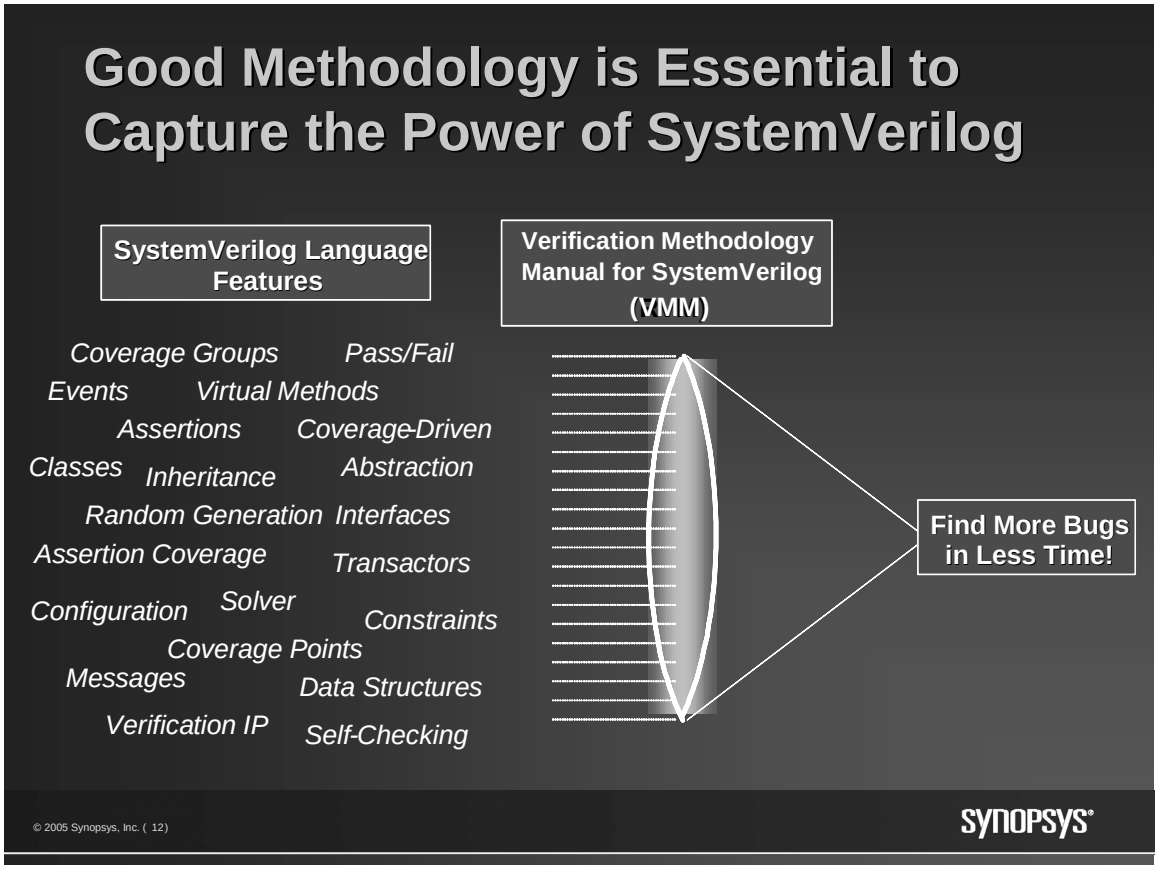


Figure 1.3-1 Impact of VMM Methodology in Capturing the Power of SystemVerilog

The VMM consists of several base classes as shown in Figure 1.3-2, and described in the *VMM for SystemVerilog* book. A brief definition of those base classes is presented below.

vmm_data: This class type is used to build the data model (e.g., transactions). It contains constraints for valid random values and ultimately randomization.

vmm_log: This is a message class, allowing the users to have consistent, flexible, and controllable message processing. This class allows the display of complex messages. It is supported by several macros to facilitate the issuance of messages.

vmm_env: This is the environment manager base class. It controls the instantiation of other classes, resets/starts/stops those classes, registers callbacks, and manages the overall flow of the simulation. It allows a test to be as simple as an instantiation of this class and the invocation of the environment's *run* task.

vmm_xactor: This is the base class used for the generators, drivers (e.g., BFMs), monitors and checkers. It is also optionally used for the scoreboard and coverage classes, as well as for other auxiliary transactor classes.

xvc_xactor: System-level transactors referred to as an extensible verification component (XVC). XVCs provide a foundation for modular, scalable and reusable system-level verification environments, with the aim of minimizing test set-up overhead. XVCs can be used to drive block interconnect infrastructures or external interfaces. They can also support other XVC components by monitoring system state and providing notification information.[9]

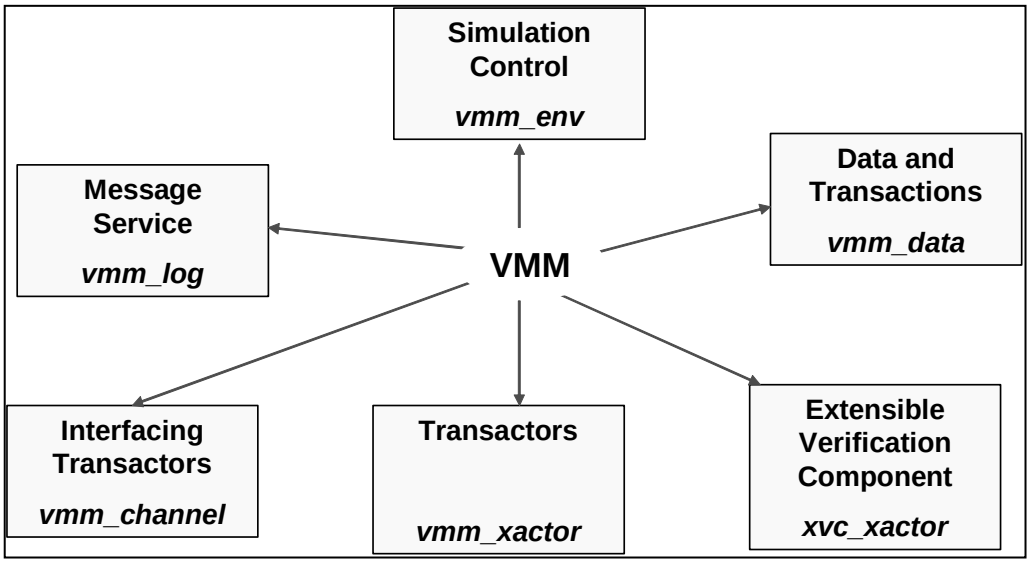


Figure 1,3-2 VMM Basic Base Classes

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. This is explained in Chapter 4 in the discussion of the environment.
2. The methodology used to generate and consume transactions, including the automation with the use of VMM macros. This is explained throughout the book.
3. The methodology and support used to adapt transactions to modifications through factories and callbacks. This is explained in Chapter 5 and 6.
4. The level of support using the various base-class methods. This is explained throughout the book
5. The methodology used to report logging and status information. This is explained and used throughout the book.

[9] XVC is not addressed in this book.

1.4 TESTBENCH ARCHITECTURE WITH VMM

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 the layered testbench concept caused the design of different verification environments even within the same organization.[10] Experience has shown that such heterogeneous verification environments lead to too much redundancy. For example, verification IP developed by one group doesn't fit easily into a slightly different project/environment. A significant amount of effort can be saved when different teams follow a unified methodology in the architecture of testbenches. For this to become reality, a reference verification architecture that is flexible enough to cater to various domains must be developed. VMM is the industry's first such non-proprietary, open, standard language-based verification methodology.

The basic idea of a transaction-level methodology, such as VMM, is to separate the *transaction* from the *transactor*. While there are various definitions for these terms, we define a *transaction* as an operation that represents the job to be performed, such as Read / Write / Idle. Transactions are implemented with a class extended from the *vmm_data* base class. For example, a *transaction* may consist of the following:

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, path, etc.

In VMM, a *transactor* is a generic name, and there are several kinds of transactors such as generators, drivers, monitors, scoreboards, etc. A direct equivalent of a typical VMM *transactor* is what's conventionally known as a BFM (Bus Functional Model) at the lower level. In this book we refer to a BFM type of transactor as a "command transactor" that accepts *transactions* and sends then to the DUT according to the underlying protocol.

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 1.4 where, in constrained-random testing, the transactions defined in a *transaction* class are randomized with a generator and are sent to a *transactor* via a *channel* for the execution of those transactions. A *channel* is an object that holds handles (i.e., links) to transaction objects, and behaves like a queue of handles to those objects. A *channel* supports methods to *put* and *get* transactions form the queue. 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 the next transaction from the *channel* via the *get* method. The *transactor* then proceeds to execute the retrieved transaction. Transactions and transactors are addressed in Chapters 2, 3, and 5.

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 synchronized 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

[10] Note: the layers in the environment and testbench are conceptual more than structural. Everything is still instantiated in a flat manner.

advantage is the ability to generate (and even consume) the transactions with different agents/transactors. This is useful for reusability.

Channels maybe implemented with queues, or mailboxes, or some other data structure, but from a user standpoint they are superior to raw SystemVerilog queues or mailboxes. This is because channels are well supported by a rich variety of methods and classes to create needed verification testcases and environments. For example, *vmm_channel* supports complex requirements in handling transactions in channels (e.g., out-of-order execution model) with methods that let transactors query the execution progress of a transaction directly from the channel itself. *vmm_broadcast* allows multiple consumers to extracting transaction descriptors from a channel, while *vmm_scheduler* lets multiple sources add descriptors to a single channel.[11]

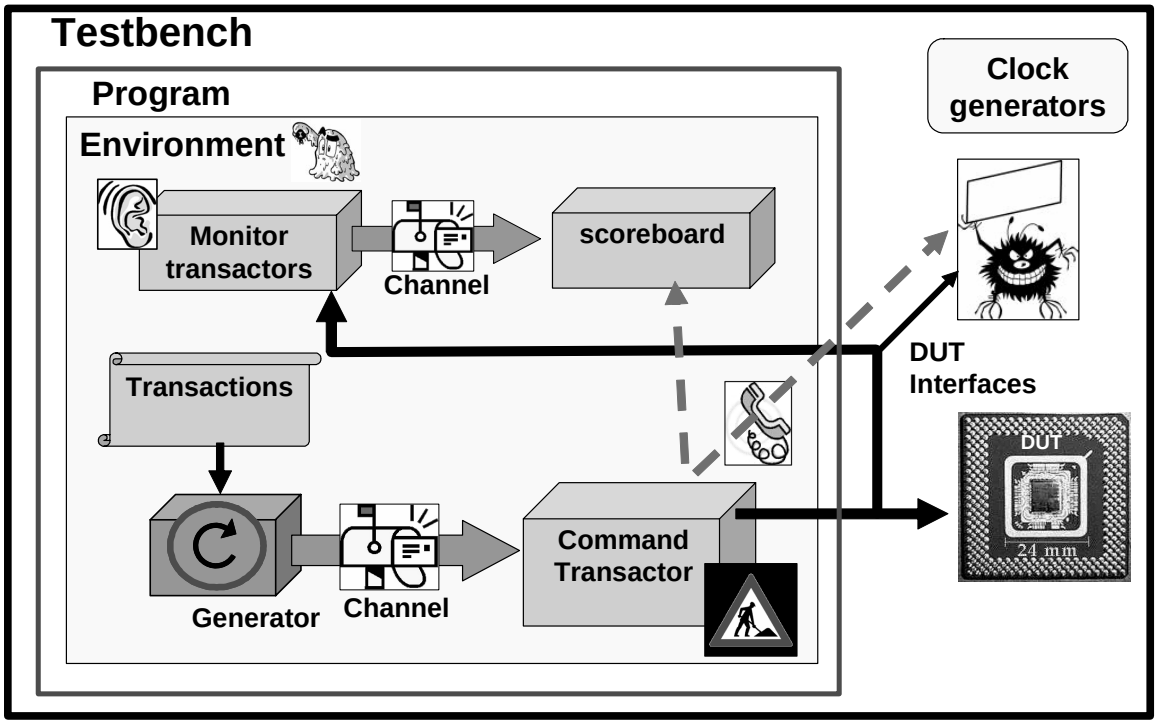


Figure 1.4 High-level View of the Testbench

1.4.1 Layered testbench architecture

One of the key aspects of reusable design is a layered architecture, as shown in Figure 1.4.1. It provides:

- Abstraction at different levels in the verification infrastructure.
- Easy plug-and-play with different levels of DUT abstraction.
- Concurrent development of various Verification environment pieces.

[11] Chapter 2 addresses channels, while channel 7 covers advanced topics in the use of channels.

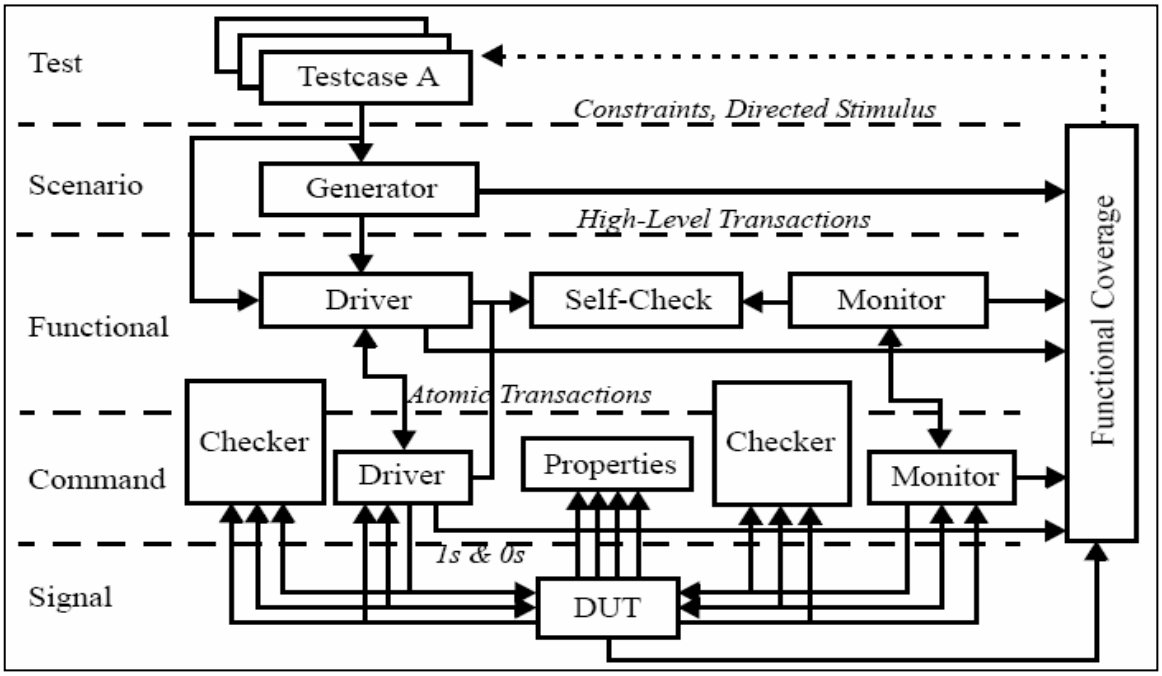


Figure 1.4.1 Layered Verification Environment Architecture

1.4.1.1 Command Layer

The command layer (a.k.a. Bus-Functional Model (BFM) layer) is the lowest layer where bit-wiggling per the bus interface protocol is defined. This layer is the least reusable (except for some standard protocols like PCI, AHB etc.). Examples: signal-level details of tasks such as read, write, drive a packet, etc.

1.4.1.2 Functional layer

This is an optional layer above the BFM layer that models the "functionality" of the system, and has no knowledge of the bus interface protocol. For example:

- Perform a DMA transfer.
- Store an image to memory (e.g., that is processed through PCI or AHB in the BFM).

1.4.1.3 Generation/Scenario layer

This layer is responsible for the generation of meaningful, interesting scenarios, which are sequences of transactions. For example, considering a *SoC* with multiple peripherals, a scenario can be a USB transfer of an image from an external memory card followed by the storage of that image onto a CD. To simulate this kind of scenario, one needs to generate first a *write-to-mem* via USB, then a *read-from-memory* via a CD-Interface. However, the requirement is that the address in the second transfer should be the same as the previous transfer (to put that same image onto the CD). Note that in a directed test, this layer is temporarily bypassed or altogether absent.

1.4.1.4 Interaction between the different layers

One of the challenges in building a layered environment is to decide how the layers communicate. A direct reference to the other layer will break the rules of reuse, maintenance, concurrent development etc. One way to prevent this is to have a generic medium of interlayer communication that provides isolation between the producers of transactions and the consumers

of those transactions. VMM channels are excellent candidates for such a requirement. At every layer, the producer can put transactions into a channel while the consumer can extract transactions from this channel. These channels are local to individual layers and hence completely independent of each other. These can be initialized through a constructor (e.g., the *new*() function in SV). A final environment layer can hook up individual layers and connect their channels accordingly.

1.4.2 Testbench Outline

Figure 1.4.2 represents a structural view of the testbench. The testbench includes the following objects:

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

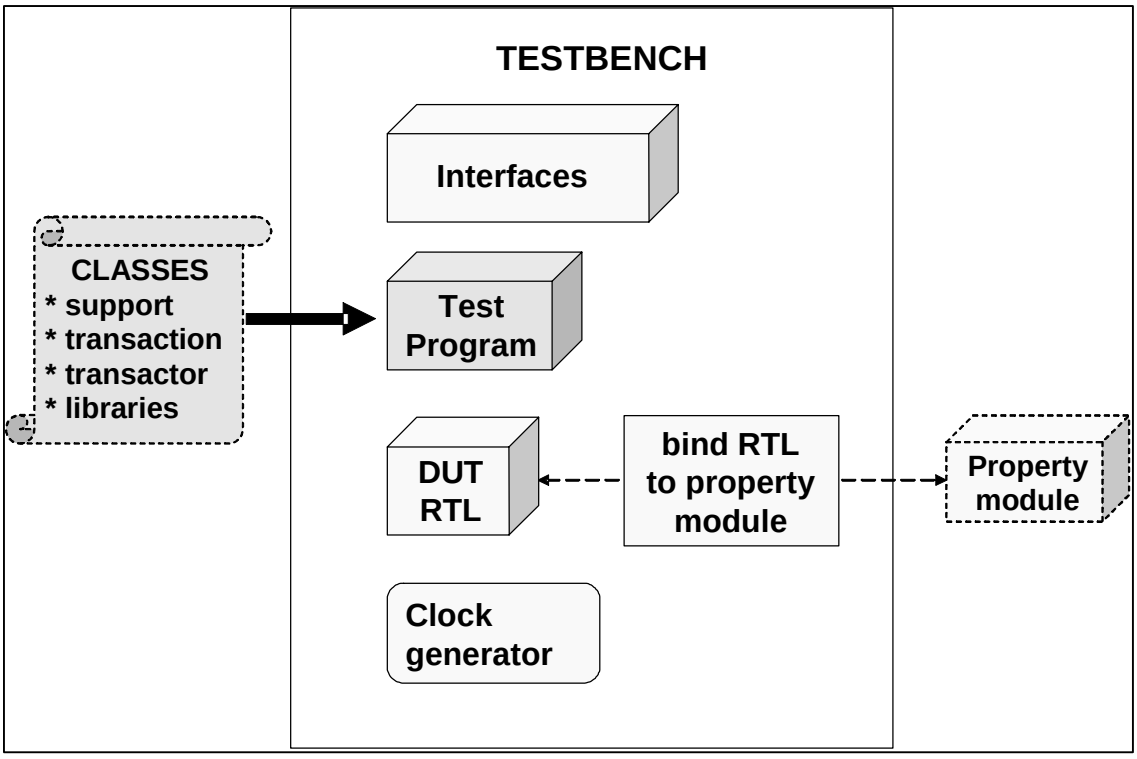


Figure 1.4.2 Testbench Structure

Chapter 4 provides details about the modeling of the environment and the testbench.

Chapter 1 Questions[12]

Q1. Why does VMM, based on SystemVerilog, use an object-oriented (OO) approach? Why are classes used instead of modules?

Q2. Why is SystemVerilog a suitable language to create a verification framework?

Q3. Why is a framework such as VMM useful for the design of testbenches?

[12] See Appendix A for answers to questions