Appendix A Q/A

Chapter 1, VMM FRAMEWORK

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

A1. . Using object-oriented approach has proven to be highly successful in managing highly complex projects in the Software domain, and given the rising complexities of hardware verification, lessons can be learned from the Software domain. Hence use of OO programming for verification is becoming the only viable way to create, maintain reusable testbenches. Specifically, OOP provides inheritance, polymorphism and virtual methods so that you can modify the behavior of a verification environment without having to modify what already works. On the specific question on the use of *class* versus *module* instances in SystemVerilog , Janick's reply to the Verification Guild posting entitled *Why VMM base on OO SV?*[1] provides an interesting summary. The advantages of using *class* instead of *module* or SystemVerilog *interface* instances include:

1. Classes can be instantiated dynamically, thus the structure of the testbench can be decided at runtime (e.g. based on a randomly-generated DUT configuration). The closest that *module* comes to meeting this requirement is with the *for-generate* and *if-generate* constructs. However, these are elaboration time features and the final structure gets frozen before any code has had the chance to run. Hence any "random" configuration will require recompilation/elaboration thereby impacting productivity.
2. You can have a base class that enforces consistency across all components (e.g., they all have a "*start*" function). You cannot have base modules.
3. You can have a base class that provides generic functionality to all modules (e.g., the messaging service). You cannot have base modules.
4. Classes can all be derived from a common base class, so you can create generic functionality. For example, you can maintain a queue of all transactors in an environment so you can easily start them all by iterating over the queue instead of having to know where each and all of the transactors are.
5. You can pass a reference to a class instance around. For example, a scoreboard could know which transactor is producing a specific transaction. You can't do that with modules.
6. You can extend the behavior of a class and modify it. For example, this is needed to add error injection, sample data for functional coverage, or integrate a scoreboard. You cannot do that with a module.
7. Randomization ease - Classes can be randomized at once thus providing a complete random transaction. However, module elements need to be randomized individually with a call to *std::randomize()* on every element. Classes also provide convenient way to manage declarative constraints using SystemVerilog's *constraint* blocks.

---

[1] http://verificationguild.com/modules.php?name=Forums&file=viewtopic&p=5238#5238

8. Queue of classes – A testbench is more of a dynamic entity, and many a times we do require an array/queue of transactions, descriptors etc. Module and interfaces are static and don't lend themselves easily to such requirements.
9. For efficiency purposes, you can pass handles of the *class* instances to other objects without creating duplicate copies all across the system.
10. Better memory usage, as Garbage Collection mechanism is built-in in SystemVerilog for class-based systems.

Using modules to build a verification environment looks easy at first glance, but they are difficult to maintain because you constantly need to modify them to adapt to changes to the verification lifecycle. That is also true for interfaces; however, with SystemVerilog virtual interfaces you can create references to their instances).

Q2. Why is SystemVerilog a suitable language to create a verification framework?
A2. Section 1.2.1 addresses the capabilities of SystemVerilog constructs for verification that supports OO programming.

Q3. Why is a framework such as VMM useful for the design of testbenches?
A3. Section 1.3 addresses the concept of a framework that facilitates the design of testbenches through quick build, reuse, flexibility, and extendibility.
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 methodology used to report logging and status information. This is explained and used throughout the book.

Chapter 2, VMM TRANSACTIONS AND CHANNELS

Q1. Why are transactions specified in a *class* instead of a *struct* within a transactor?
A1. Defining transactions in *classes* offers the following advantages:

1. It follows the Transaction-Level Modeling recommendations.
2. It allows the transactions to be extended with class extensions, thus extending the properties and constraints of the transactions.
3. It allows flexibility of use in defining in the environment which class to use as the transaction and channel.

Hardwiring the generation of transactions to a *struct* within the transactor has several disadvantages, including:

1. Lose flexibility in picking and choosing constraints and generator.
2. Creates a too closely bound verification environment with little flexibility.
3. *struct* cannot be extended, inherited etc. Basically the *struct* is not a fundamental building block of an OOP-based technique.
4. It cannot contain methods (tasks/functions). As demonstrated in Chapter 2, it is very logical to associate with transactions certain transaction specific methods (e.g. copy, display etc.).

Q2. Why can't transactions specified in a class be used without the need of a channel?
A2. The answers to Q1 also apply to this question. Channels are point-to-point data transfer mechanisms. Channels provide a separation between the generation and the consumption of the transactions. Also, channels provide additional flexibility when the requirements change. For example, if multiple consumers are extracting transaction descriptors from a channel, the transaction descriptors can be distributed among the various consumers using *vmm_broadcast*. Another distribution scheme with *vmm_scheduler* fullfills the need to distribute transactions in a controlled scheduling scheme from multiple channels connected to multiple transactors to a single target/destination.. Those transactors may drive a multi-bus system to achieve high bandwidth in the transmission of data. The use of *vmm_broadcast* and *vmm_scheduler* are addressed in Chapter 8.

Q3. How do you build a custom channel?
A3. There is no need to build a custom channel. This is the responsibility (and advantage) of the framework. To build a channel off a known transaction just use the macro *`vmm_channel(*class_name). However if you still insist to create one, SystemVerilog provides constructs such as queues, mailboxes and dynamic arrays to support building reusable channel like objects.

Q4. If I extend my transaction class (e.g., class Fifo2_xactn extends Fifo_xactn) do I also need to define and use a new channel?
A4. No, you do not need, nor should you, create a new channel. Using virtual methods, all code that is written for *Fifo_xactn* will be able to deal with *Fifo2_xactn* subclass because the latter is derived from the former. The virtual method will be used to implement transaction-dependent functionality (such as display or packing). As per general OOP, a channel of base class type, *Fifo_xactn* in this case, can carry its derived class objects as well.

Chapter 3, Transaction Generator, Command Transactor, and Monitor

Q1. Why does the transaction generator send the transactions to the channel instead of directly to the transactor?
A1. If the generator sends the transactions directly to the transactor, then a synchronization scheme would be needed for the insertion of the transactions into the transactor. There are several such techniques that can be used, such as the use of notifications. However, this implies that the generator must know the instance name of the transactor. This reduces the reusability and flexibility of the transactors and generators. Requirements do change, and there is a need for the testbench to easily adapt to those changes. For example, if the design requires redundancy with multiple instantiations of transactors, the separation between the generators and the transactors with the channels allow for such adaptation. Channels can quickly adapt to such a new environment with the use of *vmm_broacast* or *vmm_scheduler* (addressed in Chapter 8).

Q2. Why is a transactor extended off the base class *vmm_xactor*?
A2. The *vmm_xactor* provides many needed functions, including:

- Interaction with the execution of steps in the environment, as described in the VMM book on page 127. This includes the *start*, *stop*, and *reset* of the transactors.
- Provision of the logging services.
- Provision of the notification services.
- Service to the *`vmm_callback* macro (see Chapter 6).

Q3. What role does a monitor play?
A3. A monitor is a transactor extended from *vmm_xactor* class, and extracts transactions as observed on the DUT interfaces. Those transactions are then transferred to a scoreboard for verification through either a channel or through a notification, as explained in Chapter 7.

Chapter 4, Building the Environment and Testbench

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

A1. The user environment must be defined as an extension of *vmm_env* to take advantage of the automations provide by the base class.

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

A2. Using *module* for both design and verification was shown problematic in Verilog because of race conditions. SystemVerilog addressed this race condition issue by introducing a separate, exclusive timeslot for testbench code inside the *program* block as part of the regular event scheduling mechanism. Code specified in the *program* block is executed in the Reactive region separate from the usual Blocking and Nonblocking assignments of the design code. Since this separation is provided by SystemVerilog, all compliant tools/implementations should adhere to those timing execution rules, thereby eliminating those timing race conditions and enhancing portability. Hence, it is good practice to execute the environment in a *program* block.

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

A3. The testflow is initialted by the *vmm_env::run()* method. The flow is very important because it controls the steps involved in the creation of the verification process from configuration to starting the transactors to waiting until completion of end, and to stopping the transactors, cleanup and the final report. The testflow in the environment is fundamental to VMM. This testflow is also very flexible and allows user's intervention at various stages, as explained and used throughout this book. Use of such generic testflow helps in the debugging, and the transfer of ownership of a verification project to different team etc.

Chapter 5, Using the Factory Pattern

Q1: When should a factory pattern be used in a transactor, such as a generator?
A1: A factory pattern is recommended when you anticipate a change in an object allocated by the transactor. In this chapter we presented two examples, one with the transaction where the constraints were changed; and the other example with an error injector where the implementation for the error algorithm is redefined in a subclass. All generators should use factory pattern.

Q2. When would you use an atomic generator as created via the macro *`vmm_atomic_gen?*
A2. The *`vmm_atomic_gen* macro is a very convenient technique to generate a transactor generator with the simple definition of a transaction class with properties qualified with the *rand* qualifier. If this is a good match for the problem at hand, then you struck gold! Even if the problem requires something more complex than a traditional atomic generator, it is often useful to create this quick generator to allow a quick first pass test of the DUT and the test environment. Often, this step allows for the detection of infant mortality design failures (i.e., design failures that can easily be detected with a few simulation cycles).

Q3. Why is a custom generator sometimes needed?
A3. There are situations where specific transactions and data need to be processed by the DUT. For example, a DUT that performs signal processing may need to process a specific set of image data generated by an external program, and the monitored results of this processing need to be compared against expected results, also generated by an external program.

Q4. Why do generators use the *copy()* method while monitors use the *allocate()* method to create a new object?
A4. Generators use the ***copy()*** method to send a copy of a transaction object into the channel, thus maintaining the original transaction pristine. This provides a separation of handles between the original transaction manipulated by the generator and the consumer that extract the copy from the channel. In the manipulation of the transaction, the generator may also need the value of the current transaction to create the next transaction.

However, a monitor needs to allocate a new transaction object to store observed data and then put this object into the channel. The monitor has no further need for this object.

Chapter 6, Callbacks

Q1. What are good user applications of callbacks?
A1. The following represent some examples:

- Error injector
- Modify or add transaction
- Pass information to a scoreboard
- Pass information to a SystemVerilog interface
- Execute a task prior or after a normal flow of another task. For example, before sending a packet, do a callback to update the parameters of the packet.

Q2. Do the VMM base classes have callbacks? Which ones? And why?
A2. VMM provides quite a few callbacks in the base classes. For example, in the *vmm_xactor* class VMM provides:
The *vmm_atomic_gen* automatically creates the <*class_name*>*_atomic_gen_callbacks* (e.g., *fifo_xactn_atomic_gen_callbacks)* to implements a façade for atomic generator, transactor, callback methods.[2] An example of a callback is

```
virtual task post_inst_gen(<class_name>_atomic_gen gen,
                           <class_name> data,
                           ref bit drop);
```

This callback method is invoked by the generator after a new transaction or data descriptor has been created and randomized but before it is added to the output channel.

Q3. What are the advantages of transferring information from a transactor via callback to an interface?
A3. Transferring information from a transactor via callback to an interface provides the following advantages:

1. The display of the transactor variables onto the interface. This allows the waveform visualization of those variables, thus facilitating the debug and documentation of the design.
2. The use of temporal assertions. Temporal assertions are not allowed in classes, but they are allowed in interfaces. Thus, by copying class variables into interfaces, you can use those copies in the assertions.
3. The verification of the transactions. The debug interfaces can also include as inputs the DUT interfaces. This allows you to write assertions that not only deal with the verification of the DUT, but also with the verification of the transactors to insure that they abide to the bus protocols

[2] VMM Rule 5-11 Generators shall provide a procedural interface to inject data or transaction descriptors.

Chapter 7, Custom Generator and Notifications

Q1. When is a custom generator needed?
A1. There are situations where specific transactions and data need to be processed by the DUT. For example, a DUT that performs signal processing may need to process a specific set of images. The generator may call C routines or read files generated by an external source to collect the data for the transactions. If the generator has access to externally computed results, it may also send the expected results to a scoreboard though another channel.

Another example, recently reported on the Verification Guild, required the generation of random CPU instructions with address-related constraints in response to a CPU fetch. One way to implement this requirement is to generate the instructions on the fly, i.e., embed the instruction generator in the code memory and generate an instruction whenever a fetch is performed. In VMM, you would build a reactive transactor with no transaction generator or channel. You could still have a transaction class with constraints that the reactive transactor could randomize and use to generate instructions. Factory and/or callback design patterns could also be used within that reactive transactor.

Q2. When should notification via the *vmm_notify* be used?
A2. The *vmm_notify* should be used to provide synchronization between transactors or between a transactor and the environment. It can also be used to carry *vmm_data* subclass object as status of the notification to a scoreboard. An example in the use of the notification to transfer information between transactors is shown in section 7.1.2, task *Fifo_custom_gen::push(word_t data).* This technique is also addressed in the Verification Guild under the heading "RVM Scoreboards".[3]

[3] http://www.verificationguild.com/modules.php?name=Forums&file=viewtopic&t=1009

Chapter 8 ADVANCED TOPICS

Q1. Why are scenario generator needed? Why can't atomic generator be used instead?
A1. As addressed in section 8.2, an atomic generator generates independent transactions based upon a set of constraints. However, there are situations where specific sequences with constraints are needed. No matter how many atomic transactions are generated, it is very unlikely that you would satisfy such requirements because the specific set of sequences are not likely to randomly occur.

Q2. What technique can be used to define a constraint for a sequence that has 3 PUSH, 2 IDLE, 4 POP instructions?
A2. Section 8.2 demonstrates the use of iterative constraints to achieve such a goal.

Q3. How do I repeat a scenario few times?
A3. Section 8.2 addressed this. The <*class_name*>*_scenario* generated by the *`vmm_scenario_gen* macro has a property named *repeated* that enables users to repeat a specific scenario.

Q4. Why do I get a default atomic transaction when I use *`vmm_scenario_gen* macro?
A4. Section 8.2.2 addressed this. To keep the design of a scenario generator completely flexible and generic, VMM creates a default scenario of *length* == 1, which is essentially an atomic transaction (Refer to Figure 8.2.2-1).

Q5. I created a scheduler using *vmm-scheduler* – what is the quickest way to verify its behavior without going through very many DUT simulation cycles?
A5. Many a times, to verify the operation of scheduling, broadcasting etc. one doesn't require the DUT. A quick look at the output should indicate the transaction mix. This is where an artificial sink is very useful. Refer to Section 8.3.3

Q6. In VMM scheduler, how do I get a pure random scheduling (not the default round robin scheme)?
A6. VMM scheduler has an election mechanism that allows you to do this. *vmm_scheduler::randomized_sched* is an instance of *vmm_scheduler_election* class. This *vmm_scheduler_election* class has a constraint named "*default_round_robin*" that one can set the *constraint_mode(0)* on. See pp 405.

Pseudo-code:
```
program p;
  my_env_0.my_scheduler.randomized_sched.default_round_robin.constraint_mode(0);
..
endprogram : p
```