Doulos Verification TechNote 3: Observing Activity in VMM and OVM Testbenches

Welcome...

to *Doulos Verification TechNotes*, an occasional series of articles on topics that we at Doulos hope will be of interest to anyone involved in verification of digital designs. Rather than trying to duplicate the plentiful tutorial and reference material that's already available, we wanted *TechNotes* to take a thought-provoking sideways look at some of the issues we think are most interesting in the world of verification. We hope you'll agree.

Verification TechNote articles are backed up by simple working code examples on our web site, where you can also find downloadable PDF copies of the articles themselves. They are available, together with many other SystemVerilog resources including conference papers and tutorial examples, at

www.doulos.com/knowhow/sysverilog

You can also find information about worldwide availability of our training courses featuring SystemVerilog, OVM and VMM, along with online sales of the highly respected *Golden Reference Guide* series, at

www.doulos.com/systemverilog

Feedback

We welcome feedback on the content of these TechNotes. If you have any comments, or ideas for topics you would like to see in future editions of *Verification TechNotes*, please contact us by email at info@doulos.com.

Copyright © 2009 by Doulos. All rights reserved.

All trademarks are acknowledged as the property of their respective owners.

Information in this booklet is provided "as is" and without warranty of any kind.

You are welcome to make a reasonable number of copies of this material for your own personal use or to share with colleagues, but any copy must include the Doulos logo and the whole of this copyright notice.

Copyright © 2009 by Doulos. All rights reserved.

Verification TechNote 3

Observing Activity in VMM and OVM Testbenches

This *Doulos Verification TechNote* takes a look at techniques for moving data around a SystemVerilog testbench built using VMM or OVM. We'll focus on what happens when you want to extract information about device-under-test (DUT) activity and pass on that information to other processing blocks elsewhere in the testbench.

OVM and VMM both provide flexible, straightforward mechanisms for capturing this observed information and sending it to other parts of the testbench – but, confusingly, each provides more than one method! We will look closely at all these techniques and decide which is most appropriate for dealing with observed or monitored data that must be collected or analyzed elsewhere in the testbench.

Working example code illustrating the ideas described here can be downloaded from the Doulos web site:

`www.doulos.com/knowhow/sysverilog`

Copyright © 2009 by Doulos. All rights reserved.

For example...

Suppose you're verifying a nifty new video compression unit for video-over-Ethernet. Your testbench will generate a stream of video images and feed it to your DUT. It will also monitor the output data, which takes the form of a stream of TCP/IP packets.

Your company has been in the video processing business for quite a while, so you already have verification components for the video and packet-data interfaces. Those verification IP blocks have proven their worth on earlier projects, so you don't want to mess with them. But you need to pull out the data they observe on their respective interfaces, so you can feed that data into your reference model – rather like this diagram:

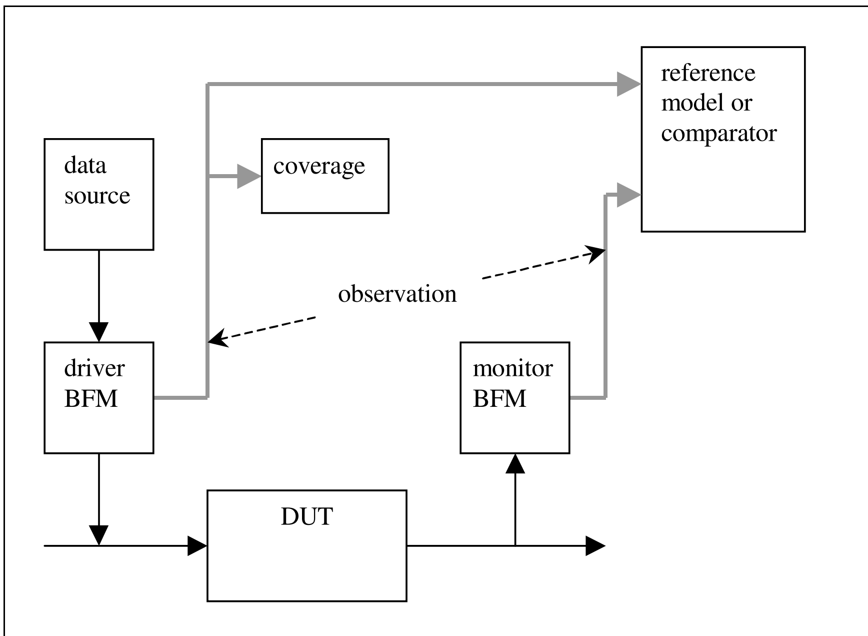

Figure 1: Outline of testbench for typical DUT

Copyright © 2009 by Doulos. All rights reserved.

There are many details missing from this picture. In particular, the precise form of the existing verification IP blocks will depend on whether you use OVM or VMM as your testbench methodology. But one thing is certain: when those blocks were written, the writer definitely did *not* know that they were going to be used in precisely this configuration, with precisely this reference model and coverage analysis block. How, then, can we hook-in to those components to extract the information our new testbench will need?

Not surprisingly, the answer depends strongly on your choice of VMM or OVM. However, the central problem is the same in both cases: *how do I pick up observed data from an existing verification component, without disturbing the code of that component*?

Requirements for observation

As we have already mentioned, we most definitely do *not* want to touch the code of the existing verification components. They are already doing a great job for us, driving stimulus on to the DUT (in the case of the driver block) and checking that the DUT is correctly obeying the relevant interface protocols. As part of this work, they will of course observe all activity on the interface. That makes it easy for them to reconstruct appropriate data objects (transactions) representing the information flowing through that interface. But what should the verification component do with each observed transaction?

Sending the transaction to another part of the testbench

Monitored transaction data is not much use if it remains hidden away inside a verification IP block (VIP). It's important for the block to send it out into the testbench, so that other components (like our coverage block and our reference model) can see it. However, the VIP was written without any knowledge of *where* the data should go. Should it be sent to one external block? Two? None? What kind of block expects to receive the data? Obviously we need a very flexible way to get transaction data out of such a VIP. Not surprisingly, the Big Two verification methodologies OVM and VMM have established ways of doing exactly that. As we shall see, in both cases there is a standard way to convey transactions around a testbench structure, but in both cases that method is not ideally suited to the needs of observation. Consequently they both provide alternative methods (two such methods, in the case of VMM) that are a better match to the requirements for observation.

Copyright © 2009 by Doulos. All rights reserved.

Both methodologies assume that you will build up your testbench from blocks, known as *components* in OVM and *transactors* in VMM. Of course, it is essential that the connection mechanism be very general-purpose so that you can design those blocks independently, without knowing in advance how they will be used – the blocks should be highly *decoupled*. The way that decoupling is achieved is one of the central differences between VMM and OVM.

In the remainder of this TechNote we will survey the component-to-component communication mechanisms in both OVM and VMM, and highlight how these mechanisms can ease the addition of observer components to your testbench.

Copyright © 2009 by Doulos. All rights reserved.

Moving data around an OVM testbench

Each interface to the DUT needs a monitor to observe its activity. Interfaces that will be stimulated by the testbench also need a driver to provide DUT inputs, and a stimulus generator to provide a stream of transactions for the driver to use. In the world of OVM this arrangement, of a monitor together with an optional driver and stimulus generator, is usually bundled into a kind of super-component known as an *agent.* Figure 2 shows the OVM agent architecture in outline. The shaded area enclosing the sequencer and driver blocks indicates that those blocks are provided only for an instance of the agent that will generate stimulus - an *active* agent. An instance that does not generate stimulus is known as a *passive* agent and has only the monitor component. Agents are designed so that the choice of active or passive configuration can be made at run-time, under the control of a command-line option or other configuration data.

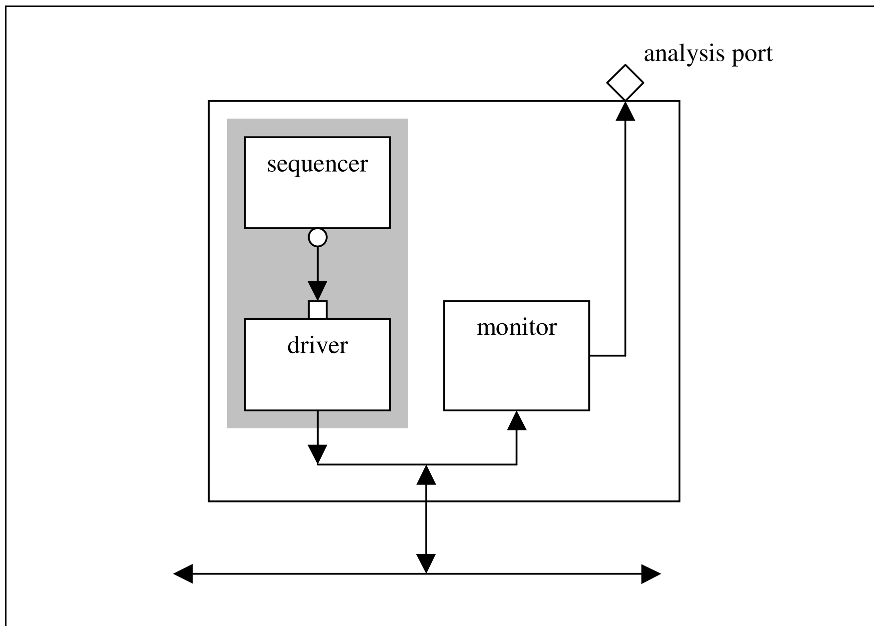

Figure 2: OVM *agent* architecture

Copyright © 2009 by Doulos. All rights reserved.

Whenever you read about OVM you are sure to come across the phrase *transaction-level modeling* (TLM) or *transaction-level connection*. In this Verification TechNote we assume you're familiar with the key ideas of TLM. If you would like an introduction or refresher on this important topic, take a look at our TechNote *Making Sense of Transaction Level Modeling in OVM*.

The connection between sequencer (stimulus generator) and driver (BFM) in this agent is of course a TLM connection. Whenever the driver needs another transaction data item, it calls the `get` method in its sequence item *port*, indicated by a square bubble on the diagram. At the other end of the TLM connection, the sequencer provides a `get` method and makes that method available through its sequence item *export* (the round bubble on the diagram). Finally, the enclosing agent takes responsibility for linking the port and export together, using OVM's built-in connection mechanism. The overall effect is that the driver can call a `get` method in the sequencer, *but neither component knows any details of the other*. Each component thinks it's working through its own port or export, and is completely independent of what is connected to it.

TLM connections like this are ideal for testbench applications where the overall structure of verification components is known. Each TLM connection is point-to-point, connecting just one port (on the block that calls a method) to one export (on the block that provides that method's implementation). That does not fit comfortably with the needs of observation, where it may be essential for more than one component to observe the same set of data. For example, in Figure 1 our stimulus source must send its observed data not only to the reference model but also to a coverage collection block. We need the observed data to be *broadcast* so that any number of such components (including, possibly, none at all) can connect to a monitor so that all those components receive the same observed data.

To see how OVM manages broadcast of observed information from one component to others, we must shift our attention to the open diamond shape at the top right of Figure 2, the *analysis port*. Whenever our agent's monitor observes a new transaction, it calls the `write` method of this port. *Any* other testbench component that is connected to this analysis port can see the observed transactions.

Copyright © 2009 by Doulos. All rights reserved.

What's the difference between analysis ports and port/export connection?

That's not really a fair question, because OVM's analysis ports are just one special case of the overall port/export connection mechanism. However, analysis ports are sufficiently different from ordinary TLM ports that they can almost be considered to be a distinct mechanism in their own right.

Conventional TLM ports are point-to-point; each port is connected to precisely one export, and *vice versa*. By contrast, analysis ports are *broadcast*: a single analysis port can be connected to zero, one or many analysis exports (but each analysis export connects to precisely one analysis port). Figure 3 shows some of the possibilities.

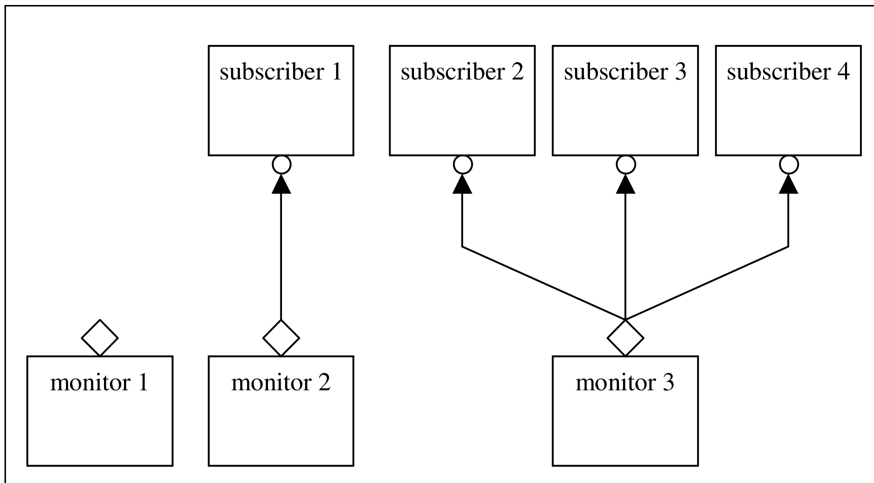

Figure 3: Analysis ports are broadcast

Recall that each monitor will call its analysis port's `write` method whenever it has a new observed transaction. In the case of monitor 1, this call has no effect because there are no connected subscribers.

When monitor 2 calls its port's `write` method, the call is automatically passed on – through the subscriber's analysis export – to a `write` method implemented in subscriber 1, with the observed transaction as an argument to the method. This method can, of course, do anything the subscriber needs it to

Copyright © 2009 by Doulos. All rights reserved.

do. It could log the transaction to a file, gather coverage, or compare the transaction against an expected value from a reference model.

Finally, when monitor 3 calls its analysis port's `write` method, the corresponding `write` methods of *all three* connected subscribers are called with the same transaction argument. The monitor is completely unaware of this; it simply calls its own port's `write` method and everything else happens automatically.

Timing of put/get and analysis connections

We have seen that OVM analysis ports are *broadcast*. A monitor or other observer can equip itself with an analysis port, and call that port's `write` method whenever it has data that might be interesting to other parts of the testbench. Any number of subscribers can now be connected to that analysis port, and all subscribers will see the same observed data.

There is, though, another sense in which analysis ports are special. When data is written through an ordinary TLM port, the writer (producer) will use either the `put` or `try_put` methods of the port. Using `put`, the producer will be stalled while the consumer's `put` method implementation does its work. When the consumer's `put` method eventually returns, the producer's `put` call will complete in its turn. This arrangement allows for blocking, synchronous communication between components, with a guarantee that the data has indeed been conveyed by the time the `put` call returns.

However, the producer might have other things that it wishes to do while the consumer goes about its business. For such situations OVM provides *nonblocking ports*. Typically a data producer would call its port's `try_put` method. This method is guaranteed to return without delay, but it will return a true/false result indicating whether the put attempt succeeded. If it was successful, we know for sure that the consumer has the data (perhaps stored in an internal buffer); but we do *not* know whether it has finished dealing with that data, and we probably need some other interaction to find out when the consumer is done. If the `try_put` attempt failed, we know that the data was not transferred; our component should then go away and do some other work before trying again.

For an analysis port the story is rather different. Its `write` operation is non-negotiated: it can never block or consume simulation time, and it has no notion

Copyright © 2009 by Doulos. All rights reserved.

of success or failure. The only guarantee it offers is that the `write` method in every connected analysis export is called with the same argument data.

Rules for using analysis data

This special behavior of the analysis port imposes some obligations on its data consumers (subscribers):

- A subscriber's `write` method must be a function, and therefore must execute in zero simulated time. Similarly, a subscriber must be ready to accept new data at any time. In many cases this can be achieved by building a suitable FIFO buffer into the subscriber.
- A subscriber must never modify the contents of the transaction object it is given, because the same object may be passed on to other subscribers.
- If a subscriber intends to store the transaction object, it must create and store a copy. There is no guarantee that the analysis data will remain valid after the `write` call has completed. A subscriber that stores a written transaction, without copying it, will probably find that some other part of the testbench has corrupted that transaction by the time the subscriber gets around to using it.

None of these requirements is especially onerous. Respecting them is essential, though, because the effects of getting them wrong can be subtle and extremely hard to detect.

Copyright © 2009 by Doulos. All rights reserved.

Moving data around a VMM testbench

In the world of VMM, connections between verification components - known as *transactors* in VMM - are handled in a rather different way than in OVM. Decoupling of transactors is primarily achieved by means of *channels*, which provide the standard way for transactors to communicate with one another. But there are two other mechanisms available: *notification* and *callback*. As we shall see, any of these three mechanisms can be used for observation, but some are more appropriate or convenient than others.

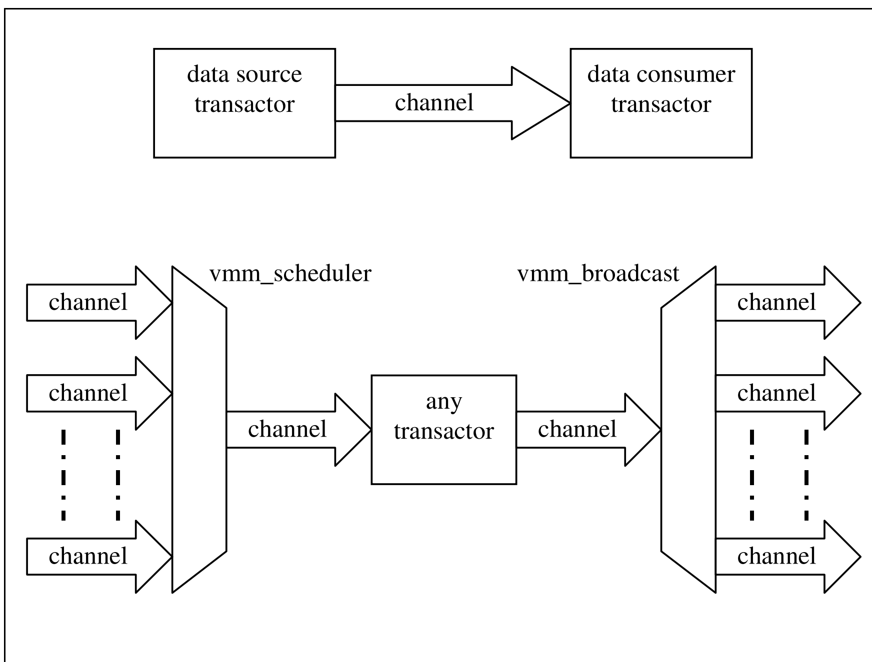

Figure 4: Channel communication in VMM

Figure 4 shows various ways of using channels in VMM. In each case, the transactor at each end of the channel has a public data member that is a

Copyright © 2009 by Doulos. All rights reserved.

reference to the channel; in this way, the enclosing environment can easily set up the required connections.

The upper diagram in Figure 4 shows the most straightforward use of a channel, to convey data from producer to consumer transactor. Even in this simple arrangement, channels offer a rich set of options. The channel is basically a FIFO for transaction objects, but it has many interesting and useful features that we don't have space to investigate here. In many situations it is appropriate to limit the channel's FIFO depth to only one, so that the producer and consumer are tightly locked together.

The lower part of Figure 4 shows how channel data can be replicated across multiple channels using the two specialized built-in transactor classes `vmm_broadcast` and `vmm_scheduler`. As its name suggests, `vmm_broadcast` simply replicates the incoming channel's data across all its output channels (although even this simple operation is highly configurable). `vmm_scheduler` allows two or more incoming channels to compete for slots in its outgoing channel – it is a multiplexer. Again, as you might expect, its operation is highly automated and configurable.

Using a channel for observation

An observation component such as a monitor BFM can use a channel to deliver its observed data. However, some care is needed. A monitor BFM is by nature purely passive, and has no way to throttle or otherwise delay the flow of information on the physical connection it is monitoring. Consequently it is essential for the monitor to deliver each observed transaction to its output without any delay, so that the monitor can immediately get back to the time-critical business of snooping its connection.

An obvious way to accomplish this would be for the monitor's output channel to have unbounded FIFO depth. This would work well, but what happens if someone mistakenly connects a 1-place FIFO to the monitor's output? The monitor would no longer work reliably. Instead, it is better for the monitor to write observed data into its output channel using the channel's `sneak` method. This mechanism allows the monitor to use *any* channel as if it had unbounded depth; the `sneak` operation takes zero time and is guaranteed to succeed, even if it "overfills" the channel.

Copyright © 2009 by Doulos. All rights reserved.

Indeed, then, it is feasible to use VMM channels for observation. However, you have probably already sensed that it may be a little clumsy by comparison with the very flexible OVM analysis mechanism. Adding multiple consumers for the observed data is likely to disrupt the top-level test environment significantly, adding a `vmm_broadcast` component and several channels. It would be good to find an approach that more readily supports multiple subscribers.

Exposing observed data using the VMM notification service

A key feature of VMM's infrastructure is the *notification service*. This is a global, testbench-wide mechanism that allows a component to signal its status to any interested observer anywhere in the testbench. Every transactor is expected to have an instance, named `notify`, of the `vmm_notify` class. You don't need to create this instance, because it is a property of the `vmm_xactor` base class. However, it is necessary for your transactor to register all the notifications it wishes to use. There are three distinct kinds of notification, all managed by the same underlying service:

- `ONE_SHOT` notifications, which work just like regular Verilog events;
- `BLAST` notifications, which work around some possible race conditions by providing functionality equivalent to the behavior of SystemVerilog's `wait(`*some_event*`.triggered);`
- `ON_OFF` notifications, which provide a Boolean flag (and the means to wait for changes on it).

For example, you might choose to create a notification to indicate that your BFM is currently in a wait state. That would be an `ON_OFF` notification because it's a simple status flag. Alternatively, you could create a notification to mark when a cache line refill has just completed; that would be a `ONE_SHOT` notification because it's an event that fires at a specific moment in time.

To maximize flexibility, user-created notifications have integer identifiers so that it is very easy to create a new notification whenever you need one – but, in practice, a transactor's notifications will all be created at the time the transactor is constructed. All the notifications registered by a given transactor must have a unique integer identifier within that transactor, but fortunately it is easy to arrange this when creating the notification. Here's one way to do it:

Copyright © 2009 by Doulos. All rights reserved.

```
class my_monitor_BFM extends vmm_xactor;
  ...
  // Variables used to label my new notifications
  int IN_WAIT_STATE;
  int CACHE_REFILL_DONE;
  ...
  // Notifications will be created in the constructor
  function new(...);
    super.new(...);
    ...
    // Create the new notifications
    IN_WAIT_STATE = notify.configure (
                     // ask VMM to give unique ID
                     .notification_ID(-1),
                     // this is a flag notification
                     .sync(ON_OFF) );
    CACHE_REFILL_DONE = notify.configure (
                     .notification_ID(-1),
                     // this is an event
                     .sync(ONE_SHOT) );
```

Having created an integer variable to hold each notification ID in this way, we can in the future simply use that variable's name as if it were the name of the notification. For example, on discovering that a cache refill had completed, some method of the BFM might trigger the notification thus:

```
this.notify.indicate( CACHE_REFILL_DONE );
```

Carrying data along with the notification

We have seen how to create and fire a notification, but how can we use this to help with the observation problem? The answer lies in the notification service's ability to tag a notification with auxiliary data. You will not be surprised to learn that this auxiliary data is a `vmm_data` reference, so it can carry any transaction data object. It is simply passed as a second argument to the `indicate` method:

```
this.notify.indicate( CACHE_REFILL_DONE, observed_data );
```


Copyright © 2009 by Doulos. All rights reserved.

Any code that waits for the notification, using the corresponding `wait_for` method, can also retrieve the auxiliary data by immediately calling the notification's `status` method. Clearly this offers a straightforward nonblocking data broadcast mechanism, well suited to exposing observation data from a monitor.

Subscribing to a notification

So far we have only discussed firing (*indicating*) a notification. Naturally, that's done by the same class that owns the notify object, so it's just a matter of calling `this.notify.indicate(…)`. A subscriber object, however, is outside the observer that's doing the notifying. How does such a subscriber detect the notification? It must call the `wait_for` and `status` methods of the *observer's* `notify` object, taking care to use the correct notification ID. But this should set your mental alarm bells ringing: if one component needs to know about another one, how can it be re-usable?

The key is to provide your subscriber with a reference to the monitor. This reference will be populated by the environment class, which of course knows about both the subscriber and the monitor. The subscriber remains re-usable.

```
class my_subscriber extends vmm_xactor;
  ...
  my_monitor_BFM cache_mon;
  ...
```

Now, when the subscriber wishes to collect observed data from the monitor, it can reach through that reference:

```
cache_mon.notify.wait_for(cache_mon.CACHE_REFILL_DONE);
```

Ah, but what happened to the auxiliary data? Now that we have detected the notification, we can collect that data using the notifier's `status` method:

```
$cast(cache_data,
cache_mon.notify.status(cache_mon.CACHE_REFILL_DONE));
```

Notice that we needed to use `$cast` because the `status` method returns a result of `vmm_data` base type, but our `cache_data` item is presumably of some derived type.

 Copyright © 2009 by Doulos. All rights reserved.

Using Callbacks

VMM offers a third mechanism for making data accessible: *callbacks*. It is probably the most attractive of all, because it offers a clean solution to the problem of how to decouple the monitor and subscriber classes so that both are re-usable, and it easily supports zero, one or more subscribers.

Transactors should have callback points

Any well-written VMM transactor, including our monitor, should have various *callback points* in its main thread of execution. In essence, whenever the transactor does anything that might be interesting to other parts of the testbench it should do a callback. You can think of the callback as a call to an empty function, giving you the chance to fill in the code for that function so that it does whatever you want it to do - *without* affecting the original transactor.

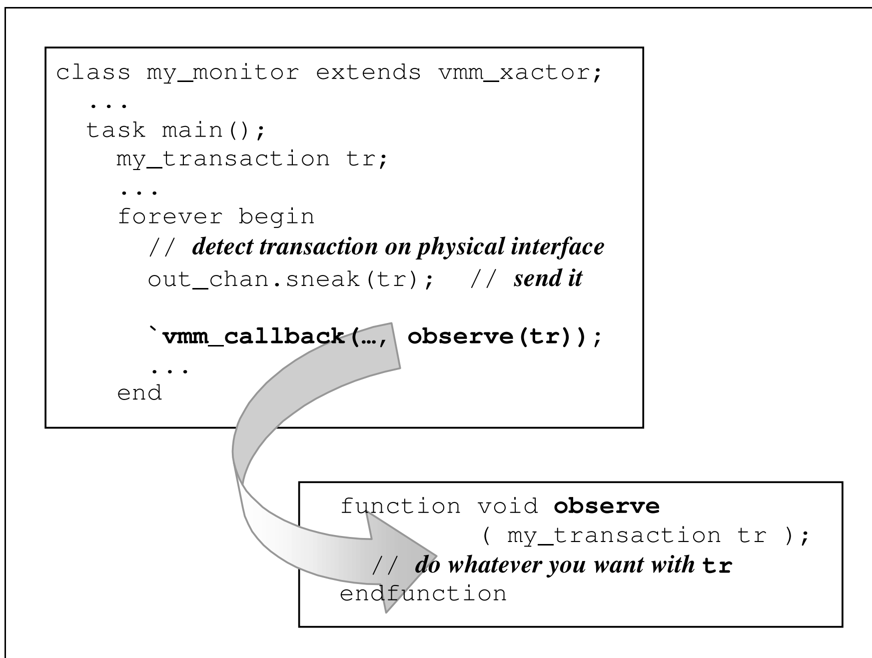

Figure 5: Callbacks allow for easy addition of user code

Copyright © 2009 by Doulos. All rights reserved.

Callback façade class

In reality it is not sufficient for the callback to call an empty function. It is better to call a method of an object, because it is then easy to extend that method by using a derived-class object in place of the base-class object. But where should that method be? VMM expects you to write a *callback façade* class to encapsulate the set of methods that will be called by a given transactor's callbacks. Putting the callback methods into a class brings important benefits. Making the methods virtual (and, indeed, empty) allows you to write the callback façade class at the same time as you write the transactor, which then simply needs to know the names and argument lists of those methods. You can then create derived versions of the callback façade class, to provide application-specific implementations of any or all of those methods, without disturbing the original transactor in any way. Figure 6 shows a skeleton of the code you would need to write to do this.

 Copyright © 2009 by Doulos. All rights reserved.

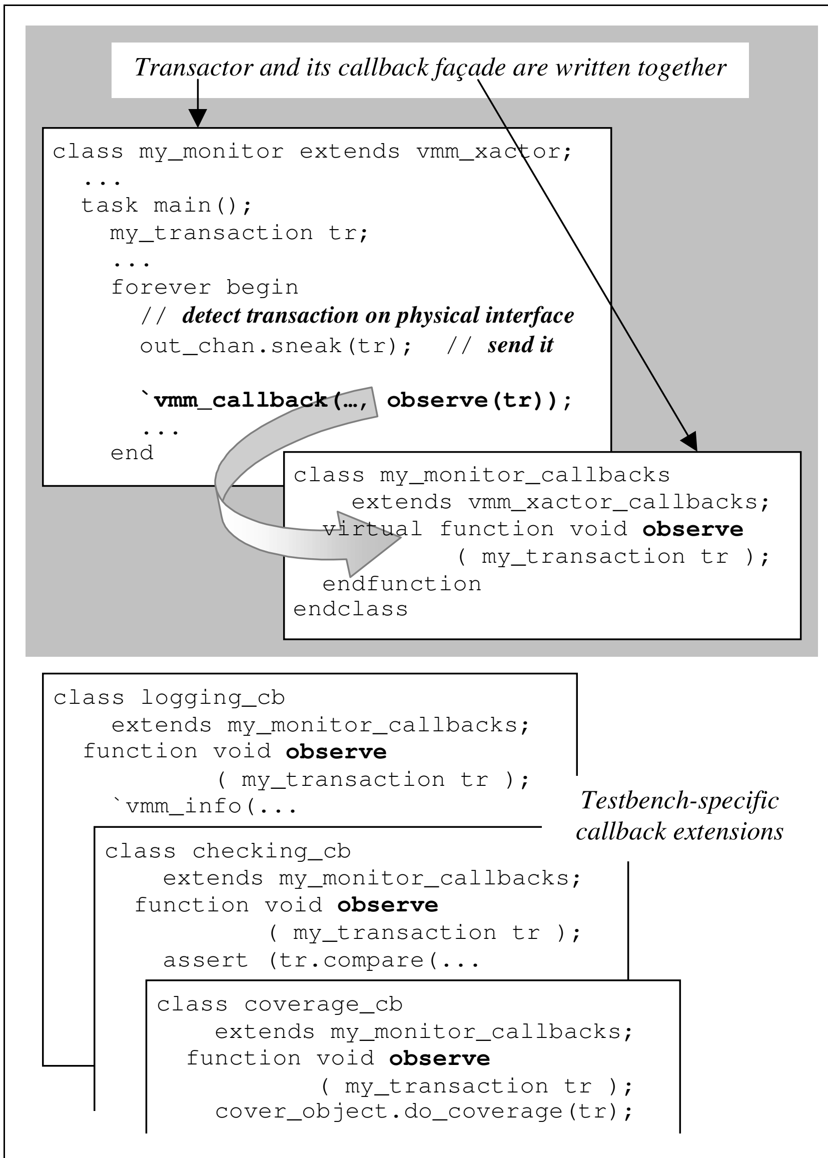

Figure 6: Writing and extending a callback façade

Copyright © 2009 by Doulos. All rights reserved.

Registering your callbacks with the transactor

We have now seen how to create customized extensions of a transactor's callback façade. Now we must arrange to attach the appropriate extensions to a specific transactor instance. The transactor's enclosing environment class should take responsibility for doing this as part of its `build` method.

First, we must create instances of our extended callback classes alongside the transactor instance. Then we *register* those instances with the transactor, adding them to the transactor's list of callback objects so that their methods will be called automatically whenever the transactor invokes the `` `vmm_callback `` macro.

```
class my_environment extends vmm_env;
  ...
  logging_cb log_cb;
  checker_cb check_cb;
  my_monitor monitor;
  ...
  function void build();
    ...
    monitor = new(…);   // build the transactor
    check_cb = new;     // build callback objects
    log_cb = new;
    ...
    // register your custom callback objects with the transactor
    monitor.append_callback(log_cb);
    monitor.append_callback(check_cb);
    ...
```

The callback objects are now acting as subscribers to the monitor's callbacks. Functions in each callback object are automatically called whenever the monitor has interesting information to share. Any number of subscribers (including zero) can be registered with the monitor, in much the same way that any number of OVM subscribers can be connected to a monitor's analysis port.

Making the callbacks fully VMM compliant

To keep our description and examples simple, we have ignored a few details that are necessary to make your callbacks fully VMM compliant. Most

 Copyright © 2009 by Doulos. All rights reserved.

importantly, whenever a transactor calls one of its callbacks it should provide a reference to itself (`this`). That provides two important pieces of functionality:

- A single callback object can be registered with more than one transactor. The callback function can use the transactor reference to determine which transactor was responsible for a given callback invocation.
- The callback method now has the means to look back into the calling transactor to find other information that was not necessarily supplied as part of the callback method's argument list.

VMM imposes a number of other rules and recommendations. In particular:

- Callbacks that must be nonblocking should be coded as void functions, not as tasks. This is by far the most common case; it is very unusual for a callback to interfere with the timing of a transactor. For monitor transactors, *all* callback methods should be functions so that they cannot consume time.
- Callbacks can have transaction arguments that are qualified `const`. This prohibits the callback method from modifying the contents of the transaction. For stimulus-generating transactors it may be useful to allow callbacks to modify transaction data, perhaps to perform error injection or payload modification. Monitor transactors, on the other hand, simply report what they see and it is usually inappropriate for a callback to modify this information. Consequently, the callback façade for a monitor transactor should have methods looking like this:

```
virtual function void observe (
  const my_monitor caller,
  const my_transaction tr );
```


Copyright © 2009 by Doulos. All rights reserved.

In summary...

It is important to be able to add new observation components to a testbench without disrupting the existing structure of components and connections. Both OVM and VMM provide mechanisms that make this possible.

OVM

In an OVM environment, observed transaction data should always be exposed through an analysis port so that any number of subscribers may connect themselves to the port and see that observed data. New subscribers can be added without affecting the existing testbench structure; it is merely necessary to instantiate the subscriber, and connect the appropriate monitor's analysis port to the new subscriber.

VMM

In a VMM environment there are at least three possible methods that a monitor can use to publish observed transaction data:

- sneaking the data on to an output channel;
- raising a notification, with the transaction provided as the notification's status data;
- invoking a callback.

Of these three methods, Doulos recommends callbacks as being the most elegant, scaleable and flexible. New callback subscribers can be added without affecting the structure of an existing testbench, and subscribers and monitors can be almost completely decoupled because the callback façade class defines the interface between them.

Copyright © 2009 by Doulos. All rights reserved.