Doulos Verification TechNote 2: VMM Productivity Enhancements

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`.

Verification TechNote 2

VMM Productivity Enhancements

This *Doulos Verification TechNote* highlights the productivity macros that have been added to recent releases of the VMM Standard Library. These new features are completely optional, and therefore have no compatibility impact on your existing VMM code. For any new code, though, they offer a dramatic reduction in programming effort, removing the need to write error-prone and time-consuming boiler-plate code in every new class you create.

Readers of this TechNote should already have some familiarity with the VMM Standard Library, perhaps from constructing some verification components or from customizing an existing testbench.

As always, your primary resource for VMM is the community site:

`www.vmmcentral.org`

Additional SystemVerilog verification resources can be found on the Doulos site:

`www.doulos.com/knowhow/sysverilog`

The need for automation

As soon as you begin to write your own VMM code, you recognize how much of that code is necessarily repetitive. For example, suppose you wish to create a new transaction data class. Of course you base it on the `vmm_data` class, with your own extensions.

Very soon, though, you will find that your extensions – the part of the new class that does your real work – accounts for only a small part of the code you need to write. Much of the derived class code is apparently unproductive bureaucracy although, of course, you know that you *must* create it so that the VMM infrastructure can use your new data class correctly.

An especially frustrating example is the `allocate` method, which is a mandatory part of any `vmm_data` extension. Here's what it must look like:

```
class my_extended_data extends vmm_data;
  ...
  virtual function vmm_data allocate();
    my_extended_data it = new();
    return it;
  endfunction
  ...
```

Although this is so straightforward, you nevertheless have no choice but to write it yourself. The required code pattern is *exactly* the same for any derived `vmm_data` class, but you cannot rely on a base-class implementation because the data type of the variable `it` must match the derived class you are creating. Similar needs arise with the constructor `new` and with several other standard VMM methods such as `copy`, but the `allocate` method is especially frustrating because it contains *absolutely no code* that relates to your own custom extensions.

You probably know already that the VMM software distribution comes to the rescue with its `vmmgen` script. You can use `vmmgen` to create the skeleton of your derived class, with much of the boiler-plate code already written for you, leaving you to complete the interesting part – the custom extensions that make your derived class behave in the way you want. However, many people feel a

little uncomfortable using code-generating scripts such as `vmmgen`. They may generate superfluous code that you must remove for your specific requirements, adding to the risk of accidentally introducing errors. Furthermore, since `vmmgen` knows nothing about your custom data extensions, it cannot fully automate the creation of some data-dependent methods such as `compare` and `copy`. A more complete solution to this problem would bring real benefits.

Automation using macros

The VMM Standard Library makes heavy use of Verilog `` `define `` macros to automate the construction of some user code. Take, for example, the `vmm_channel` macro. This macro, usually invoked at the end of the file that creates a user-specified extension to `vmm_data`, automatically writes the code for a new channel class that is customized to carry transactions of the new user-defined data class.

Two even larger macros, `` `vmm_scenario_gen `` and `` `vmm_atomic_gen ``, are generally used in the same way as `` `vmm_channel ``; they automatically create class definitions for an atomic random generator and a scenario generator, again both customized for the newly defined data class. Other, slightly less spectacular, macros are used to simplify a user's life when writing message output code, callback invocations and so on.

Macros, macros...

VMM users are familiar with macros. Everyone agrees that macros are a necessary evil: they bring huge benefits in convenience and portability, but they can easily lead to code that is hard to understand and even harder to debug. In the VMM Standard Library, however, the macros are robust and their purpose is clear, so users are not afraid to exploit them.

Given this background, it will come as no surprise to find that the new facilities for automating boiler-plate code in derived classes make heavy use of macros.

Automation of `vmm_data` extensions

Any attempt to automate some of the work of building custom `vmm_data` derived classes must deal with at least two problems:

- Some standard patterns of code, such as the `allocate` method, must be created and adjusted to match the newly written class. This requirement is fairly straightforward, because the new class name can be provided as an argument to a suitable code generation macro.
- Some methods, such as the `copy` method, must manipulate individual fields (data members) of the new class in various ways. This is more difficult, because it requires a macro to generate configurable patterns of code for each field in turn.

At the 2008 Synopsys User Group (SNUG) meeting in Munich, Germany, a group of expert VMM users presented a paper describing how they had met some of these automation challenges:

Kevin Hyland and Vishal Patel:
Super vmm_data! Automating vmm_data methods in data structures.
SNUG Europe 2008. Available at `www.snug-universal.org`

However, in roughly the same time-frame (late 2008) the VMM Standard Library was enhanced with a new set of macros designed to handle exactly these issues, not only for `vmm_data` but also for many of the other important VMM base classes.

Getting started with the automation macros

Let's begin with a very simple example of a user—defined transaction data class based on `vmm_data`. Suppose our data class needs three data members:

```
rand logic [7:0] data;
rand bit enable;
rand enum { READY, WAITING, DONE } mode;
```

Here is a *complete* implementation of our new data class:

```
class sample_data extends vmm_data;

  rand logic [7:0] data;
  rand bit enable;
  rand enum { READY, WAITING, DONE } mode;

  // add your randomization constraints here

  `vmm_data_member_begin(sample_data)
    `vmm_data_member_scalar(data, DO_ALL)
    `vmm_data_member_scalar(enable, DO_ALL)
    `vmm_data_member_enum(mode, DO_PRINT+DO_COPY)
  `vmm_data_member_end(sample_data)
  `vmm_data_byte_size(2, 2)

endclass
```

The new automation macros are shown in bold. As you can see, the amount of work required to create a new `vmm_data` class has been dramatically cut. In simple cases it is sufficient to add your own data members and constraints, and then invoke the appropriate automation macros, as we have done here.

What are the macros doing?

Working together, the group of macro invocations in our example creates:

- a static instance of `vmm_log`, as required for all `vmm_data` classes;
- standard implementations of `new`, `allocate` and `is_valid` methods;
- customized implementations of the remaining key methods `psdisplay`, `copy`, `compare`, `byte_pack` and `byte_unpack`.

Some of this work is completely automatic and is based only on the new class's name. However, many of the methods may have subtly different behaviors on different user-defined data members. In our example, the enumerated field `mode` should be displayed and copied along with the other fields. but it should not participate in compare and pack operations because it is not part of the core data that will be transferred to or from a DUT. To cope with this requirement, each `vmm_data_member...` macro (except for `begin` and `end`) has a second argument that specifies how this particular field should be

handled. The simplest approach is to specify `DO_ALL` for this second argument, indicating that the field should participate in all the methods. For our `mode` field, however, we specified `DO_PRINT+DO_COPY` indicating that this field should participate only in `psdisplay` and `copy` methods, and should be skipped (ignored) by the other automatically-created methods. As you might imagine, the predefined constants `DO_PRINT`, `DO_COPY` and so forth are bit-masks that can be added or OR'd together. Alternatively, you can specify that a field should participate in all operations except those you specify, by subtracting bit-masks: `DO_ALL-DO_PRINT` would specify that a field should participate in all methods except `psdisplay`.

There is a summary of the available bit-mask constants in the *VMM User Guide* under the heading `vmm_data::do_what_e` (the name of the enumeration type that specifies the various constants).

Types of field

The handling of each data member depends somewhat on its data type. In particular, arrays, enumerations and strings require special arrangements. Consequently you must be careful to choose the correct automation macro for each of your fields.

The table on the next page summarizes the different `vmm_data_member` macros that are available to specify each field. Don't forget, though, that the list of such macros *must* begin with an invocation of `vmm_data_member_begin` and finish with `vmm_data_member_end`. As always when using complex macros, if you get this wrong you will encounter some very bizarre and confusing compiler error messages!

Note, too, that *you must not* add any code of your own between the various macro lines. These macros do not stand alone; each of the field macros creates just a small part of the code of a large and complicated function, and it is very important that you should not interfere with this process.

Summary of available `vmm_data_member` macros

vmm_data_member_...	used for:
...scalar	Any integer-like or vector field (either 2-state or 4-state) such as `bit`, `logic`, `integer`, `reg [7:0]` etc, up to a maximum of 4096 bits
...scalar_array	Any fixed-size unpacked array of scalar
...scalar_da	Dynamic array of any scalar
...scalar_aa_scalar	Associative array of any scalar, with a scalar index type
...scalar_aa_string	Associative array of any scalar, with a string index type
...string ...string_array ...string_da ...string_aa_scalar ...string_aa_string	Similarly for variables and arrays of string type
...enum ...enum_array ...enum_da ...enum_aa_scalar ...enum_aa_string	Similarly for variables and arrays of any enumeration type
...vmm_data ...vmm_data_array ...vmm_data_da ...vmm_data_aa_scalar ...vmm_data_aa_string	Similarly for variables and arrays of any class type based on `vmm_data` – useful when you have data objects that contain instances of other such objects. See further details on the next page.
...handle ...handle_array ...handle_da ...handle_aa_scalar ...handle_aa_string	Similarly for variables and arrays of any class type that is not based on `vmm_data` – for example, references to other parts of the testbench

Extending the macros for user-defined behavior

In addition to the field macros listed in the table, you can specify `vmm_data_member_user_defined` to get fully customized control over the handling of a field. To do this requires thorough understanding of the mechanisms used by these macros; if you wish to use it, we suggest you read the section **Shorthand Macros** in the VMM User Guide that you can find in the `doc/` subdirectory of the VMM distribution.

Adding the `vmm_data_byte_size` macro

To complete the automation, you should add an invocation of `vmm_data_byte_size` immediately after the `vmm_data_member_end` invocation. This provides default implementations of the required `byte_size` and `max_byte_size` methods. The two arguments to this macro are SystemVerilog expressions, evaluated in the context of the appropriate methods. The first one computes the maximum possible size of the packed data, and is usually a constant; the second expression computes the actual size of the current object, and could be an arbitrary arithmetic expression. In our simple example the structure's size is fixed and so the size and maximum size are identical.

Applying macros to members of class type

If your data class has any data members that are themselves of class type, you must take care to decide how those embedded objects should be handled. If the data contents of the embedded object form part of the real data of your transaction, then any operation on your object should also descend into the contents of the embedded object. On the other hand, a class-type data member may be merely a reference to some other object (for example, a transaction keeping a reference to the transactor that created it). In such a case the contents of that other object do not form part of your object's data, and it should be skipped for copy, comparison and other methods.

Members that reference an object whose class is derived from `vmm_data`

Data members whose contents form part of your object's data will certainly be of a class type that is derived from `vmm_data`. They should be handled by the

vmm_data_member_vmm_data... macros, all of which take three arguments rather than the normal two. The third argument is another bit-mask that specifies how the copy and compare methods should operate on this field.

There is a summary of the available bit-mask constants in the *VMM User Guide* under the heading vmm_data::do_how_e (the name of the enumeration type that specifies the various constants).

Members that reference an object of some other class

Member variables that are references to other objects, not derived from vmm_data, should be included using the vmm_data_member_handle... macros. Of course, VMM knows nothing about such objects and cannot do anything with their internal structure. Consequently, the various automatically created methods are fairly simple and, in particular, copy and compare are invariably shallow – it is only the reference that is copied or compared.

Automation of other VMM classes

In much the same way as for vmm_data, automation macros exist for other key VMM classes. Most users will find that it is the vmm_data macros that yield the greatest immediate benefit, so we do not describe the others in detail here; instead we refer you to the VMM User Guide and, indeed, to the source code itself.

Automation macros exist for vmm_xactor, vmm_env, and vmm_subenv as well as for vmm_data.