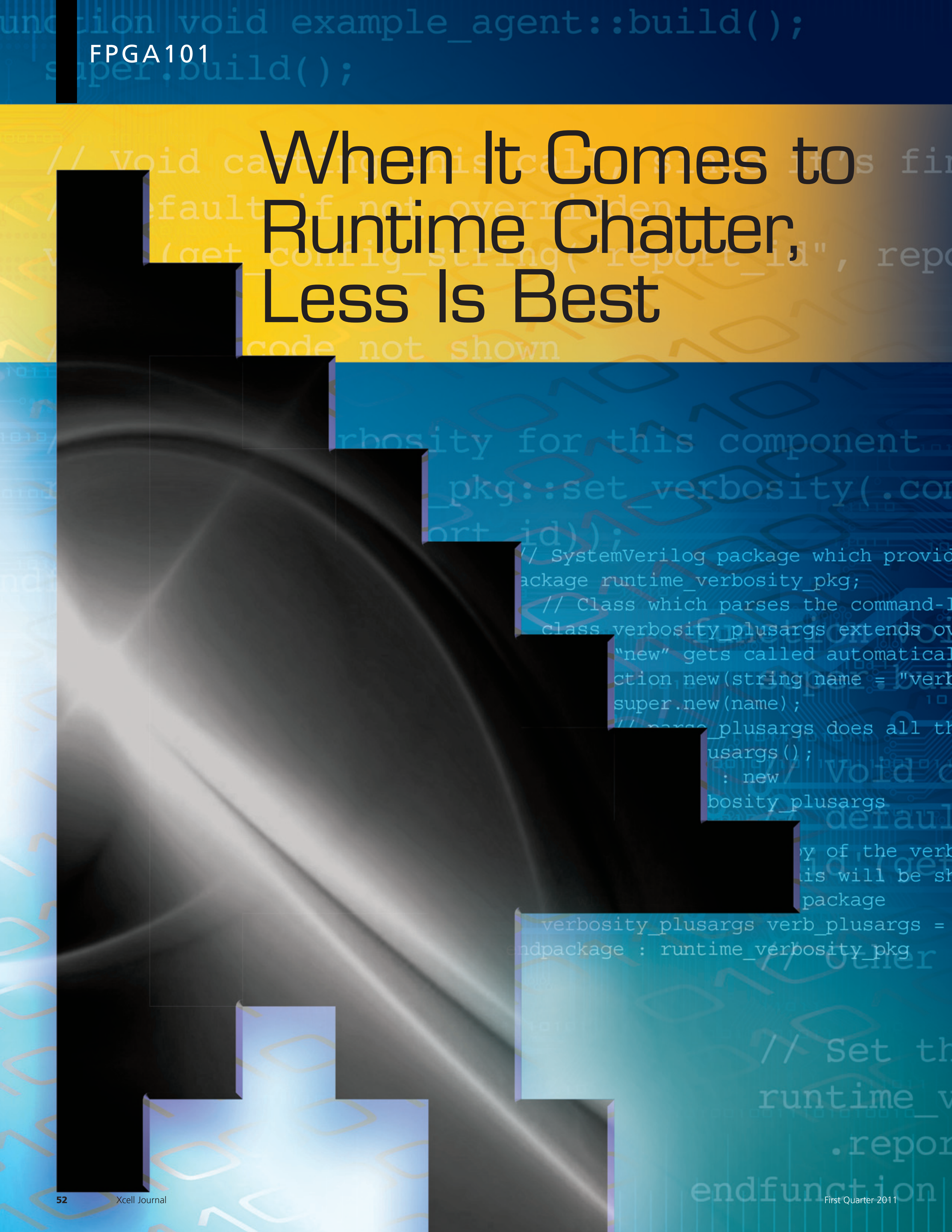
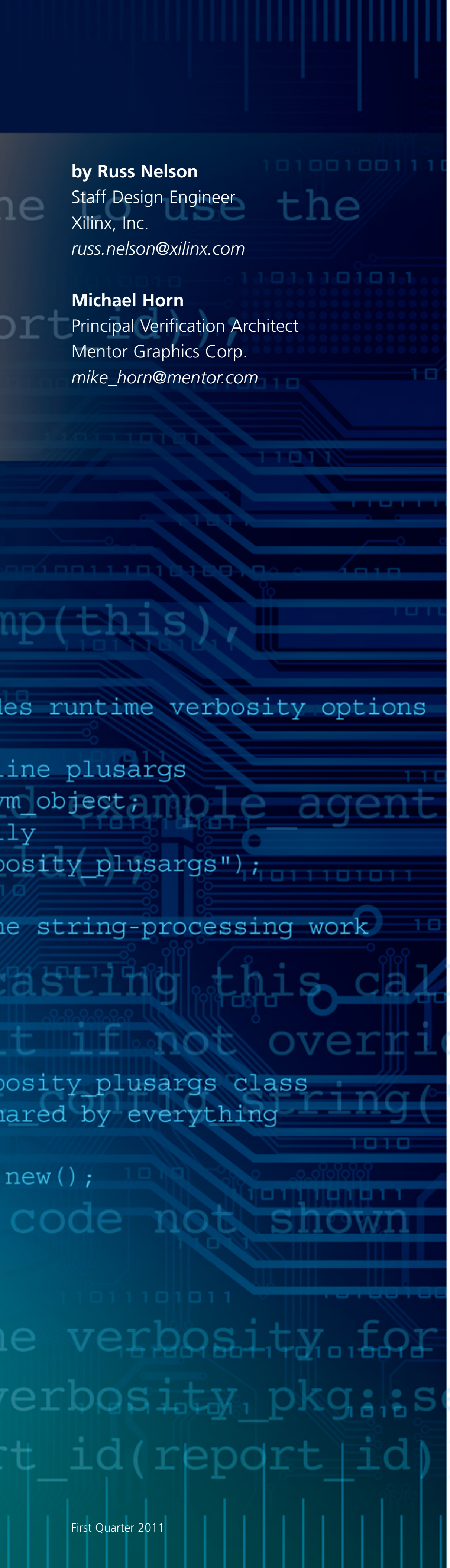
When It Comes to Runtime Chatter, Less Is Best





by Russ Nelson
Staff Design Engineer
Xilinx, Inc.
russ.nelson@xilinx.com

Michael Horn
Principal Verification Architect
Mentor Graphics Corp.
mike_horn@mentor.com

Xilinx add-on package simplifies testbench debug by controlling verbosity on a component level.

The end goal of a testbench is to show that your design achieves its purpose without having any bugs. Getting to that point, however, takes work. You will have to find, diagnose and fix bugs, in both the testbench and the design. Anything that makes the debugging process easier and achieves closure on the testbench and design under test (DUT) is valuable—both in terms of time and money.

Controlling verbosity on a component level can save considerable time and effort when debugging failing tests by presenting only information relevant to the area where a failure is occurring. Controlling verbosity at runtime saves considerably more time by allowing engineers to focus on a problem area without having to edit and recompile the testbench code or sift through excessively verbose log files. While there is always the option of using “grep” or other command-line tools to post-process logs, having the right information in the log files to begin with saves effort.

Some testbench methodologies do a great job of controlling verbosity in a granular way, but they lack flexibility at runtime. This is the case with the Open Verification Methodology (OVM), the de facto industry testbench methodology standard.

The OVM provides a great framework for reporting debugging information. You can specify severity and verbosity levels for each reported message and implement different actions for various combinations of the two, specifying different verbosity thresholds and actions down to the ID. Moreover, during debug, OVM allows you to increase or decrease the verbosity threshold for the testbench as a whole. What OVM doesn’t do is provide the ability to change verbosity for individual components at runtime.

To solve this runtime issue, we developed an add-on package at Xilinx, with additional direction from Mentor Graphics. When we employed it in a recent testbench environment, it saved us considerable time. Because of the advantages it delivers, we have adopted this package for several current designs at Xilinx. In fact, we almost always use runtime verbosity controls to zero in on problem areas. We have found that not having to recompile or sift through excessively verbose log files makes the designer’s life a lot easier.

In this article, we will show the basic steps required to implement these verbosity controls. You can download the full methodology (including the standalone package, which you can easily add to existing testbenches) from *http://www.xilinx.com/publications/xcellonline/downloads/runtime-verbosity-package.zip*. Everything said here about OVM should apply equally to the Universal Verification Methodology, the next evolutionary step for testbench methodology. The initial UVM version is derived from the OVM with a few additions.

A FEW CHOICE WORDS MAKE THE POINT

Selectively increasing verbosity for a component or set of components is an easy and effective way of accelerating the time it takes to find bugs in a particular area of a verification testbench. Passing in directives via plusargs turns up the verbosity for components in the area of interest.

Take, for example, the following:

```
<sim_command> +OVM_VERBOSITY=OVM_LOW
+VERB_OVM_HIGH=TX_SCOREBOARD
```

In this command, OVM_VERBOSITY=OVM_LOW is the built-in OVM method for reducing the overall verbosity to a minimum, while VERB_OVM_HIGH=TX_SCOREBOARD is our command for boosting the TX scoreboard to a high level of verbosity.

Before adding any code to a testbench, it is important to define the methodology that you will use to identify each component. It is also helpful (but not necessary) to define what kind of information will be presented at each verbosity level—consistency makes the debugging process easier. (See the table on the final page for an example of how to plan the verbosity levels.) Once you have formulated a plan, there are two key testbench features required for enabling runtime verbosity controls. First, you must parse the command-line directives. Second, you must carry out the directives to change verbosity on a component-level basis.

The first step is to create a procedure for distinguishing one component from another; in other words, how you will identify the areas of interest. We recommend using a unique tag (we call it report_id) for each component, with a default value that you can override via the OVM configuration database. The following code snippet illustrates one way of implementing this functionality:

```
class example_agent extends ovm_agent;
   string report_id = "EXAMPLE_AGENT";
   // Other code not shown
endclass
```

```
function void example_agent::build();
   super.build();

   // Void casting this call, since
   // it's fine to use the
   // default if not overridden
   void'(get_config_string("report_id",
                           report_id));

   // Other code not shown
endfunction
```

If you will use a particular component in several locations within a testbench, it can be helpful to override its default tag (report_id) with a new tag in such a way that each component's tag is unique within the testbench. When using the tag to control verbosity, the level of control is only as precise as the tag—if two components share a tag, they cannot be controlled independently.

Another method for identifying a component is by its hierarchical name. The hierarchical name is an OVM built-in property of each component in a testbench. While there is nothing wrong with using the hierarchical name to identify a component, it takes a lot more typing to specify components of interest, as the names can be quite long. However, OVM requires that the names be unique, which can be an advantage. The example code provided with this article supports identifying components either by tag or by hierarchical name.

The next step is to determine the runtime changes to verbosity. The OVM takes care of default verbosity via the OVM_VERBOSITY plusarg (OVM_VERBOSITY=OVM_LOW, etc.). We recommend implementing a similar method, with a plusarg for each OVM-defined verbosity level.

We used VERB_OVM_NONE, VERB_OVM_LOW, VERB_OVM_MED, VERB_OVM_HIGH, VERB_OVM_FULL and VERB_OVM_DEBUG. These correspond to the verbosity enum built into OVM using similar names.

You can specify each verbosity level using a single report tag, a hierarchical name or a colon-separated list of tags or names. Additionally, for each tag, you can specify that the verbosity setting should be applied recursively to all the

Before adding any code to a testbench, it is important to define the methodology that you will use to identify each component. It is also helpful (but not necessary) to define what kind of information will be presented at each verbosity level—consistency makes the debugging process easier.

specified component's children. The included code uses the prefixes "^" (for a recursive setting) and "~" (for a hierarchical name). Here are a few examples:

- Low verbosity for most of the testbench, high verbosity for the scoreboard:
  ```
  <sim> +OVM_VERBOSITY=OVM_LOW
  +VERB_OVM_HIGH=SCOREBOARD
  ```

- Full verbosity for the slave and all its children:
  ```
  <sim> +VERB_OVM_FULL=^SLAVE
  ```

- Full verbosity for the TX agent and its children, except for the startup module:
  ```
  <sim> +VERB_OVM_FULL=^~ovm_test_top.tb.
  tx.tx_agent   +VERB_OVM_LOW=~ovm_test_
  top.tb.tx.tx_agent.startup
  ```

You must determine the passed-in arguments early in the simulation so that components can pick up the changes before they start reporting. The example code accomplishes this by using a singleton class instantiated in a package. By including the package somewhere in the testbench (we suggest the top-level module), the command-line processing gets called automatically at the beginning of the simulation. The following shows a simplified version of the package code, illustrating just the key concepts.

```
// SystemVerilog package which provides
// runtime verbosity options
package runtime_verbosity_pkg;
  // Class which parses the
  // command-line plusargs
  class verbosity_plusargs extends
                                ovm_object;
    // "new" gets called automatically
    function new(string name =
                    "verbosity_plusargs");
      super.new(name);
      // parse_plusargs does all the
      // string-processing work
      parse_plusargs();
    endfunction : new
  endclass : verbosity_plusargs

  // Instantiate one copy of the
  // verbosity_plusargs class in the
  // package - this will be shared by
  // everything which imports
  // this package
  verbosity_plusargs
                 verb_plusargs = new();
endpackage : runtime_verbosity_pkg
```

Finally, you must change the verbosity setting for each reporting item. It is important to apply the setting early in the item's life span, before reporting begins. OVM components operate in "phases," namely build, connect, end_of_elaboration, start_of_simulation, run, extract, check and report. UVM has an expanded phasing mechanism, but the need for applying the verbosity setting early still applies.

To ensure that you have tweaked the verbosity before reporting begins, you should make the adjustment in the build() task of the component. UVM adds the capability to customize the phases somewhat, but it still uses build() as the first phase. The example code provides a simple call to update the component's verbosity setting. Building on the example agent shown above, the build() function now looks like this:

```
function void example_agent::build();
   super.build();

   // Void casting this call, since
   // it's fine to use the default
   // if not overridden
   void'(get_config_string("report_id",
                            report_id));

   // Other code not shown

   // Set the verbosity for this component
   runtime_verbosity_pkg::set_verbosity(
      .comp(this),.report_id(report_id));
endfunction
```

For simplicity, we recommend adding this functionality to the build() function of each component in the testbench. However, if that is not feasible (for example, if it's too late to change the components or if you are using legacy or third-party verification components), you can override the default reporter, which is called by each component. The downside of overriding the reporter is that it won't increase the verbosity when using the OVM convenience macro `ovm_info, since the macro checks the component's verbosity level before calling the reporter (it still works for decreasing the verbosity). Overriding the default reporter in the example code is as simple as adding the following line somewhere in the testbench (we recommend adding it to the top-level module):

```
import runtime_verbosity_server_pkg::*;
```

You can combine the two methods if you need to support legacy or third-party components yet still want to use the macros in the code.

The methodology shown here also works with UVM, with small changes to a few function names. With a little more effort, you can also extend it to work with objects, such as transactions and sequences. We've included a small example testbench showing the method presented in this article in the methodology zip file at *http://www.xilinx.com/publications/xcellonline/downloads/verb-ovm-environment.zip*.

As the examples presented here show, with a little thought and planning, enabling runtime verbosity in a testbench is a very simple matter. This approach has proven itself in production testbenches and designs. Our experience shows that the effort required for this small addition has a disproportionately large payoff—debugging tests become much quicker and easier to do. This means the simulator is running simulations more often and simulation cycles are not wasted. This is crucial, since simulation cycles are not a renewable resource. If a simulator is not running at a particular moment, that time could be viewed as a wasted resource. Quicker debugging means more up-time for testbench development and shorter time-to-closure.

The end goal of writing a testbench is to get a design implemented and in the customer's hands. Any advantage that accelerates this goal is a welcome addition to the designer's bag of tricks. Adding finer-grained verbosity control at runtime, instead of requiring a loopback to compilation, saves valuable time and energy, speeding delivery of the end product and startup of your next project.

WHICH INFORMATION AT WHICH LEVEL?

It can be helpful to define what type of information is presented at each verbosity level, especially if a large design team is involved. Establish a list of all the types of information you expect to generate and create a table showing the verbosity for each message type. Our example table shows a few common types of messages.

VERBOSITY	MESSAGE TYPE	EXAMPLES
OVM_LOW	Major environment/DUT state changes	Beginning reset sequence; link established
OVM_MEDIUM	Other environment/DUT state changes	Interrupt asserted; arbitration timeout
OVM_HIGH	Sequences started and finished	Beginning read sequence; ending reset
OVM_HIGH	Scoreboard activity	Compare passed; check skipped
OVM_HIGH	Major driver and monitor events	Packet corrupted due to reset
OVM_HIGH	Reporting high-level configuration settings at startup	32-bit datapath; two masters and four slaves instantiated
OVM_FULL	Transactions driven and monitored	Full dump of transaction contents
OVM_FULL	Driver and monitor state changes	Arbitration request/grant; inserting <n> IDLE cycles
OVM_DEBUG	Objections raised and lowered	Raised objection; lowered objection
OVM_DEBUG	Detailed driver, monitor and scoreboard activity	Loop iterations, if-branches followed, etc.