User Programmable Targeted UVM Debug Verbosity Escalation

Sam Mellor
sam.mellor@arm.com
Arm, City Gate, 8 St Mary's Gate, Sheffield, S1 4LW, United Kingdom

***Abstract* - Universal Verification Methodology (UVM) [1] supports various verbosity levels (i.e. UVM_FULL, UVM_NONE) for outputting debug to a transcript or log as configured by the report action. This can be set independently for IDs on a component level.**

Debug output to a transcript or log can consume a large amount of simulation time and generate large log files which can become difficult to parse. Large log files also add additional storage costs. These issues worsen as the verbosity level is increased, leading to more output. Verbosity is often increased to track specific transactions through the Bus Functional Model (BFM) however this causes debug output for all transactions to be generated.

We demonstrate a method that borrows from the tried and tested UVM factory method to easily and dynamically escalate the verbosity of transactions and streams. This allows the verification engineer to focus on only relevant debug, accelerating the time to reach a failure point compared to just simply increasing the test verbosity an d enabling a quicker turnaround time on fixes. The method is invoked at run-time, often avoiding the need to recompile.

I. Introduction

The aim of targeted verbosity escalation is to work with the existing UVM debug implementation, not replace it. The existing UVM debug output permits debug messages to have a particular ID, and the verbosity can be changed per ID and per component. This however does not permit a debug message to be output depending on the contents of the transaction (i.e. display debug for transactions targeting a particular address and not others).

There are already built in mechanisms within UVM to permit the user to filter messages. The report catcher for instance allows for multiple callbacks to be registered and the messages inspected and have the verbosity modified to filter out the debug message. This can be a powerful way to cut down the log output and on the face of it can achieve the same functionality. There are however potential problems with doing this. The filtering is done on the final output message, which therefore requires the message to be created first, and then the string parsed. As we show, creating and handling strings is expensive, requiring a lot of additional simulation overhead to achieve. Later versions of the BFMs may change the format of the debug output string, requiring any string parsers to be subsequently updated. For new protocols under development the debug output strings can change frequently, whereas older protocols this is less of an issue.

The effect of simply increasing the verbosity on simulation performance can be quite stark, leading to longer run times and larger log files. We took a typical back-to-back testbench of two AMBA Distributed Translation Interface (DTI) [2] protocol (BFMs) and ran a quick sanity check sim at UVM_NONE and the same test at UVM_FULL. The simulation time was in the order of 100x slower (note this is heavily dependent on the storage solution and the test). An hour-long sim could suddenly take days to reproduce. The size of the log file went from 25Kb to 524Mb. We then ran the same simulation at UVM_FULL with the action set to UVM_NO_ACTION. This approximates the simulation cost of using the UVM report catcher, without doing any special message parsing to filter (which would add additional cost). The simulation time in this instance was more than 3x slower. Using our method of verbosity escalation to pick out just one transaction of interest kept the simulation time to a statistically insignificant 0.5 seconds slower with a log size of 70Kb. The pertinent information is rapidly available.

Asides from the cost on simulation from generating such large logs, there was the additional problem of navigating the log to get to the relevant information (a half-gigabyte log is too unwieldly to use effectively). For example, when chasing a failure to write data to memory, the failure is likely to be caused by transactions on a particular address (or range of addresses). All other transactions are likely to be irrelevant for debug. It would be useful to find a method for easily filtering out transactions which weren't on the desired address, without incurring the penalty of creating debug messages for irrelevant transactions.

II. Our Method

Our method started out as a mechanism for filtering transactions out that weren't on a particular cache-line. When dealing with multiple protocols, the debug output of transactions could vary in formatting and style. Given the simulation cost in producing debug messages, it would be desirable to filter the transactions out before the message is created. TABLE *1* gives an indication of the cost of creating debug messages by running a sim with UVM_FULL and UVM_NO_ACTION to filter out all debug messages, the simulation running nearly 4 times slower, and this is without processing the message to determine if it's interesting.

UVM already has a method of filtering messages using the verbosity mechanism which we wanted to work with. This resulted in the design decision of using a mechanism to determine if an address should 'escalate' the verbosity from a default of UVM_FULL to UVM_NONE. This would than go through the standard UVM logging mechanism as before.

The goal therefore is to filter out irrelevant debug information by dynamically changing the verbosity based on characteristics of the transaction. Cutting out irrelevant debug can result is much smaller logs and a smaller performance penalty when debugging. Our method is to provide a debug interface class that provides an Application Programming Interface (API) in which to query whether a piece of information (i.e. a particular address) is being targeted for verbosity escalation. The debug verbosity can therefore be dynamically altered or escalated from UVM_FULL to a higher verbosity (i.e. UVM_NONE). For example, a user may wish to highlight any classes that contain a particular address.

To enable filtering by address, a mechanism to get the address into the simulation is first required. There are several mechanisms for doing this. Our original method was to use the configuration database, as UVM automatically provides command line parsing and permits it to be enabled by scope. e.g.

```
+uvm_set_config_string='uvm_test_top.*,address_debug,0xffffffc0'
```

Later implementations also support direct plusarg use e.g.

```
+vip_address_debug=0xc0,0x1000
+vip_address_debug_scoped=uvm_test_top.env.transmitter*,0xc0,0x800
```

Both these methods permit the simulator to be re-invoked without compiling. The advantage of using plusargs without the configuration database is to permit multiple addresses to be added in a CSV format, and we found it to be faster within simulation, especially with using a global enable to cut down on regex scope matching.

This adds a small minor step in front of the debug message where the verbosity level is first calculated before a call to *uvm_report_enabled*.

```
virtual function
bit is_debug_enabled(string id_, bit escalated_, output uvm_verbosity verbosity_);
  verbosity_ = escalated_ ? UVM_NONE : UVM_FULL;
  return uvm_report_enabled(id_, verbosity_);
endfunction : is_debug_enabled
```

Figure 1. Determine required verbosity level and if the verbosity level is enabled

To handle whether an address is being targeted a class is used to handle all the parsing and storage, with the addresses stored in a hash for fast lookup. It's represented below as a call to *is_debugging_address*.

Once it has been determined that the verbosity is enabled for that address, then a call to *uvm_report_info* can then be made to output the string.

```
uvm_verbosity verbosity;
bit debug_enabled;

debug_enabled = is_debug_enabled("MEM_MODEL",
  is_debugging_address(address_), verbosity);

if (debug_enabled)
begin
  // Construct debug message and output
  uvm_report_info("MEM_MODEL",
    $sformatf("Writing to address 0x%0h", address_),
    verbosity, `uvm_file, `uvm_line, , 1);
end
```

Figure 2. Integrating verbosity escalation with UVM reporting

This also works with the existing *`uvm_info* macro. The advantage of splitting it is for when multiple debug messages are present in a function so that whether the report is enabled or not is effectively cached.

```
uvm_verbosity verbosity;
verbosity = is_debugging_address(address_) ? UVM_NONE : UVM_FULL;
`uvm_info("MEM_MODEL", $sformatf("Writing to address 0x%0h", address_), verbosity)
```

Figure 3. Alternative integration with `uvm_info

As the standard UVM reporting mechanism is then used, then any existing additional post-processing, whether via callbacks or actions set in a report catcher or server will function as before, but with less work and filtering to do.

III. Scaling Up

This works well in memory models and caches, but it does not scale very well for escalating verbosity for different transactions. For example, a user may want to see debug output in both a monitor, driver, scoreboard or sequence for sequence items that contain an interesting address. The transaction may have conditions such as whether it's carrying an address or not. If the component is shared between multiple protocols, then the component won't even be aware the transaction may be carrying an address.

It would be better to have the transaction determine itself if it has an address to be escalated. To get the address debugging information to the transaction/sequence item, an interface class is used to provide an API.

```
interface class vip_debug_interface;

  // Return 1 if verbosity escalation enabled
  pure virtual function bit is_debug_escalation_enabled();

  // Return 1 if debugging the requested address
  pure virtual function bit is_debugging_address(vip_types::address_t address_);

endclass : vip_debug_interface
```

Figure 4. A debug escalation interface

Note this doesn't have to be an interface class. It does allow a component to implement the debug interface, and thus can use itself as the interface, which enables verbosity escalation via scoping. We implement the functionality in a template class which can sit on top of any UVM component. We also implement the same interface class in sequences which divert to the sequencer's report handler. This was chosen because it enables verbosity escalation to be enabled only on components within a particular scope.

Now the sequence item/object can be made aware of the debug interface and can use the API provided to work out if the user is interested (in this case, the address). This is split into two functions, a base class one not intended for override and a user one that can be overridden to provide additional functionality. The goal here is simulation performance. It is the expectation that regressions are the common use case, in which case verbosity escalation would not be enabled (or can even be compiled out, a compile define VIP_DISABLE_DEBUG_INTERFACE is shown in

Figure 5 that achieves this – our profiling shows the effect is small and will need to be balanced out against the need to re-compile for reproduction).

```
// Base class implementation
function
bit is_debug_escalated(vip_debug_interface debug_interface_);
`ifndef VIP_DISABLE_DEBUG_INTERFACE
  if ((debug_interface_ != null) &&
      debug_interface_.is_debug_escalation_enabled())
  begin
    return is_user_debug_escalated(debug_interface_);
  end
  // Else, no debug escalation provided
  return 0;
`else
  // In batch, never escalate
  return 0;
`endif
endfunction : is_debug_escalated

// The extended class can query the internal functions
// Each protocol transaction sequence item can implement this function
virtual function
bit is_user_debug_escalated(vip_debug_interface debug_interface_);
  // No verbosity escalation
  return 0;
endfunction : is_user_debug_escalated
```

Figure 5. Using a debug interface within a class/object/sequence item

The individual protocols can then handle any address debug internally. This provides a single point of maintenance and allows shared components to know if the user is interested in the class without being aware of the type of class.

```
// The extended class can query the internal functions
// Each protocol transaction sequence item can implement this function
virtual function
bit is_user_debug_escalated(vip_debug_interface debug_interface_);
  if (!super.is_user_debug_escalated(debug_interface_))
  begin
    // Address escalation (if valid for this transaction)
    return is_address_valid() && debug_interface_.is_debugging_address(address);
  end
  // No verbosity escalation
  return 0;
endfunction : is_user_debug_escalated
```

Figure 6. Example implementation within a protocol specific sequence item

Using this method the address can be quickly queried for any transaction that supports it and is immune to message parsing errors (i.e. hexadecimal versus decimal debug output), varying positions. Crucially it does not require the message to be created first, improving simulation performance by reducing string handling.

Targeting a particular address is a common use case, particularly when tracking data loss. This approach is powerful in and of itself, but it does have limitations. It requires the BFM implementer to always implement the address functionality for instance, and it does not provide the user with full control. Not all transactions have an address, or the address may not be the pertinent issue. It is also not possible for the BFM designer to anticipate all possible requirements of the user. For example, they may wish to track a particular transaction ID, or a particular ARID/AWID for the AMBA AXI [3] protocol. It would be desirable to have a mechanism that a user could program.

IV. Beyond Filtering Addresses - Custom verbosity escalation

The goal of our method is enable the user to interrogate a class to determine if it is relevant for debug without having to recompile or first incur the penalty of creating the debug message. Our method allows the user to create a debug helper class which uses a delegate pattern. The *vip_debug_interface* class can be updated to support this "custom" debug, and the sequence item/object *is_debug_escalated* function can call these debug helpers with a pointer to itself.

The debug helper classes can then be added to the BFMs in the test bench, however that would require the test bench code to be modified and re-compiled. It would be simpler if the classes could be instantiated from the command line. There is an existing pattern within UVM that already does this: the selection of the test to run. From a library of tests, a plusarg is used to pick a test name and the UVM factory instantiates the test.

The ability to create a class from a string is already embedded into the UVM framework and can be adopted here to create debug helper classes. As the plusargs are themselves strings, it is possible to format the string to give a class name and a list of parameters to instantiate. The main limitation is the parameters can't use characters that are used within the shell used to launch the command, though this can be worked around by having the parameter point to a file where there's more flexibility.

To take advantage of this mechanism a base UVM object class is required. Any classes derived from the factory then should register themselves with the UVM factory using the *`uvm_object_utils* macro. All UVM classes inherit from *uvm_void*, so this can be used as a generic base class. The debug helper class would benefit from being able to parse parameters to make the debug helper configurable, which improves re-use.

```
virtual class vip_custom_debug_helper extends uvm_object;
  // The user implementation needs to declare `uvm_object_utils
  // for the class to be created using the UVM factory

  function
  new(string name = "vip_custom_debug_helper");
    super.new(name);
  endfunction : new

  // Requires user override. This function is called passing in
  // a UVM class object
  // The helper class should then interrogate the class (if it
  // matches an expected type) and return whether to verbosity escalate
  pure virtual function bit is_debugging(uvm_void class_);

  // User provided parameters for the debug helper to parse
  // that configure the helper
  pure virtual function void set_parameters(const ref string parameters_);

endclass : vip_custom_debug_helper
```

Figure 7. Abstract base of a debug helper class

The debug interface class (*vip_debug_interface* implementation) can then parse any plusargs on the command line to look for custom debug helper classes. The actual parsing is left entirely to the user, the BFM developer does not need to be aware of it. This enables any number of schemes to be used for the parameter string.

For example, the verification engineer may want to see all AMBA AXI [3] transactions where ARID/AWID is 5 or 7. A debug helper class that looks for a list of IDs can be used with (for example) a comma separated value (CSV) list of interesting IDs used as the parameter string. Similarly to how +vip_address_debug was used for targeted address debugging, the user can now target anything with a particular ID. The debug helper simply has to cast the *class_* parameter passed into *is_debugging* and on a match, check if the ID matches one of the parsed values.

```
+vip_custom_debug=vip_axi_arid_debug_helper:5,7
+vip_custom_debug_scoped=uvm_test_top.env.transmitter*,vip_axi_axid_debug_helper:1,3
```

The debug helpers can then easily be created via the class name provided in the string, which any optional configuration parameters passed in (plusarg parsing not shown).

```
uvm_factory factory = uvm_factory::get();
// A queue of debug helpers enables multiple helpers to be used in simulation
Vip_custom_debug_helper debug_helpers[$];

// Parsing of plusargs not shown - uvm_cmdline_processor can be used
virtual function
void parse_entry(const ref string classname, const ref string parameters);
  vip_custom_debug_helper helper_class;
  uvm_object object;

  // Attempt to create the helper class from the string
  object = factory.create_object_by_name(classname,
             parent.get_full_name(), "");
  // Objects created are uvm_objects. Test for success and cast
  if ((object != null) &&
      $cast(helper_class, object))
  begin
    // Set the parameters, if provided. Allowing the helper to
    // parse its own parameters provides maximum flexibility
    if (parameters != "")
    begin
      helper_class.set_parameters(parameters);
    end
    // Add to the list of debug helpers
    debug_helpers.push_back(helper_class);
  end
endfunction : parse_entry
```

Figure 8. Parsing a plusarg to instantiate a class via string and configure with command line parameters

The example shown in Figure 8 stores the debug helpers in a queue. This permits multiple helpers to be used in the same simulation at the same time, each targeting separate items (e.g. flow control and transaction IDs)

Now that the debug interface can create objects specified on the command line, all that remains is to make the classes aware of it. To achieve this a new function *is_debugging_custom* can be added to *vip_debug_interface*.

```
interface class vip_debug_interface;

  // Return 1 if verbosity escalation enabled
  pure virtual function bit is_debug_escalation_enabled();

  // Return 1 if debugging the requested address
  pure virtual function bit is_debugging_address(vip_types::address_t address_);

  // Return 1 if debugging the class
  pure virtual function bit is_debugging_custom(uvm_void class_);

endclass : vip_debug_interface
```

Figure 9. Updated debug interface to support using custom debug helpers as call backs

The implementing class needs to check the class against all the provided debug helpers.

```
virtual function
bit is_debugging_custom(uvm_void class_);
  // Test the class against each registered debug helper
  foreach (debug_helpers[i])
  begin
    if (debug_helpers[i].is_debugging(class_))
    begin
      return 1;
    end
  end
  return 0;
endfunction : is_debugging_custom
```

Figure 10. Calling the debug helpers

The default implementation of *is_debug_escalated* can then call any provided debug helpers. Note that this does not conflict with address debug. Again the define *VIP_DISABLE_DEBUG_INTERFACE* is used to compile out the functionality (it can also be used to compile out the debug helpers themselves if not required).

```
// Base class implementation with custom debug
function
bit is_debug_escalated(vip_debug_interface debug_interface_);
`ifndef VIP_DISABLE_DEBUG_INTERFACE
  if ((debug_interface_ != null) &&
      debug_interface_.is_debug_escalation_enabled())
  begin
    return debug_interface_.is_debugging_custom(this) ||
           is_user_debug_escalated(debug_interface_);
  end
  // Else, no debug escalation provided
  return 0;
`else
  // In batch, never escalate
  return 0;
`endif
endfunction : is_debug_escalated
```

Figure 11. Integrating it all together

The verification engineer can write as many debug helper classes as they want by implementing the *is_debugging* function in *vip_custom_debug_helper* and invoke them when reproducing a failure using plusargs on the command line. *Figure 12* shows an example that highlights AXI transactions on a particular ARID and AWID. It shows an example of parsing a comma separated value list of IDs.

The custom debug helper methodology doesn't impose any rules on how to decode these parameters. Other examples with more complex parsers could include key-value pairs, simple Boolean logic, or even a reference to a file containing an entire scripting language. With a debug helper with a sufficiently flexible configuration set, a simulation can be re-invoked with a different parameter set to target additional (or fewer) classes without having to recompile.

This is a simulation performance overhead from calling *is_debugging_custom* for every class prior to determining the verbosity of a debug message, but as shown in Table 1 and Table 2, this is small, particularly in comparison to the cost of first creating the debug message then post-filtering.

```
// Class: vip_axi_axid_debug_helper
// This helper object prints out transactions on a specified AxID
class vip_axi_axid_debug_helper extends vip_custom_debug_helper;
  // Register with the factory
  `uvm_object_utils(vip_axi_axid_debug_helper)

  // A hash of interesting AxID values
  protected bit axids[vip_axi_types::id_t];

  // Constructor
  function
  new(string name = "vip_axi_axid_debug_helper");
    super.new(name);
  endfunction : new

  // Determine if the class is interesting
  virtual function
  bit is_debugging(uvm_void class_);
    vip_axi_transaction_item item;
    if ($cast(item, class_))
    begin
      // Escalate if the transaction has an ID that matches the provided list
      return axids.exists(item.get_axid());
    end
    return 0;
  endfunction : is_debugging

  // Expect a number which corresponds to the AxID. Multiple IDs can be given
  // by providing a list of AxID values as a comma separated list
  virtual function
  void set_parameters(const ref string parameters_);
    string fields[$];
    uvm_split_string(parameters_, ",", fields);
    if (fields.size() > 0)
    begin
      vip_axi_types::id_t value;
      foreach (fields[i])
      begin
        // Determine AxID from field
        // NOTE: Assume base is decimal
        if (vip_string_to_integer#(vip_axi_types::id_t)::get_value_with_base(
              value, fields[i], 10))
        begin
          // Add to the set
          axids[value] = 1;
        end
      end
    end
  endfunction : set_parameters

endclass : vip_axi_axid_debug_helper
```

Figure 12. Example custom debug helper for highlighting AXI transactions on a particular ARID/AWID

Note that the custom debug implementation is not a complete replacement for address debug because it requires a class. Though this can be worked around by putting the address in a class first it adds complexity in the path of coding, particularly when the address is passed in as a parameter in a function.

V. Results

TABLE 1. REPRESENTATIVE DTI SIMULATION

	UVM_NONE	UVM_FULL (UVM_NO_ACTION)	UVM_FULL	Custom Debug
Simulation time (seconds)	17.64	58.11	1822	18.05
is_debug_escalated simulation time (seconds)	0.08	0.18	0.1	0.34
Log size (Kb)	24.8	64.5	523,641	69

Table 1 details the results from a simulation performance test for a DTI protocol back-to-back simulation. DTI sits on top of AXI-Stream. These results were recorded on the following set up

- Siemens Visualizer 2023.2
- UVM 1.2
- Cloud using x86 architecture

The results demonstrate the difference in simulation performance between UVM_NONE and UVM_FULL. The UVM_FULL (UVM_NO_ACTION) run filters all UVM_INFO messages in the simulation (from connect phase onwards). This is a proxy for the cost of generating debug messages. For the DTI protocol this is in the order of 3x slower in simulation. No attempt is made to inspect the messages to filter by regex.

Conversely it can be observed that outputting to a log is very expensive, essentially going from 1 minute to 30 minutes. As this is cloud based, there's also the question of additional cost to access all that data. A simulation on a local machine is likely to be faster. Even with faster storage a file that's half a gigabyte is cumbersome to load and search.

The Custom Debug column details the cost of the verbosity escalation method to pick out all transactions on a particular translation ID. It can be observed that the log size does not grow substantially, and the additional cost in terms of simulation performance is minimal. For reference, the simulation time spent in *is_debug_escalated* has been included.

TABLE 2. REPRESENTATIVE AXI SIMULATION - FILTER BY AXID

	UVM_NONE	UVM_FULL (UVM_NO_ACTION)	UVM_FULL	Custom Debug
Simulation time (seconds)	33.63	39.06	490	67.53
is_debug_escalated simulation time (seconds)	< 0.1	< 0.1	0.1	0.1
Log size (Kb)	34.3	49.6	75,725	1,642

Table 2 details the results from a simulation performance test for an AXI protocol back-to-back simulation. This test filters by ARID/AWID as described earlier. Our AXI BFM has less verbose debug output, which will reduce the impact of generating the messages. This can be seen above with the smaller log files, and correspondingly smaller penalty for generating messages. Even in this less verbose scenario, there's a greater than 10x simulation performance hit when increasing verbosity to UVM_FULL.

The results do however show that the penalty for filtering the messages before creation (0.1 seconds in *is_debug_escalated*) is still an order of magnitude less than the cost of generating the messages to filter later (over 5 seconds). The cost of filtering in the AXI BFM (0.1 seconds) is similar to the much more verbose DTI BFM (0.34 seconds), indicating that there is not much penalty for making the BFM more verbose in its debug output using this method.

VI. Conclusion

The results given in Table *1* and Table *2* show the simulation performance overhead of both creating and outputting debug messages to the log. They also show the size of the logs can quickly become unwieldy, particularly (but not limited to) general purpose BFMs. All the UVM filtering mechanisms are post message creation. Given the simulation penalty of logging, any form of filtering is a productivity win, but it is even better to filter the message out before it's even created. Avoiding doing unnecessary work is an excellent way to increase productivity and efficiency.

Our method provides a mechanism that ultimately aims to enhance the functionality of *uvm_report_enabled*. It does this by dynamically changing the verbosity before the message is created, in such a way as to be highly configurable by the user, in many instances (for example, with address-debug or a sufficiently flexible custom debug helper) without requiring a recompile.

The custom debug helper mechanism permits the user to inspect the content of objects and transactions used within the BFM. This is far faster than parsing a string output and allows other more complex logic style operations to be used for filtering the message. This is down to the imagination of the user. The use of the UVM factory to dynamically create the debug helpers further enhances this capability beyond a simple delegate callback pattern by allowing the debug helpers to be dynamically selected at run-time. This is similar in many ways to being able to change the verbosity mid-simulation, only can now be done on a per-object basis. It is achieved in a manner that does not preclude the additional use of the provided UVM filtering mechanisms, we believe it enhances them by providing them with much less filtering work to do.

The examples provided refer to its use in escalating verbosity for debug output (either to a transcript or log file), but it can also be used outside of debug reporting to identify specific classes or transactions to output to visualization streams or other custom debugging tools. By only highlighting relevant transactions, the reproduction time to failure is much faster, resulting in a more efficient workflow.

We hope our method demonstrates that there is additional value is filtering messages before they are even created. Indeed, we have found that because the penalty of debug is reduced, it encourages better and more verbose debug information as the costs are lower.

References

[1] IEEE Std 1800.2 TM, IEEE Standard for Universal Verification Methodology Language Reference Manual, 2020.

[2] AMBA (Advanced Microprocessor Bus Architecture) Distributed Translation Interface (DTI) – ARM IHI 0088 Issue G (developer.arm.com)

[3] AMBA AXI – ARM IHI 0022 Issue K (developer.arm.com)