Navigating your way toward UVM version 1.2

What's new about UVM 1.2

David C Black

Eileen R Hickey

Doulos America

San Jose, CA USA

www.doulos.com

ABSTRACT

This paper examines features of v1.2 of the Universal Verification Methodology (UVM). We exercise and measure the performance of specific features in the current and earlier versions of UVM to demonstrate their performance impact. Using DPI code to obtain CPU performance information, we demonstrate advantages of UVM 1.2 in both performance and coding safety. Our research includes some of the compatibility aspects of UVM 1.2 to consider while making the transition. Suggestions help leverage the improvements while avoiding the potential pitfalls of migrating to v1.2 in your verification environment. We also make some recommendations on UVM guidelines that should help with interoperability and coding clarity.

Table of Contents

Table of Figures

1. Introduction

This paper investigates some of the issues surrounding the latest version of the Universal Verification Methodology known as UVM 1.2. The sub-title, What's new about UVM 1.2 refers to improvements made with v1.2 and issues regarding some of the changes. For example `uvm_objection`, a mechanism used to determine when to stop simulation, has been enhanced for better performance. However, more than one individual has voiced concern over the release of a standard that is not completely backward compatible with its predecessor. For example, a few enumerated values have been renamed to have a `UVM_` prefix requiring global changes in established code. As the major justification for allowing backward incompatibility, this Accellera standard provides the basis for an IEEE standard UVM. IEEE standards are well known for their adherence to backward compatibility. We will consider both the positive and negative impacts of the features and make suggestions for moving forward.

Despite the controversy, anybody doing verification needs to consider the latest proposed version of UVM, whether they are using it currently or about to. The topics discussed should be understandable[1] even if you are not currently a user. This paper examines some of these issues. This information should be helpful for both planning and adoption of UVM (any version).

Before diving into the core of this paper, let's briefly take a step back and consider what UVM provides. The following are the main benefits of UVM:

- A single standard fully supported by multiple EDA tool vendors.
- A single standard that leverages knowledge to allow engineers to have a common understanding.
- Standard spatial organization of the test environment provides guidance on how various verification components connected to each other and to the RTL.
- Standard temporal phases that control how the test proceeds.
- Standard mechanisms for controlling configuration of tests and components.
- Standard description of memory mapped registers and memories.
- Based on a common grammar to enable focus on the methodology rather than tools.
- Ability to create truly reusable Verification IP (VIP) for inter/intra company use.

UVM also has some minor downsides including:

- Requires learning object oriented programming (OOP) concepts that may prove challenging to some hardware designers.
- Reuse requires overhead to properly implement.
- Simple projects may become overly complex.
- It is not always obvious how to apply UVM to some problem domains.

It is also worthwhile to consider what UVM does not provide. In particular, UVM does not provide a complete step-by-step methodology. More to the point, different companies with different needs tend to apply it differently. Whereas one camp of UVM enthusiasts insists that constrained randomization with minimal steering from the test writer provides the ideal results;

[1] Short primers on some topics are provided later.

others insist they must direct more of the test activities due to the nature of their designs. Both groups are correct in the sense that UVM allows them to consistently ship their designs with higher quality and fewer bugs, therefore everybody wins.

UVM has a rich history as seen in Figure 1. In fact, all of the features of UVM have evolved from existing methodologies. As one would naturally expect, many of the contributors to the standard are eager to see their favorite features incorporated into the latest version, occasionally leading to additional complexity.

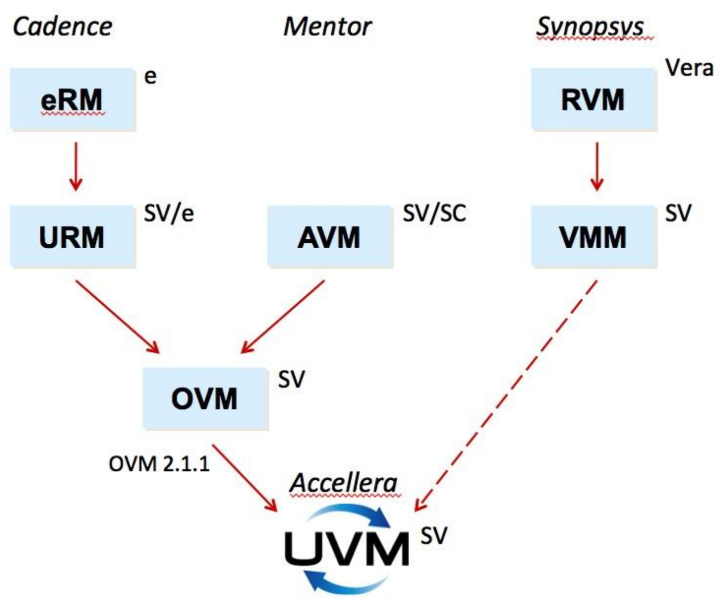


Figure 1 - UVM History

2. An Overview of UVM 1.2 Features and Concerns

In this section we review some of the changes and issues with the UVM 1.2. [1] [2] [3]

UVM 1.2 benefits

- Bug fixes
- Performance fixes
- Cleanups
- Enhancements
- API and semantic changes
- User guide cleanup and update
- Improved checks for compliance with the standard

Notable changes

- Completely revamped reporting scheme
- Message severities now `enum` rather than simple `integer`
- `set_propagate_mode()` removes rippling of objections upward through the hierarchy

- Automatic objections
- Guarded starting phase
- Data access policies added for safety (`uvm_set_before_get_dap`, and `uvm_get_to_lock_dap`)
- Conversions from `string` to `enum`
- New `uvm_integral_t`
- VHDL backdoor support for the register model (DPI)
- Transaction ordering policy for registers
- Renamed `enum` values with `UVM_` prefix
- Deprecated `set_config_*`, `get_config_*`
- Removed `uvm_pkg::factory` global reference
- Default sequences set from the command-line

Backward incompatibility

Let's talk about the importance of backward compatibility in the software world. Accellera's UVM v1.2 is not entirely backward compatible with v1.1. Why the fuss? UVM has been around since 2010, touted as the Universal Verification Methodology. Many companies accepted it as the solution to interoperability across simulators and engineering teams. Let's consider the learning curve to have a team absorb this new methodology as their own. Fast-forward five years - how many UVM components and lines of UVM code have they created to support their verification needs? Now change just 'some' of the methodology, have the team re-learn and adopt a new process for the methodology and adapt how many lines of code to the 'new' way? Backward incompatibility directly opposes the desired tighter schedules engineering teams look for as they mature in their use of UVM reusable components.

The other side of the backward compatibility question is how to go about migrating your old code? Should you just switch to v1.2? After all, that is what the IEEE specification will most likely be based on. What about maintenance or product engineering needs? What about future possible users of the internal UVM IP – how can they easily tell that a UVM model in the library is v1.2 compatible? So much for running a Perl script and moving on... Many considerations to be debated, one is which UVM models are worth the effort to upgrade? Are those models you bought from the external vendor written for v1.1, v1.2 or both?

It is rather presumptive that users would simply apply the UVM v1.2 provided Perl script, `uvm11-to-uvm12.pl`, to their code. When a project has many hundreds of simulation hours verifying a shipping design already under UVM, the risks associated with using such a script are not insignificant.

Backward compatibility is important in many ways. Later in this paper, we point out a way to deal with this.

Focus of data collection

As we reviewed the changes introduced by v1.2, we focused on specifically testing the expected improvements in the objection overhead. We did this mainly because objection overhead has been an issue throughout v1.1, culminating in an improvement in v1.1d. We wanted reassurance that the improvement changes in v1.2 were measureable. Since we were creating numerous transactions to get statistically reasonable numbers for the objection measurements, it seemed useful to add measurements of the reporting mechanism changes in v1.2 since it had been extensive and appear to add complexity. The intent was to measure if the reporting changes in v1.2 were measurable vs reporting in v1.1d.

Quick Primer on objections

For those that may not be familiar with UVM's simulation control mechanism known as uvm_objection, here are some basics. [4]

UVM uses a concept known as objections to determine when a simulation has completed its work. The idea is that components actively processing transactions should raise an objection to indicate their need for the simulation to continue. In other words, the component objects to stopping the simulation. This reduces the burden on the top-level test, and enables the idea of independent verification IP by allowing each verification component to independently indicate completion by lowering its objection. Components may even raise and lower objections repeatedly as long as objections are raised within the special timeout known as the drain time. **Error! Reference source not found.** illustrates this concept.

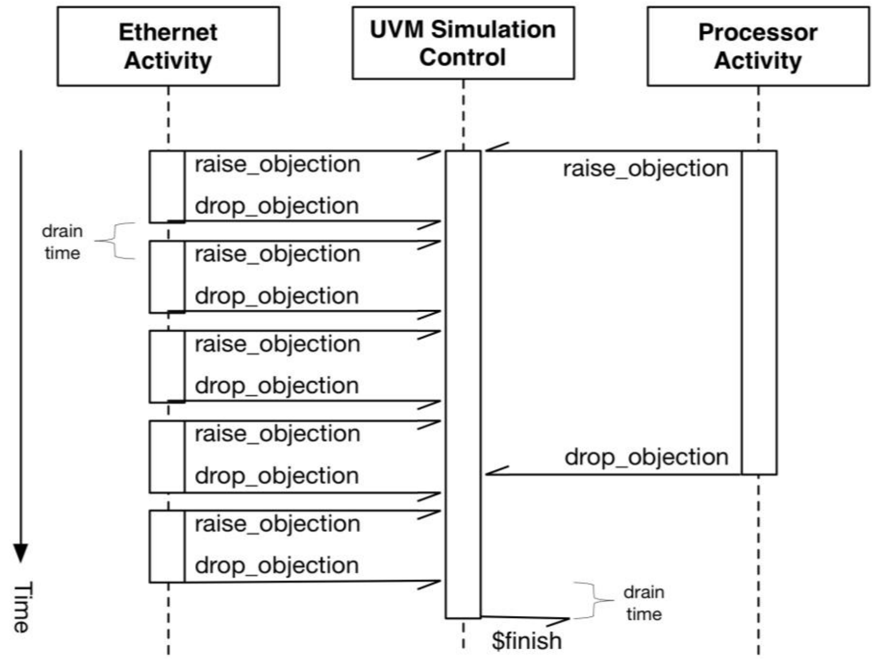


Figure 2 - Objections

When individual components raise and lower objections per transaction, there is an associated processing overhead. If there are a large number of transactions, then the overall simulation performance can be degraded. Additionally, UVM v1.1 had a feature to aid debugging that caused objections to propagate upwards through the verification hierarchy, which added even more overhead. UVM v1.2 allows us to disable objection propagation, while making it available to use in objection debug.

3. Techniques for Measuring the Performance Impact

Given all of the above, how shall we proceed? We've already decided to measure the timing aspects for two of the most debated features: UVM objections, and the completely revamped reporting mechanisms. Before we report on our observed results, we must first consider how to properly measure these features.

A real complaint from end-users is that some of the new features may slow down already painfully slow simulations. In particular, the revamped reporting mechanism adds a lot of code. On the other hand, users look forward to features that purportedly will speed up their simulations, namely improving objection overheads. So our measurements will have to focus on processor clock time consumed during simulation. Additionally, we want to ensure that our measurements focus on the features and not on aspects used to create the environment. This means we need some method of extracting time information that allows us to focus on specific features being measured.

In order to minimize the impact of the design on our measurements and to avoid months of effort designing, we created a small scalable verification environment, as illustrated in **Error! Reference source not found.**. UVC is the generic 'UVM Verification Component' notation. We architected the hardware model to be as simple as possible so that it did not affect the timing outcome as the verification environment changed.

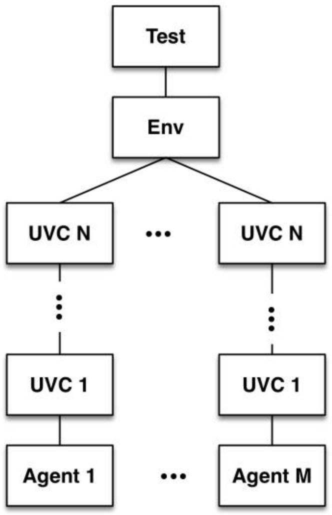


Figure 3 - Scalable verification environment

We created support functions to obtain timing information using operating system calls via SystemVerilog's Direct Programming Interface for C (DPI-C). The following figure defines our SystemVerilog API for this timing information. To ensure that we only measure execution of the run-time code and not the setup (e.g. `build_phase`, `connect_phase`, etc.), we placed calls to extract the start and stop of the simulation during the run phase precisely around beginning and ending of the relevant activities. To obtain statistically valid numbers, we simply increased the number of transactions into the millions. Each simulation ran for more than an hour.

```
import "DPI-C" get_cpu_time  // elapsed in CPU seconds
                 = function real get_cpu_time();
import "DPI-C" get_wall_time // wall clock in seconds
                 = function real get_wall_time();
```

Figure 4 - API for Measuring Time via DPI

The implementation code for the preceding API uses suggested techniques from the web [5] and even allows for platform independence[2].

Contrary to Doulos' general UVM coding guideline recommendations, we chose to put all the SystemVerilog UVM source code into a single file, named `performance.sv`. This simplifies editing and reviewing of the contrived code for our experiments.

Another consideration was the creation of code that could represent arbitrary, but controllable complexity. For UVM, this is easily controllable at run-time; however, a few hardware features had to be coded with macros. To augment this, we built a scripting environment around the code that would allow us to easily run our code on all[3] the EDA vendor simulators[4].

We then considered what would constitute the complexity that might affect run-time behavior. Certainly size would be one aspect, but depth of our verification environment needs also to be varied since one of the interesting UVM 1.2 intended improvements has to do with limiting the rippling of objections as they are passed up the verification hierarchy, see Figure 5.

Because we are not interested in testing the DUT[5] at all, for these tests, we used a single interface and simply connected up all the agents to the same interface instance and a single piece of hardware for the DUT. This means that verification driver writes contended with the logic, and functionally SystemVerilog simply does "the last driver wins". Although no real design would do this, it allowed us to focus our measurements on the verification environment and avoid measuring effects due to scaling of the DUT itself. In other words, the DUT is not scaled.

[2] In case you happen to be using one of the few simulators that works on Windows.
[3] "All" simulators in this case refers to ALDEC, Cadence, Mentor and Synopsys.
[4] See Disclaimer in section 7.
[5] DUT (design under test) was used in preference to DUV (design under verification) simply because we feel a larger audience will be familiar with this terminology. The distinction of test vs verification is minor since we are certain our audience will understand.

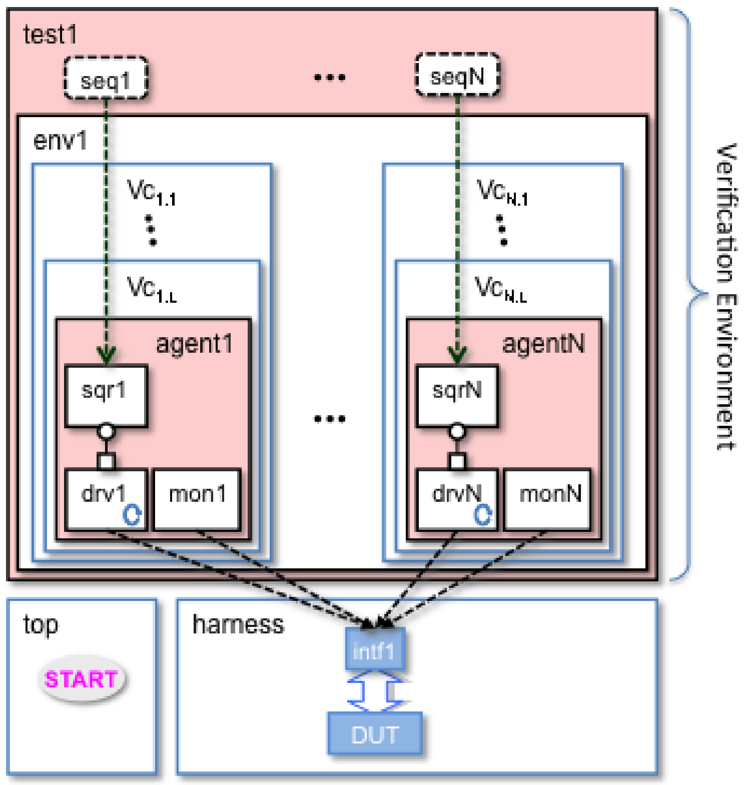
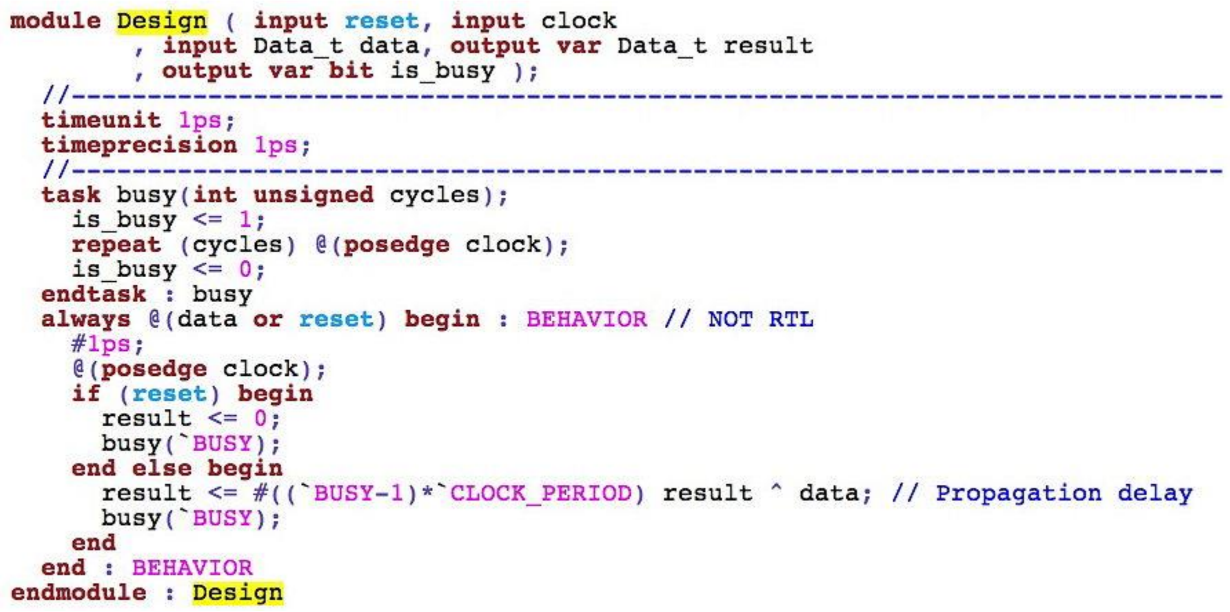


Figure 5 - Scalable Verification Environment Connected to Hardware

The following code describes the entire DUT (Design Under Test). The design is purely behavioral (not RTL).



```
module Design ( input reset, input clock
         , input Data_t data, output var Data_t result
         , output var bit is_busy );
  //-----------------------------------------------------------------------------
  timeunit 1ps;
  timeprecision 1ps;
  //-----------------------------------------------------------------------------
  task busy(int unsigned cycles);
    is_busy <= 1;
    repeat (cycles) @(posedge clock);
    is_busy <= 0;
  endtask : busy
  always @(data or reset) begin : BEHAVIOR // NOT RTL
    #1ps;
    @(posedge clock);
    if (reset) begin
      result <= 0;
      busy(`BUSY);
    end else begin
      result <= #((`BUSY-1)*`CLOCK_PERIOD) result ^ data; // Propagation delay
      busy(`BUSY);
    end
  end : BEHAVIOR
endmodule : Design
```

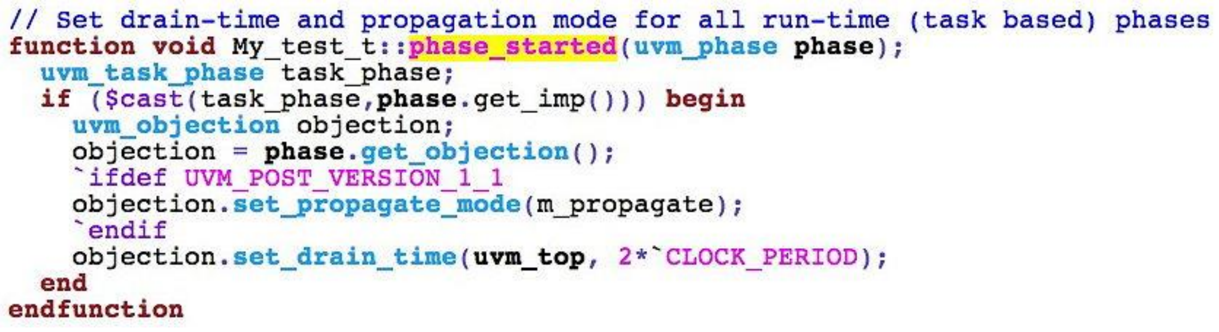
Figure 6 - The Design Under Test

Our measurements are focused on the verification machinery. In one version of the tests, we don't stimulate a design at all. We just provide a simple delay.

4. UVM “Features” Lessons Learned

During development, a few aspects of UVM (aka “features”) reinforced themselves, and are worth noting in this brief list.

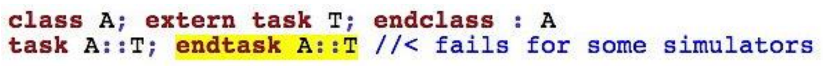
- Drain-times, the special timer associated with when to end the phase with all objections dropped, are attached to task-based phases, so when using objections in more than one task-based phase (e.g. main_phase and run_phase), the drain time has to be set on all phases that use objections.
- Drain-time needs to be set before any objections are raised (i.e. at t=0), but there could be a race condition between processes in different components. To solve this, we use the `phase_started()` method, as shown in Figure 7.
- Propagation mode is similarly attached to a phase's objection. See Figure 7.



```
// Set drain-time and propagation mode for all run-time (task based) phases
function void My_test_t::phase_started(uvm_phase phase);
  uvm_task_phase task_phase;
  if ($cast(task_phase,phase.get_imp())) begin
    uvm_objection objection;
    objection = phase.get_objection();
    `ifdef UVM_POST_VERSION_1_1
    objection.set_propagate_mode(m_propagate);
    `endif
    objection.set_drain_time(uvm_top, 2*`CLOCK_PERIOD);
  end
endfunction
```

Figure 7 – Setup with phase before objections are raised.

- UVM events and event pools are now parameterized and yield warnings if the data type parameter is not provided.
- Not all simulators fully support all the SystemVerilog features . We had trouble with labeling the endtask of an externally implemented class member task. As a work-around, we commented out the matching label for those simulators. See Figure 8.



```
class A; extern task T; endclass : A
task A::T; endtask A::T //< fails for some simulators
```

Figure 8 - Unsupported syntax example

5. Understanding the Results

This section provides simulation results for a SystemVerilog simulator running with specific versions of UVM.

Access to the source code can be found on the Doulos website under UVM Know How [6]. Customers may choose to run the tests for themselves. Certainly different versions of the simulators and/or compute platforms will result in different numbers; however, the trends should be similar. IMPORTANT: Comparisons need to be run on identically equipped systems. Therefore, it would be incorrect to compare our specific numerical results here against runs on a different system. Our tests are designed to run multiple simulations on the same machine, and allow you to compare between those simulation sets.

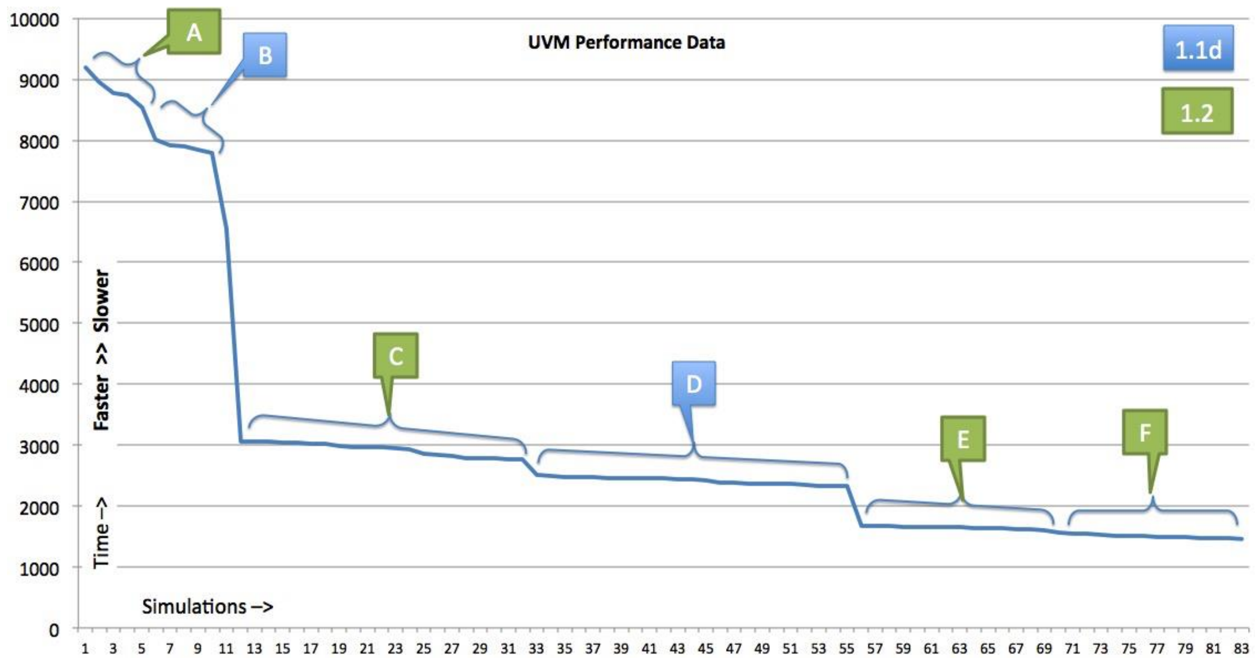


Figure 9 - Simulation Performance Results

In Figure 9, we show elapsed process times for simulations consisting of 21 million transactions each, using UVM 1.1d and UVM 1.2 in a variety of configurations. The data was then sorted from slowest to fastest based on CPU time. We examined the data to see what might be responsible for the trends. Objections were raised and lowered around each transaction, both in the driver and monitor. This resulted in 84 million calls to raise and lower objections during the simulation. The labels indicate various types of simulations features as follows:

- First, the slowest simulations on the left, labeled [A] and [B], represent simulations with a deep hierarchy (32 levels) and objections being propagated across all levels. [A] represents v1.2 and B represents v1.1d. UVM 1.1d has a clear performance edge over v1.2.
- The next large section, labeled [C] and [D], represents simulations still propagating objections, but only across a few levels (4) of hierarchy more typical of UVM designs. Again version 1.1d clearly has a slight performance edge over v1.2.
- Finally, the last relatively flat segment, labeled [E] and [F], represents v1.2 without any propagation of objections across hierarchy. In other words, using the new feature to suppress propagation that is not present in UVM versions prior to v1.2. Segment [F] ran slightly faster than [E], because the 21 million transactions were spread across five agents rather than two, which caused the total simulated time to be reduced. That is, the same number of transactions divided 5 ways instead of two ways – the reduction isn't as significant as by half, though, since higher context switching for 5 agents plays a part in the time consumption.

Clearly**, turning off objection propagation improves performance significantly, by our measurements, approximately 4x, for deep testbenches**. Since the propagation is intended as part of the objection debug, it is not necessary to include it generally in the simulations. The speed difference. This is especially noticable if objections are raised and lowered frequently. It is advised to avoid using objections more than necessary. Putting objection driving in sequences reduces the ratio of objections to transactions. Thus the speed differences noted here become inconsequential, because you will be raising and lowering objections less than a few dozen times rather than millions. Coding considerations for these strategies are discussed in section 'More on Reducing Objection Impact'

Interestingly, we did not see any discernable impact attributable to the reporting, even though some simulations were setup to deliberately exacerbate this aspect. We will continue to investigate this.

6. Conclusion: Using UVM 1.2 for Better VIP

Despite the backward incompatibilities, we feel that UVM 1.2 really is a good choice for verification for all the original reasons. As a standard methodology, UVM provides a stable reference point that can save companies millions of dollars in training. Engineers familiar with the standard have a much easier time coming up to speed on a UVM verification project. Projects also benefit from the ability to purchase commercial VIP, thereby saving months of verification development time for standard interfaces. If you have an interface that is a variation of a standard, you could buy the commercial VIP, and then extend it for your variations. Reusing internal VIP provides the same benefits. In other words, building VIP with reuse is worthwhile.

Our simulation results demonstrated that objection mechanism performance improvements were measurable. That is, v1.2 with objection propagation turned off, demonstrates a clear speed superiority over v1.1d. However, if objection propagation is turned on, then v1.1d has a slight speed advantage.

We also highlighted some of aspects that the end-user can control. Typically, excessive I/O causes significant simulation slow-downs and emphasizes the need to use the UVM messaging macros with careful selection of verbosity. Despite the restructuring of UVM message reporting mechanisms in v1.2, we did not see any statistically measurable timing differences versus v1.1.

Backward Compatibility Advice

Although the safety additions (e.g. `starting_phase` variable) of UVM 1.2 cause backward compatibility issues, using these improvements in v1.2 will lead to better, cleaner code.

One of the notable backward compatibility concerns the `starting_phase` variable, its use-model exhibits one of the new variable use-model safety capabilities in v1.2.

> "Internally, the `uvm_sequence_base` uses a `uvm_get_to_lock_dap` to protect the `starting_phase` value from being modified after the reference has been read. Once the sequence has ended its execution (either via natural termination, or being killed), then the `starting_phase` value can be modified again." [1]

The `uvm_get_to_lock_dap` and `uvm_set_before_get_dap` base classes support the use of these data access policies for user-defined data members, adding to safety of user algorithms.

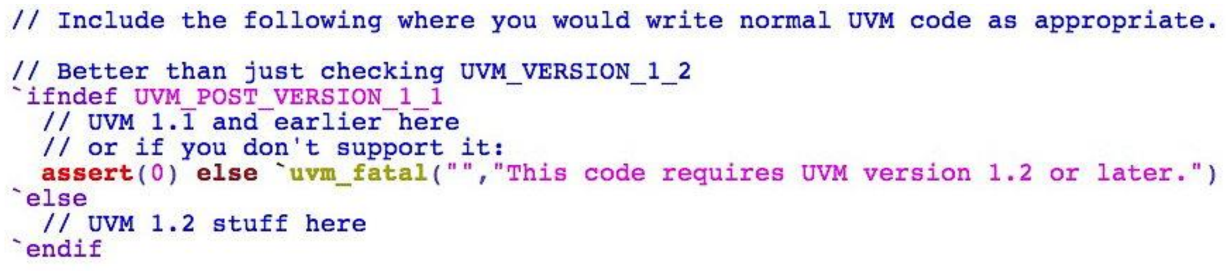
Another backward incompatibility relates to raising objections on non-task (function) phases, that is, phases that do not consume simulation time e.g. `build_phase()`, `connect_phase()`. Although it does define an incompatibility between v1.1 and v1.2 code, the actual use of objections in a function phase does nothing but add to the objection execution overhead -- particularly in a propagated simulation with multiple levels. The actual user intent for using the objection mechanism in a function phase is probably misdirected, so dis-allowing objections in these phases will lead to safer, more efficient code.

Expanding use of the prefix `UVM_` for enumerations also falls into the category of safer coding. This avoids accidental user name clashes. Changes in UVM 1.2 only affected three enumerations, `uvm_sequencer_arb_mode`, `uvm_sequence_state` and `uvm_sequencer_base::seq_req_t`, as well as the typedef `SEQ_ARB_TYPE`.

Deprecating `set_config_*`, `get_config_*` accesses to the config_db, and removing the `uvm_pkg::factory`, reduces the choices and confusion on proper coding of the methodology, providing a safer environment for newbies and eliminating chances for misuse.

Another safety addition is the ability to set enumerations using strings. This allows the creation of enumerations via the config_db, as well as from the command line. Using enumerations instead of hardcoded numbers provides safer and more resusable code.

Finally, for those situations that need to be coded for UVM 1.1 and UVM 1.2, we suggest using conditional compilation. See code in **Error! Reference source not found.**. Assuming UVM 1.2 in its entirety becomes the IEEE-1800.2 UVM standard, IEEE standardization usually disallows backward incompatibility.



```
// Include the following where you would write normal UVM code as appropriate.

// Better than just checking UVM_VERSION_1_2
`ifndef UVM_POST_VERSION_1_1
  // UVM 1.1 and earlier here
  // or if you don't support it:
  assert(0) else `uvm_fatal("","This code requires UVM version 1.2 or later.")
`else
  // UVM 1.2 stuff here
`endif
```

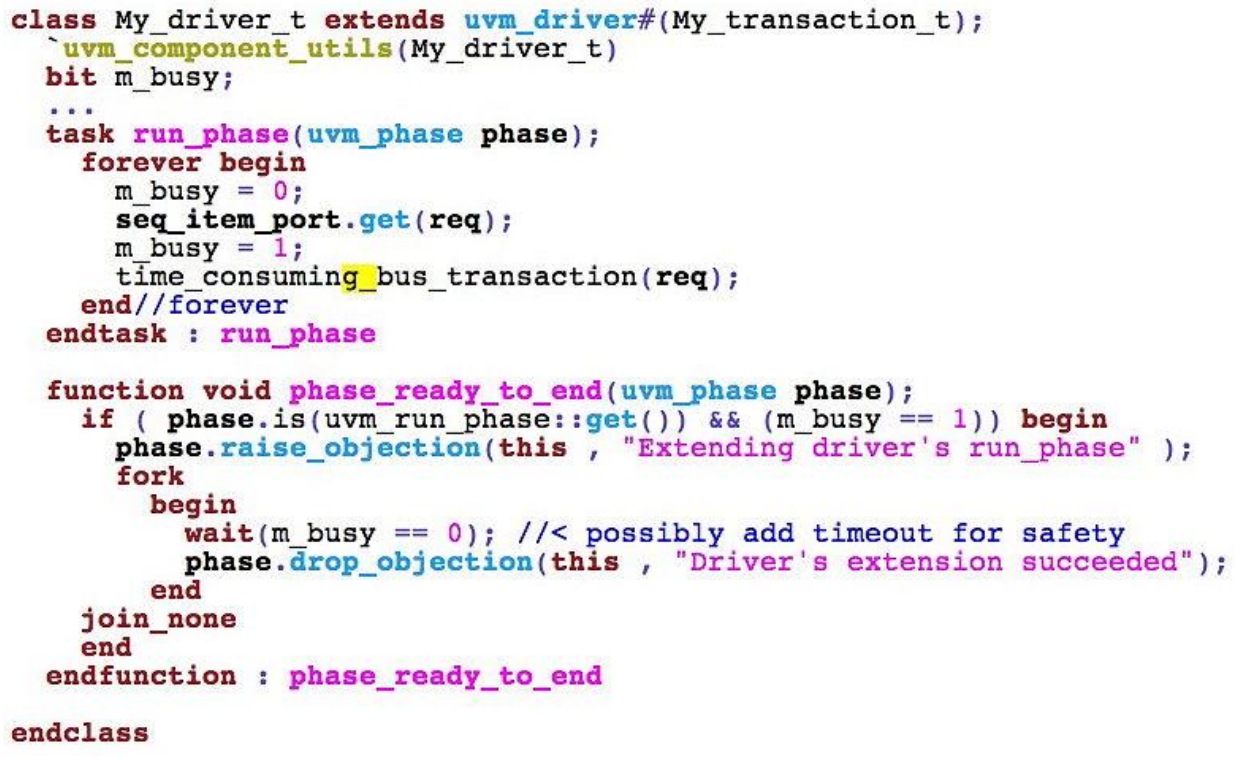
Figure 10 - UVM Conditional Compilation

More on Reducing Objection Impact and Other Coding Considerations [7]

At this point, we offer a few guidelines beyond what is contained in the UVM standard. The following is only a very short list related to the UVM 1.2 standard.

Turning off objection propagation clearly improves wall clock simulation speed. However, consider reducing objections in the first place. Improvements to performance in the UVM library are always welcome, but they are not substitute for good programming. Proper objection handling allows simulation of all intended activity. There is nothing more frustrating than debugging a simulation that terminates early due to insufficient objections. Let's look at one strategy to handle this.

To minimize the objection overhead in a simulation, that is, to reduce the total number of objections being raised and lowered, a top-level virtual sequence can drive a single objection for the entire simulation. Since the sequence hierarchy is establishing stimulus, its objection mechanism often doesn't account for end-to-end time through the design. The last few transactions at the end of the stimulus need time to propagate. Due to this, and in keeping with reducing the objection overhead, the `phase_ready_to_end()` callback can be invoked by the components. This allows the functional completion of those last transactions. See Figure 9 for code. Since `phase_ready_to_end()` has a maximum invocation of 20, this algorithm is specific to final transaction(s) in the `run_phase`.



```
class My_driver_t extends uvm_driver#(My_transaction_t);
  `uvm_component_utils(My_driver_t)
  bit m_busy;
  ...
  task run_phase(uvm_phase phase);
    forever begin
      m_busy = 0;
      seq_item_port.get(req);
      m_busy = 1;
      time_consuming_bus_transaction(req);
    end//forever
  endtask : run_phase

  function void phase_ready_to_end(uvm_phase phase);
    if ( phase.is(uvm_run_phase::get()) && (m_busy == 1)) begin
      phase.raise_objection(this , "Extending driver's run_phase" );
      fork
        begin
          wait(m_busy == 0); //< possibly add timeout for safety
          phase.drop_objection(this , "Driver's extension succeeded");
        end
      join_none
    end
  endfunction : phase_ready_to_end

endclass
```

Figure 11 - Extending the last transaction

Another, more intuitive strategy is to use the `get_next_item/item_done` driver/sequencer handshake as described in Doulos' UVM GRG [4] with a per transaction objection in the associated scoreboard. This handshake requires that the sequence not complete until the transaction is driven to the DUT, or whatever timeframe the driver needs to ensure that the transaction has been driven. Making sure that the drain time is at least two clocks long to avoid the code order issues of a delta cycle, then the scoreboard raising an objection on receipt of the transaction from the monitor as it awaits the 'matching' transaction on the next interface, allows a proper amount of time for the transaction to progress without needing the coding effort above. `phase_ready_to_end` would be unnecessary in this coding strategy.

Another guideline relates to command-line options. Although more command-line options were added to the standard, we suggest they be used with caution. Once you start invoking simulation runs primarily through the command-line, you must put the information into a script to allow reproducibility. Code reviews have to include these invocation scripts, and you should revision control those. Overall, it can make it more difficult to understand what happened during a simulation since you now have to read both the code and the scripts. We recommend limiting command-line options to the original +UVM_TESTNAME= and the randomization seed for regressions. For debug and what-if analysis, the command-line options are a powerful tool.

We also recommend using the safety features noted previously. In particular, use enumerations as SystemVerilog and UVM have powerful features for their use. We continue to see UVM testbenches containing hardcoded numbers. Hardcoded numbers make it difficult for code maintenance, reuse and readability. Many bugs in software and hardware can be attributed to difficult to find constants that are better-expressed and more obvious using enumerations.

A key to success in UVM is using a complete set of coding styles and guidelines. UVM is challenging enough on its own without adding the challenge of navigating multiple coding styles. We strongly suggest you obtain quality formal training on UVM. For more on UVM coding and guidelines, please go to the Know How/Easier UVM coding guidelines section of the Doulos website [8].

Future performance investigations

- more I/O investigation
- automatic phase objection
- uvm_events versus SystemVerilog events
- uvm_do macros
- field_macros
- clocking blocks
- expandable RTL to see how multiple interfaces might affect it
- user-defined callbacks

7. Disclaimer

Benchmarks are fraught with perils for the unwary. Although, the code used to compare and evaluate UVM has undergone a lot of careful thought, there are many factors that affect the results. The Accellara downloaded UVM code is considered by some to be a Proof-of-Concept implementation. Other implementations conforming to the UVM standard API will exhibit different performance characteristics. Also, in real world verification, the ratio of verification environment to RTL hardware differs from that used in our experiments. We purposely skewed the model to emphasize testing of specific features. If your design does not use these features to the same extent, then the impact may not be meaningful. Lastly, simulators evolve over time and can certainly be tweaked to address particular coding styles. It is likely that the particular simulator used for data gathering does not match yours. If you really want to evaluate the claims, you should run your own tests, either with our code or something of your own making. It is suggested that you consider using a project representative of your own design problems.

8. References

[1] *UVM 1.2 Class Reference*, by Accellera
http://www.accellera.org/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[2] *What's New in UVM* presentation from DVCon 2014
http://www.accellera.org/resources/videos/uvmnownext

[3] *UVM 1.2 release notes*, (in release kit), by Accellera
http://www.accellera.org/downloads/standards/uvm/uvm-1.2.tar.gz
http://www.doulos.com/knowhow/sysverilog/uvm/uvm-1.2/bin/uvm11-to-12.pl

[4] *Doulos UVM Golden Reference Guide*, by Doulos Ltd
http://doulos.com/content/products/golden_reference_guides.php#anchor uvm

[5] http://stackoverflow.com/questions/17432502/how-can-i-measure-cpu-time-and-wall-clock-time-on-both-linux-windows

[6] *UVM introduction* by John Aynsley (updated for UVM 1.2)
http://doulos.com/knowhow/sysverilog/uvm/

[7] *Making the most of SystemVerilog and UVM: Hints and Tips for new users*, by Dr David Long
http://www.doulos.com/knowhow/sysverilog/SNUG13_hints&tips/snug_boston2013_paper_56_paper.pdf

[8] *UVM Coding Guidelines*, by Doulos Ltd
http://www.doulos.com/content/events/easierUVM.php