2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Slicing Through the UVM's Red Tape

A Frustrated User's Survival Guide

Jonathan Bromley, Verilab Ltd, Oxford, England (*jonathan.bromley@verilab.com*)

***Abstract*—The Universal Verification Methodology (UVM) has brought extensive benefits to the field of functional verification using SystemVerilog. However, applying it to real projects can bring challenges and frustrations for novice and intermediate-level users. This paper examines typical examples of such challenges, and offers solutions that respect fundamental aims of the UVM: consistency, reusability, and expressive power.**

Examples include integration of directed tests or external models into the sequences mechanism, reconciling the abstract and untimed nature of sequences with the need for precise control over stimulus timing, proper use of the configuration or resource databases and when it is better not to use them, and working with a parameterized device under test.

While each individual example shown here may provide a useful resource, it is also hoped that the solutions taken as a whole will indicate criteria, objectives and techniques that must be considered when applying the UVM.

Keywords—functional verification; SystemVerilog; Universal Verification Methodology; UVM;

I. INTRODUCTION

A. *Contribution of the UVM*

The Universal Verification Methodology (UVM) [1] has had a major impact on industry practice in functional verification using SystemVerilog. It provides not only a library and toolkit of base classes and other code, but also a set of guidelines for their use. Taken together, this has encouraged consistency and interoperability across components from numerous verification IP suppliers, and has facilitated the development of an industry-wide portable skill set for engineers involved in developing and deploying functional verification environments.

B. *Users' learning experience*

Our anecdotal experience when supporting clients with application of the UVM is that users go through a series of stages in their learning experience.

At first the UVM seems large, complex and daunting. After some initial basic training or learning from suitable texts, however, it becomes clear that UVM offers much value, and it is not difficult to put together a verification component or environment using some straightforward guidelines and code patterns, finally adding custom code to implement problem-specific features.

Once an initial structure has been built, the focus turns to development of interesting verification scenarios and activity. At this stage the user – who is likely to have rich and deep prior experience of other verification technologies – begins to find some aspects of the UVM frustrating, with its comparatively rigid organization and strictly layered stimulus hierarchy getting in the way of what the user feels is their real work. Not all of these complaints are entirely directed at the UVM itself, but for a majority of users the UVM is their first experience of SystemVerilog verification using class-based (object-oriented or OOP) coding, and such users will see restrictions imposed by the core language to be conflated with those of the UVM.

C. *Target Readership*

The content of this paper is most likely to be useful to users who have a secure grasp of the basics of the UVM and a good understanding of OOP in SystemVerilog, and who are now beginning to take on more ambitious projects (for example, the construction of an in-house verification component). Such users are likely to have already encountered situations where the UVM's structures appear to obstruct rather than to facilitate their tasks, and where existing published guidance does not readily provide the help they need.

Even the most experienced UVM practitioners continue to discuss and refine their approaches to common verification concerns, and so we cannot reasonably expect users to be fully "up to speed" within the lifetime of their first UVM project. This paper aims to provide a useful resource for intermediate-level users who are

beginning to experience the frustrations outlined above, offering solutions to a selection of problems that have been encountered by the author in recent work. Inevitably the specific problems faced by any one user may not align with those presented here, but it is hoped that the general approach taken by these examples – and, perhaps, some of the specific techniques in them – may prove to be helpful in a wider context.

II. Examples and Solutions

- I want to use directed or modeled stimulus in a verification component that uses UVM sequences. How can this be done easily and efficiently without abusing the sequences paradigm?
- My agent has a config object, and it needs to pass that object to several subcomponents. Using the config_db looks like overkill for the subcomponents, and gets me into a mess with setting the target name/path because I need it to target multiple component objects. How can I tidy that up in a consistent way that everyone will understand?
- I have a SERDES interface with a parameterized number of lanes. But I don't want my virtual interfaces to be parameterized (well-known problem/limitation). How can I deal with that?

III. Passing Directed Stimulus Through a Sequence Stack

In a typical UVM active agent, the driver accepts sequence items from the sequencer, and performs the necessary pin-level activity to implement each sequence item in turn. For maximum flexibility, sequence items should normally be specified as the lowest-level individual transaction that makes any sense as a self-contained object. More elaborate stimulus is built from those items by composing them into sequences that are run on the sequencer. For example, in a SERDES protocol based on 8b10b encoding, the lowest-level meaningful item is a single 10-bit symbol.

The sequence item representing such a symbol needs to be randomizable. It also needs to be capable of introducing erroneous behaviour (e.g. not-in-table symbol, running-disparity error, shortened symbol to support the modelling of timing errors). An outline of the data in such a symbol is shown in Code Example III-1 below. A soft constraint is used to give the symbol the correct number of bits by default, while still making it easy for a specialized timing error sequence to introduce deterministic or randomized errors in symbol length.

```
class serdes_symbol extends uvm_sequence_item;
  rand bit [9:0] symbol_bits;
  rand int number_of_bits;
  rand bit timing_error;
  constraint symbol_10_bits {
    soft !timing_error;
    if (timing_error) {
      number_of_bits inside {[1:10]};
    } else {
      number_of_bits == 10;
    }
  }
  …
endclass
```

Code Example III-1: Sequence item for 10-bit symbol

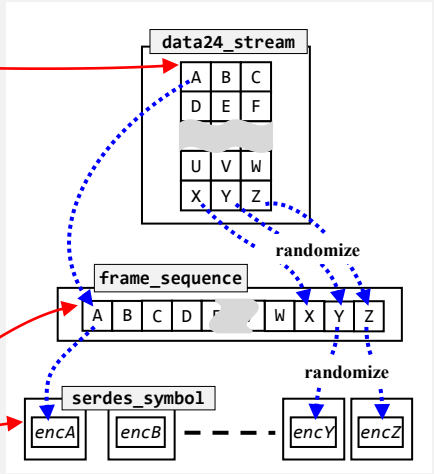
A higher-level sequence (or even a test) may run sequences on the agent's sequencer. Those sequences are ultimately decomposed into streams of sequence items – symbols, in this case – that are passed to the driver for execution. In a protocol UVC example taken from the author's recent experience, symbols were composed into "frames" of hundreds or even thousands of 8-bit data bytes that must then be encoded as 10-bit symbols. At a yet higher level, frames were populated from a stream of multi-byte data items according to a well-defined data packing algorithm. Test writers were primarily interested in this top-level stream of data items, which might be randomized or might be constructed from data stored in a file or generated by a model. Consequently, it was necessary to provide a sequence representing such a stream, for use by these test writers.

A. Sequence stack with randomization at each level

It is very common to see higher-level sequences composed using the `uvm_do` family of macros. These macros are convenient and expressive, and encapsulate several distinct steps that are required to launch a sub-sequence from within a higher-level sequence. However, there are some compelling disadvantages to using those macros in this situation. Consider the following possible implementation of the sequence stack described above:

```
class data24_stream extends uvm_sequence;
  rand bit [23:0] stream_data[];
  task body();
    frame_sequence fr_seq;
    `uvm_do_with( fr_seq, {
       frame_data.size() == 3*stream_data.size();
       foreach (stream_data[i]) {
         frame_data[3*i]   == stream_data[i][23:16];
         frame_data[3*i+1] == stream_data[i][15:8];
         frame_data[3*i+2] == stream_data[i][7:0];
       }
    } )
  endtask
  …
endclass

class frame_sequence extends uvm_sequence;
  rand bit [7:0] frame_data[];
  task body();
    serdes_symbol sym_item;
    foreach (frame_data[i]) begin
      bit [9:0] symbol_value = encode(frame_data[i]);
      `uvm_do_with( sym_item, {symbol_bits == local::symbol_value;} )
    end
  endtask
  …
endclass
```



Code Example III-2: Frame/stream sequence hierarchy

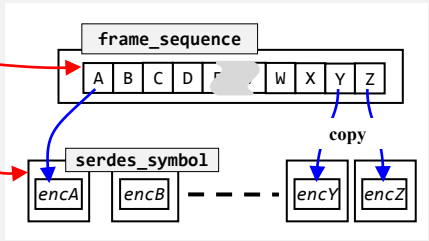
The `body()` method of `data24_stream` is an example of a situation where randomization is neither necessary nor helpful. Packing of stream data into the frame follows an algorithm that would be rather easy to describe in procedural code, but is clumsy to express in constraints. In the author's real-world application, the packing algorithm was significantly more complicated than this, and it was infeasible to use constraints at all. Instead it would have been necessary to build an intermediate array *exactly matching the desired frame data*, and then write a constraint that the frame sequence's `frame_data` array was identical to that array. Clearly this would have been both clumsy and extremely inefficient. Randomization of large arrays is expensive even when the constraints are as simple as in this case.

At the next lower level, in the `body()` method of `frame_sequence`, we can see that an additional randomization step is required *for each and every data symbol*. Once again this is absurdly inefficient when randomization is not required at all for the specific application. However, many users are reluctant to abandon `uvm_do_with` randomization, partly because it is built-in and therefore readily available, and partly because of fears that sidestepping randomization may lead to a less flexible sequence library.

B. Sequence stack without unnecessary randomization

The offending sequences can easily be rewritten as follows. First let us consider the frame sequence.

```
class frame_sequence extends uvm_sequence;
  rand bit [7:0] frame_data[];
  task body();
    serdes_symbol sym_item;
    foreach (frame_data[i]) begin
      `uvm_create( sym_item )
      sym_item.symbol_bits = encode(frame_data[i]);
      `uvm_send( sym_item)
    end
  endtask
  …
endclass
```



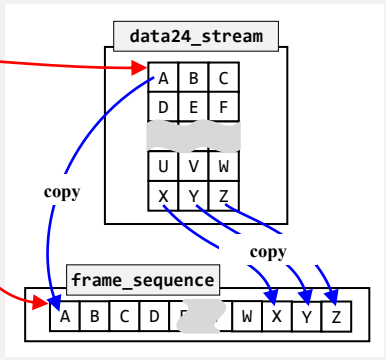
Code Example III-3: Frame sequence without randomization

As in Code Example III-2, the `frame_data` array is declared `rand`. This allows a higher-level sequence to randomize this sequence if required, but the random data will be converted into individual symbols in an efficient, deterministic manner. The code is hardly altered. It is merely necessary to split out the `uvm_do_with` macro into

its component parts, first constructing the sequence and attaching it to the same sequencer as its parent `frame_sequence`. Procedural code is then used to populate the item's data before using `uvm_send` to deliver it to the driver in the same way as for any other sequence. This simple modification not only makes the code clearer but also can save the runtime cost of many thousands of `randomize()` calls during a typical simulation run.

Next we consider the higher-level stream data sequence, which in its turn makes use of a frame sequence.

```
class data24_stream extends uvm_sequence;
  rand bit [23:0] stream_data[];
  task body();
    frame_sequence fr_seq;
    `uvm_create( fr_seq )
    fr_seq.frame_data = new[3*stream_data.size()];
    foreach (stream_data[i]) begin
      fr_seq.frame_data[3*i]   = stream_data[i][23:16];
      fr_seq.frame_data[3*i+1] = stream_data[i][15:8];
      fr_seq.frame_data[3*i+2] = stream_data[i][7:0];
    end
    `uvm_send( fr_seq )
  endtask
  …
endclass
```



Code Example III-4: Stream data sequence without randomization

Once again the sequence's data array is declared `rand`, leaving open the option to randomize it even though it then uses entirely deterministic code to send its randomized data to the driver. In this example the construction of the frame data has been done in a manner that obviously corresponds to the constraints in Code Example III-2, but other more elegant versions are possible (for example by using the `{>>{}}` streaming concatenation operator).

C. Benefits of a sequence stack without randomization

A test writer can now easily construct an instance of sequence `data24_stream`, populate its `stream_data` array from a file, by randomization or from any other suitable data source, and run it on the appropriate agent's sequencer. There is no loss of versatility or reusability, because each sequence in the stack uses `rand` data so that a different parent sequence can easily randomize its content. A small and easy-to-understand change in UVM macro usage has provided superior efficiency and greater clarity, without compromising key aims of the methodology.

IV. Deploying a Configuration Object

A. The UVM's configuration and resource databases

As is well known, the UVM's resource database (or its legacy support variant, the configuration database) makes it possible for any code to deposit useful values in a global repository. A classic, canonical use case is that the top level of a testbench prepares a configuration object, and then deposits that configuration object in the resource database with a resource name corresponding to the pathname of a component that will use it (as described in [2] and elsewhere). Because these databases are global and use string keys to locate entries, you can construct a resource item destined for a particular UVM object or component instance, and deposit it in the database before that instance has been constructed. It is not necessary to have a reference to a target object in order to deposit into the database a resource destined for that object. Furthermore, the lookup (retrieval) mechanism supports keys containing wildcard and regular expression patterns, so that they can match only part of a target object's full instance pathname. This supports two very important use cases:

- A single resource can target multiple objects, each of which can then retrieve the same resource.
- A resource can target an object without knowledge of the exact full pathname of that object – and, indeed, even without the target object having been created. This helps with vertical reuse where a piece of verification infrastructure can be reused in a larger testbench, appearing at a somewhat different location in the new testbench's instance hierarchy. It also allows configuration objects to be constructed by the top-level test runner module and "delivered" to their target components before those components have been brought into existence.

This flexibility comes at a price, however. The wildcard matching is quite costly of runtime, and there is also a risk that an excessively permissive wildcard pathname may cause the resource to be retrieved by target objects for which it is completely inappropriate.

B. The databases are commonly used to excess

The author's experience of many UVM environments at many different organizations is that the resource or configuration database is often used far more heavily than is necessary or desirable, potentially leading to the performance and precision difficulties mentioned at the end of the previous subsection. Within a well-understood hierarchy of testbench components, it is usually more convenient for parent components to deliver configuration values to their children by direct copy rather than by using the configuration database.

As we shall see below, it is rather easy to combine the best of both techniques. Configuration values can be distributed down a component hierarchy by simple copy, while still leaving open the option to override those values in individual components using the configuration database.

The remainder of this discussion is largely concerned with configuration *objects*. The author feels no need to apologize for this focus, as it is almost always desirable to encapsulate configuration into objects. Doing so brings many important benefits in addition to the obvious elegance of encapsulation:

- The object as a whole can be randomized, taking advantage of constraints describing important relationships among the individual configuration values.
- A configuration class can be extended. In particular, new constraints can be layered on those in the base class, a powerful and convenient way to create new test scenarios using an existing codebase.
- If an object *handle* is passed around, so that all components have references to the same single configuration object, then changes to that object's contents are immediately visible to all components (and can easily be notified by firing a `uvm_event` in the object).
- An object handle value of `null` unambiguously indicates that no configuration object has yet been provided.
- Classes are strongly typed, reducing the risk of a configuration being applied to the wrong target. By contrast, simple values (particularly those of integral type) are easily copied to the wrong variable.

C. Downward propagation of configuration objects

A UVM testbench's test runner module (in which an `initial` block calls the `run_test` method) has no access to components in the environment that will be created. Consequently, if it creates configuration objects, it must use the configuration database as a way to pass them to components in the UVM environment. However, this should be restricted to only the topmost component that needs the configuration object. Children of that topmost component (typically an environment or agent) should receive their copies of the configuration object handle by simple copy from their parent's `build_phase` method.

```
class my_agent extends uvm_agent;
  my_config cfg;
  my_monitor monitor;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    void'(uvm_config_db#(my_config)::get(this, "", " cfg", cfg));
    if (agt_cfg == null) `uvm_error(...)
    monitor = my_monitor::type_id::create(...);
    monitor.cfg = cfg;
    ...
```

copy the handle directly to a child component

Code Example IV-1: Parent component copies configuration object handle to its child

The monitor component's `cfg` data member has now been correctly set, after construction but before its `build_phase` method runs (recall that `build_phase` operates strictly top-down). This allows subcomponents to make use of the configuration object to control their own build activity, and allows the entry in the configuration database to be very specifically targeted to the parent component alone – there is no need to create a risky wildcard pathname that will encompass the parent and its children.

D. Allowing for exceptional override from the configuration database

Although the code pattern in Code Example IV-1 easily and correctly copies the configuration item to a child component, it is occasionally useful to be able to provide a different configuration to a specific child. For

example, a driver component might be given a configuration object that is different from its agent's in order to request the injection of certain errors. You can enable this child-specific targeting, from any upper level of the testbench, by providing the following code pattern in each component's `build_phase`. The example shows some added diagnostics; they are not strictly necessary, but are useful debug aids.

```
class my_monitor extends uvm_monitor;
  my_config cfg;
  bit cfg_override;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (uvm_config_db#(my_config)::get(this, "", " cfg", cfg)) begin
      `uvm_info("CFG_OVERRIDE", {"Config replaced by ", cfg.get_name()}, UVM_MEDIUM)
    end
    if (cfg == null) `uvm_error(...)
```

don't complain if not found in config DB (copy-from-above value is used)

fail if the final value is `null` for any reason

Code Example IV-2: Parent component copies configuration object to its child

This tiny variation on the commonly-seen pattern of "throw an error if there's nothing provided by the configuration database" is applicable to almost any component. It adds the option to provide configuration by copy from above. If that option is *not* used, then the traditional approach of providing everything from the configuration database remains possible. If it *is* used, it can nevertheless be overridden from the configuration database if required. Finally, if the parent does not provide a handle by copy, *and* there is no suitable entry in the configuration database, then this problem is detected and an error is signaled[1].

V. AVOIDING PARAMETERIZED VIRTUAL INTERFACES

Taking a virtual interface reference to a parameterized interface is problematic in SystemVerilog. However, when applying the UVM it is often necessary to work with DUT interfaces and subsystems that are inherently parameterized. With careful partitioning of the interface hierarchy, it is often possible to reconcile these conflicting concerns using a top-level interface that is parameterized to match the DUT, but contains sub-interface instances that are parameter-free and thus easy to reference from the UVM testbench.

A. Malign effects of parameterized virtual interfaces in a class-based testbench

An instance of a parameterized interface can only be referenced by a virtual interface whose data type has parameter specialization exactly matching that of the interface instance itself, as defined in [3] and shown below.

```
interface bus_if #(parameter DBITS=8);
  logic [DBITS-1:0] data;
endinterface
...
module TB_top;
  bus_if #(.DBITS(12)) bif_12 ();
  initial begin
    virtual bus_if vbi = bif_12;
    virtual bus_if#(12) vbi_12 = bif_12;
...
```

instance of parameterized interface

ILLEGAL: default specialization has parameter #(8), does not match the instance's data type

OK: virtual interface data type matches instance

Code Example V-1: Virtual interface parameterization rules

This typically becomes troublesome when a generic, reusable UVM verification component makes use of such an interface. Some UVM component class (such as a driver or monitor) must declare a virtual interface of the appropriate parameterized type. Parameters must be specialized using constant expressions, so the class must provide the virtual interface's declaration with appropriate parameter values in one of the following ways:

- The parameter can be specified as a simple literal in the virtual interface declaration, just as in Code Example V-1. Clearly this is too inflexible for a reusable component.
- The parameter's value can be provided from a Verilog `define macro, or from a parameter value specified in a package. This method allows for reuse in a different application, but it is unsatisfactory because every instance of the reusable component must adopt the same parameterization.

[1] It is worth noting that it is better to use `uvm_error` than `uvm_fatal` here. A `uvm_error` during `build_phase` automatically causes a *fatal* error before entering `run_phase`, but allows earlier phases to continue. Any further build errors, which would be hidden had `uvm_fatal` been used, can then be reported.

- The parameter can be derived from a parameter of the class itself. In this way, each instance of the class can potentially be given a different parameter specialization.

Only the last of these choices (virtual interface's parameter is obtained from a class parameter) is viable for a truly reusable verification component. However, experience shows that parameterizing a UVM component class is usually unsatisfactory. Any class that declares a variable or instance of that parameterized component class must itself provide the necessary parameter value, passing on the same problem to any object that instantiates it. Parameterization thus ripples out through the hierarchy of object instances. It also affects sequence classes, which must be parameterized to match the sequencer on which they will run. The result is an unmanageable plethora of parameterization throughout the testbench.

B. General solutions to the parameterization problem

Several possible solutions to this parameterization problem have been discussed in published material. Probably the most accessible, and also one of the most flexible solutions is the so-called "maximal footprint" approach in which the interface is not parameterized, but instead is given bit widths (or other parameterization) corresponding to the largest possible parameter value. Run-time masking and sign- or zero-extension can then be used to match DUT data to the oversized elements in the interface. Other viable solutions have been proposed, but (in the author's experience) have not gained wide acceptance in the UVM user community.

C. A common special case: parameterized lane count

A good example from the author's recent experience was a multi-lane SERDES protocol that had common reset and utility signals, but a parameterized number of lanes. In this case it is easy to see how to match UVM's requirement for parameter-free interfaces to the obvious RTL requirement for parameterized connection. An enclosing interface, parameterized for the number of lanes, contains a generated array of parameter-free interface instances with one instance for each lane. A corresponding array of virtual interface references is then constructed, with one element for each lane interface; there is no virtual interface reference to the parameterized outer enclosing interface.

Unfortunately, this attractively simple idea falls foul of a SystemVerilog language restriction. When referencing a member of an instance array, or an element of a generate-loop, the index (array subscript) must be a constant expression. It is therefore not easy to use procedural code to iterate over an array of this kind, as can be seen in the following example (which is illegal SystemVerilog code).

```
interface multi_lane_intf #(parameter N = 4); // with N lanes
  single_lane_intf lane_intfs [0:N-1] (...);
  ...
  virtual single_lane_intf lane_vif[N];
  for (int i=0; i<N; i++) begin
    lane_vif[i] = lane_intfs[i];
  end
endinterface
```

Array of interface instances

ILLEGAL:
variable `i` used to index into instance array `lane_intfs`

Code Example V-2: ILLEGAL example of access to elements of an instance array using a variable index

An alternative, legal solution uses a generate loop to populate the array of virtual interface references:

```
interface multi_lane_intf #(parameter N = 4); // with N lanes
  single_lane_intf lane_intfs [0:N-1] (...);

  virtual single_lane_intf lane_vif[N];
  for (genvar g=0; g<N; g++) begin : vif_loop
    initial lane_vif[g] = lane_intfs[g];
  end
endinterface
```

Array of interface instances

LEGAL:
genvar `i` is a constant expression

Code Example V-3: Generate loop used to initialize members of a virtual interface array

Once the virtual interface array has been populated in this way, it can of course be accessed using variable subscripts. However, this solution is flawed. The generated `initial` blocks execute at time zero in a nondeterministic order relative to other `initial` blocks, including the block that reads the virtual interface array and uses it to construct configuration database entries before executing UVM `run_test`. Consequently, the test runner could observe corrupt (typically `null`) values.

D. *Preferred solution using an initialization function*

A more satisfactory solution depends on the fact that declaration initializations are guaranteed to have completed before any `initial` block begins to execute. We initialize the virtual interface array by calling a function. Unfortunately, implementation of the function is not trivial. It cannot use a variable to index into the instance array. One possible implementation is as follows. Note that it uses only constant expressions to index into the instance array, but uses a comparison between a generate index and a variable to decide which element to return as it traverses the generated array of functions. No attempt has been made to optimize this function for performance, as it will be called only once at initialization time.

```
interface multi_lane_intf #(parameter N = 4); // with N lanes
  typedef virtual single_lane_intf vif_array_t[N];
  vif_array_t lane_vif = init_vifs();
  function vif_array_t init_vifs();
    for (int i=0; i<N; i++) init_vifs[i] = function_generator_loop[0].get_vif(i);
  endfunction
  for (genvar g=0; g<N; g++) begin : function_generator_loop
    function virtual single_lane_intf get_vif(int i);
      if (i==g) return lane_intfs[g];
      else      return function_generator_loop[(g+1)%N].get_vif(i);
    endfunction
  end
endinterface
```

This function executes before any `initial` block

If no match, pass request on to the next generate iteration

%N ensures all indices are within bounds

Code Example V-4: Implementation of the initialization function

VI. CONCLUSIONS

Examples have been used to illustrate the design and decision-making process that engineers need to undertake in order to solve some common usage problems in the UVM. It is hoped that these examples may indicate some simple strategies and code patterns that can be applied to common problems of UVM deployment. When solving such problems, it is important to keep in mind the essential aims of the methodology. In many real-world situations it is unfortunately easy to make design decisions that satisfy a project's immediate goals but which compromise the reusability and consistency that should be enabled by the UVM.

REFERENCES

[1] Accellera Systems Initiative, "UVM (Standard Universal Verification Methodology)," June 2014. [Online]. Available: http://www.accellera.org/downloads/standards/uvm. [Accessed April 2015].

[2] V. Cooper and P. Marriott, "Demystifying the UVM Configuration Database," in *DVCon*, San Jose, CA, 2014.

[3] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.

[4] J. Bromley and D. Rich, "Abstract BFMs Outshine Virtual Interfaces," in *DVCon*, San Jose, 2008.