Abstract- The main goal of functional verification in hardware design is to find out the bugs in design description given by design engineers and to check the functionality of design whether the output matches with desired value or not and then customize the design to get the desired functionality of DUT. Various verification techniques have been developed from past few years to make the verification process much easier and user friendly. This paper presents a recent approach to using UVM, the Universal Verification Methodology, for functional verification by mainstream users. The goal is to identify a minimal set of concepts sufficient for constrained random coverage-driven verification in order to ease the learning experience for engineers coming from a hardware design background who do not have extensive object-oriented programming skills. We describe coding guidelines to address the canonical structure of UVM component and transaction, the construction of the UVM component hierarchy, the interface with the design-under-test, the use of sequences, and the use of the factory and configuration mechanisms. We also provide some key factors which allowed the user to migrate from OVM to UVM for verification purpose.

Keywords- SystemVerilog, OVM, UVM, Functional verification.

I. INTRODUCTION

Today’s traditional design flow involves design at multiple levels of abstraction. As the design implementation is refined and the verification vehicle changes, the test bench needs to adjust to abstractions from transaction-level simulation, and RTL simulation to hardware acceleration. But the challenge is that separate verification components, test benches, tests, and plans are developed by separate teams at different abstraction levels. This requires more expertise to learn and understand, more code to develop and maintain, and larger teams to fund and manage.

This paper provides an overview to use Universal Verification Methodology (UVM) for functional verification by mainstream users as opposed to highly skilled verification specialists. It arises from SystemVerilog and UVM provide mechanisms to create verification components for checking, coverage collection, and stimulus generation, and to modify the behaviour of those components for specific tests. But SystemVerilog and UVM provide more than this, so much more in fact that the learning curve can be daunting for non-specialists.

The goal of this paper is to enable the user with experience in Verilog or VHDL to become productive in UVM by learning a small number of new coding idioms, selected to minimize the conceptual clutter they have to deal with. As users become fluent with this set of basic idioms, they can then branch out to embrace the full feature set of UVM as and when they need. We describe coding guidelines to address the canonical structure of a UVM component and a UVM transaction, the construction of the UVM component hierarchy, the interface with the design-under-test, the use of UVM sequences, and the use of the factory. Starting from these simple guidelines, user can create constrained random verification environments in an object-oriented coding style that are fully compliant with the UVM standard, and hence are interoperable with UVM verification IP from other sources.

II. SYSTEMVERILOG AND UVM

Over the past few years, constrained random coverage-driven verification has been increasingly adopted as the methodology-of-choice for simulation-based functional verification to the point where it is widely used on the largest ASIC projects. SystemVerilog, as the only industry standard hardware verification language supported every one of the three largest EDA vendors, has displaced its rival single-vendor solutions in many companies.

But SystemVerilog is not without its problems. Although current SystemVerilog implementations are in many ways both mature and robust, SystemVerilog remains under-specified as a language.

SystemVerilog was an extremely ambitious standardization project that was undertaken prior to the development of any complete proof-of-concept implementation, and as a result the IEEE 1800 SystemVerilog Language Reference Manual still has many areas of ambiguity, and more than one of the major implementations has significant gaps. The simulation tool vendors themselves are in the unenviable position of having to invest significant engineering resources in trying to pin down and implement a very complex language against an
ambiguous definition, and so take the very reasonable position of prioritizing their implementation choices according to customer demand.

Nevertheless, certain areas of the SystemVerilog language do stand out as being well-defined and consistently implemented across all the major simulators. These include:

- The concise RTL features such as ANSI-style port and parameter declarations, always_comb, always_ff, unique, priority, and the abbreviated port connection syntax
- C-like control constructs such as for, foreach, do-while, break, continue, return
- C-like data type features such as typedef, enum, struct, and the 2-valued integer types
- VHDL-like package and import features
- Classes and the features for constraints and functional coverage based thereon
- The class-based data types, namely strings, queues, dynamic arrays and associative arrays
- Interfaces and virtual interfaces sufficient for communication between classes and modules

One reason that the features from the above list have been implemented so thoroughly and consistently is that, with the exception of the concise RTL features, they have been widely used to create the libraries of base classes that underpin the functional verification methodologies AVM, URM, VMM, OVM, and now UVM. Customer demand has pushed the EDA vendors to support each others' methodology class libraries, which in turn has driven the implementations to converge on a common understanding of the features and semantics of class-based SystemVerilog.

SystemVerilog and UVM now form a virtuous circle. The class-based SystemVerilog features that support constrained random verification are sufficiently well-defined and well-implemented to allow the development of robust and portable verification class libraries, and the widespread use of those libraries ensures the ongoing support of the necessary language features by the tool vendors. In addition to the Verilog-like and C-like features of SystemVerilog, we make use of classes, constraints, covergroups, packages, interfaces and virtual interfaces. UVM makes heavy use of type parameterization to classes, and fortunately all of the major simulator implementations now agree on the semantics in this area.

III. OBJECT ORIENTED CONCEPT

Object-oriented (OO) or aspect-oriented programming concepts are key to contemporary constrained random verification methodology because they enable reuse, yet these techniques are amongst the hardest to learn. OO techniques allow verification components to be specialized to the needs of a specific test bench or test without modifying their source code and enables well-structured communication between those components using function calls. We wanted to provide some of the expressive power of the OO paradigm without getting drawn into the full set of issues involved in OO programming. In UVM, the class is used as a container to represent components, transactions, sequences, tests, and configurations. Let's take the component. Unlike the VHDL design entity or the SystemVerilog module, a component represents an abstraction across a whole family of possible structural building blocks. A component picks out what is common across several such building blocks, but a component is not concrete, meaning that it is not the final once-and-for-all definition of the thing. A VHDL design entity or a SystemVerilog module can describe a family of related components, but only if the variants are anticipated in advance and are explicitly captured in the source code by means of language features such as generic parameters and generate statements. Because it is a class, a UVM component can be extended after-the-fact in arbitrary ways. An extension can add new features or can modify existing features. In particular, we require this extension capability so that a test can extend a transaction or a sequence in order to add constraints, and then use the factory mechanism to override the generation of those transactions or sequences.

IV. UVM CONCEPTS

The goal of this paper is to identify a minimal set of concepts sufficient for constrained random coverage-driven verification in order to ease the learning experience for engineers coming from a hardware design background who do not have extensive object-oriented programming skills. At the same time, we did not want to strip down the conceptual framework to the point where it lost all the expressive power of the object-oriented paradigm. Some other attempts to present verification class libraries to hardware designers have ended up doing little more than represent the semantics of VHDL or Verilog using classes, which was a pitfall we wished to avoid. In our experience, hardware designers moving to a new language for verification, SystemVerilog in this case, do indeed want to benefit from the increased expressive power and flexibility afforded by a new paradigm.

Our conceptual vocabulary is listed below. These terms are elaborated later in the paper, and expanded definitions can be found in the documentation accompanying the UVM distribution.

- Component – a structural building block, conceptually equivalent to a Verilog module
• Transaction – a bundle of data items, which may be distributed over time and space in the system, and which form a communication abstraction such as a handshake, bus cycle, or data packet
• Sequence – an ordered collection of transactions or of other sequences
• Phase – execution is subdivided into various predefined phases that permit components to agree on when to build components, connect ports, run simulation, and so forth
• Factory – a function call that returns a component, transaction, or sequence, the type of which may be overridden from a test
• Port and export – connection points for transaction-level communication between components
• Generation – the creation of components, transactions, or sequences, where the properties of each may be set deterministically or at random under the control of constraints
• Test – a top-level component, which drives generation
• Configuration – an object associated with a component which may be set or randomized by a test and which it is used to configure that component as the component hierarchy is built
• Sequencer – a component that runs sequences and that sends transactions generated by those sequences downstream to another sequencer or to a driver
• Driver – a component that receives transactions from a sequencer and that drives the signal-level interface of the Design Under Test (DUT)
• Monitor – a component that senses the signal-level interface of the DUT and that sends transactions to the verification environment
• Coverage – functional coverage information can be collected using SystemVerilog cover groups within a component
• Checking – functional correctness of the DUT can be checked using either procedural code within a component or SystemVerilog assertions within an interface

A. Components
Components are used to build a component hierarchy, conceptually very similar to the design hierarchy in VHDL or the module hierarchy in Verilog. In this case the component hierarchy is part of the verification environment rather than the design, and components represent stimulus generators, drivers, monitors, coverage collectors, and checkers. The component represents the reusable unit of the verification environment, so has a standard structure and conventions for how it can be customized. Components are created quasi-statically at the start of simulation.

Here is a skeleton component. The uvm_component_utils macro and the function new should be treated as boilerplate code and written exactly as shown.

Class my_comp extends uvm_component;
`uvm_component_utils(my_comp)
function new(string name, uvm_component parent);
super.new(name, parent);
endfunction
...
Endclass

B. Transactions
Transactions are the basic data objects that are passed between components. In contrast to VHDL signals and Verilog wires, transactions represent communication at an abstract level. In order to drive stimulus into the DUT, a so-called driver component converts transactions into pin wiggles, while a so-called monitor component performs the reverse operation, converting pin wiggles into transactions.

Here is a skeleton transaction. Again, the uvm_object_utils macro and function new should be treated as boilerplates.

class my_tx extends uvm_sequence_item;
`uvm_object_utils(my_tx)
function new(string name = "");
super.new(name);
endfunction
...
Endclass

C. Phase
Every component implements the same set of phases, which are run in a predefined order during simulation in order to synchronize the behaviour of the components. When compared with VHDL or Verilog, UVM provides rather more temporal structure within a simulation run. The standard phases are as follows:

1. build – create child component instances
2. connect – connect ports to exports on the child components
3. End of elaboration – housekeeping
4. Start of simulation – housekeeping
5. run – runs simulation
6. extract – post-processing
7. check – post-processing
8. report – post-processing

Each phase is represented by a function within the component, except for run, which is a task because it alone consumes simulation time. If a function is absent, that component will be inactive in the given phase.
As you can infer from the above list, the primary distinction amongst the phases is between the phases for building the component hierarchy, connecting the ports, and running simulation, with additional housekeeping phases pre-pended and appended to the simulation phase.

D. Sequences
Sequences are assembled from transactions and are used to build realistic sets of stimuli. A sequence could generate a specific pre-determined set of transactions, a set of randomized transactions, or anything in between. Sequences can run other sequences, possibly selecting which sequence to run at random. Sequences can be layered such that higher-level sequences send transactions to lower-level sequences in a protocol stack.

Here is a skeleton sequence. It is similar to a transaction in outline, but the base class uvm_sequence is parameterized with the type of the transaction of which the sequence is composed. Also every sequence contains a body task, which when it executes generates those transactions or runs other sequences.

```verilog
class my_seq extends uvm_sequence #(my_tx);
    `uvm_object_utils(my_seq);
    function new (string name = "");
        super.new(name);
    endfunction
    task body;
        ...
    endtask
endclass
```

Transactions and sequences together represent the domain of dynamic data within the verification environment.

E. Factory
The UVM factory mechanism is an implementation of the so-called factory pattern described in the OO literature. The UVM factory can make components, transactions, and sequences. Use of the factory enables the choice of object type to be overridden from the test, although a given component, transaction or sequence can only be overridden with one that extends the class of the original. This is one of the main mechanisms by which a reusable verification component can be customized to the current environment.

Here is an example of a component creating child components during the build phase:

```verilog
class A extends uvm_component;
    `uvm_component_utils(A)
B b; // Child component
C c; // Child component
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
function void build; // Build phase
    super.build();
    // Factory calls to create child components
    b = B::type_id::create("b", this);
    c = C::type_id::create("c", this);
endfunction

Endclass
```

F. Port and export
Ports and exports are analogous to ports in VHDL or Verilog, but are used for transaction-level communication rather than signal-level communication. A component can send out a transaction through a port, or receive an incoming transaction through an export. Transactions are passed as arguments to function calls, which may be non-blocking (return immediately) or blocking (suspend and wait for some event before returning), which is sufficient for basic synchronization within the verification environment. All detailed timing information should be pushed down into the driver and monitor components that connect to the DUT so that the timing can be determined by the DUT interface, which is typically locked to low-level clocks and other synchronization signals. Within the verification environment, control flow radiates outward from the DUT, with drivers calling get to request transactions from sequencers when they are ready for further stimulus, and monitors calling write to distribute transactions around the verification environment for analysis.

A call to get from a driver may block if the stimulus generator is coordinating its activities with some other part of the verification environment. A call to write from a monitor is not permitted to block, because the DUT cannot be stalled waiting for analysis activity.

The example below shows a component A containing two child components B and C. The connect function connects p_port of component B to q_export of component C.

```verilog
class A extends uvm_component;
    `uvm_component_utils(A)
B b; // Child component having p_port
C c; // Child component having q_export
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    function void build; // Build phase
        super.build();
        b = B::type_id::create("b", this);
        c = C::type_id::create("c", this);
        b.p_port.connect( c.q_export );
    endfunction

Endclass
```
G. Generation
Generation exploits SystemVerilog randomization and constraints. Component generation occurs quasi-statically during the so-called build phase of UVM, when a component is able to access its configuration object (if there is one) in order to control the generation of lower-level components. This is analogous to the elaboration phase in VHDL or Verilog. Sequence generation occurs dynamically. Control over the precise sequence of transaction that finally arrives at the DUT can be distributed across pre-defined sequence generation components and the test, which is able to extend and constrain existing sequences.

Here is an example showing transaction generation within the body task of a sequence:

class my_seq extends uvm_sequence #(my_tx);
...
task body;
my_tx tx;
tx = my_tx::type_id::create("tx");
start_item(tx);
assert( tx.randomize() with { cmd == 0; } );
finish_item(tx);
...
endtask
endclass

In the example above, the transaction is being constrained as it is randomized in order to set the value of the command field to a specific value before sending the transaction downstream. The start_item and finish_item functions synchronize with the component that is pulling transactions from the sequencer, which could be a driver or another sequencer. start_item waits for the downstream component to request the transaction, finish_item waits for the downstream component to indicate that it has finished with the transaction. For its part, the downstream component calls get to fetch the transaction and may call put if it needs to send back a response.

H. Sequencer
A sequencer is a variety of component that runs sequences and sends them downstream to drivers or to other sequencers. At its simplest a sequencer looks like any other component, except that it has an implicit transaction-level export for connection to a driver.

We can now get a little more ambitious and show an example including sequence, sequencer, and launching the sequence from the test:

// A SEQUENCER is a component
class my_sqr extends uvm_sequencer #(my_tx);
`uvm_component_utils(my_sqr)
function new(string name = "");
super.new(name, parent);
endfunction
task body;
my_tx tx;
tx = my_tx::type_id::create("tx");
start_item(tx);
assert( tx.randomize() );
finish_item(tx);
endtask
endclass

class my_test extends uvm_test;
`uvm_component_utils(my_test)
...
my_env env;
...
task run;
my_seq seq;
// Create the sequence
seq = my_seq::type_id::create("seq");
// randomize it
assert( seq.randomize() );
// and start it on the sequencer
seq.start( env.agent.sqr );
endtask
endclass

A sequencer can also receive transactions from other sequencers. Here is an example of one that does just that:

class ano_sqr extends uvm_sequencer #(ano_tx);
`uvm_component_utils(ano_sqr)
uvm_seq_item_pull_port #(my_tx) seq_item_port;
...
function void build;
seq_item_port = new("seq_item_port", this);
endfunction
endclass

Conceptually, all that is happening here is that a sequence, running on a sequencer, is pulling in transactions through a port on that sequencer. In order to do so, it needs direct access to the sequencer object that it is running on, which is provided by the predefined variable p_sequencer. Through p_sequencer, a sequence can refer to variables declared within the sequencer class, including ports and references to other external components. It turns out that this coding trick is all that is needed in order to have a sequence start child sequences on another sequencer, that is, on a sequencer other than the one it is itself running on. Such a sequence is called a virtual sequence because it does not itself generate transactions but instead controls the execution of other sequences, running on other sequencers.
I. Monitor
A monitor is a variety of component that is confined to having passive access to the signal-level interface of the DUT. The monitor monitors traffic going to and from the DUT from which it assembles transactions which are distributed to the rest of the verification environment through one or more analysis ports.

All of the necessary concepts have already been discussed above. Here is an example:

```verilog
class my_monitor extends uvm_monitor;
```

```verilog`
'`uvm_component_utils(my_monitor)

```verilog`
virtul dut_if dut_vi;
```

```verilog`
... task run;
begin
my_tx tx;
// Sense the DUT pins on a clock edge
@(posedge dut_vi.clock);

```verilog`
tx = my_tx::type_id::create("tx");
tx.cmd = dut_vi.cmd;
tx.addr = dut_vi.addr;
tx.data = dut_vi.data;
aport.write(tx);
end
endtask
endclass
```

J. Driver
A driver is a variety of component that always sits downstream of a sequencer. The driver pulls transactions from its sequencer and controls the signal-level interface to the DUT. The transaction-level interface between the sequencer and the driver is a fixed feature of UVM, and is unusual in the sense that both the port and the export required for TL-communication are implicit.

```verilog
class my_driver extends uvm_driver #(my_tx);
```

```verilog`
'`uvm_component_utils(my_driver)

```verilog`
virtul dut_if dut_vi;
```

```verilog`
function new(string name, uvm_component parent);
begin
uvm_object obj;
my_config config;
get_config_object("config", obj, 0);
assert( Scast(config, obj) );
dut_vi = config.dut_vi;
end
endfunction
```

```verilog`
task run;
begin
my_tx tx;
// Wiggle pins of DUT
dut_vi.cmd = tx.cmd;
dut_vi.addr = tx.addr;
dut_vi.data = tx.data;
end
endclass
```

In the example above, the driver communicates with the sequencer by means of the call `get(tx)` through the implicit `seq_item_port`. Having got a transaction, it then drives the signal-level interface to the DUT by making assignments to the members of a SystemVerilog interface, which is done through the virtual interface `dut_vi`. The virtual interface is the SystemVerilog language mechanism that is used to pass data between structural Verilog modules and the class-based verification environment.

The example above all shows how a configuration object can be used to pass the virtual interface down to the driver during the build phase.

K. Transaction operations
It is common to need to perform operations on transactions, operations such as printing out the contents of a transaction, making a copy of a transaction, or comparing two transactions for equivalence. For example, the subscriber described above may wish to log the data from a transaction or compare a transaction from the DUT with another transaction representing the expected behaviour. UVM provides a standard set of functions for purposes such as these, as illustrated below:

```verilog
Function void write(my_tx t);
```

```verilog`
my_tx tx;

```verilog`
tx.copy(t)
history.push_back(tx);
if ( !t.compare(expected))

```verilog`
uvm_report_error("mismatch",
$tformatf("Bad transaction = %s",
t.convert2string()));

```verilog`
endfunction
```

Firstly, the `copy` function takes a complete copy of the transaction passed as an argument, in this case storing the copy in a queue of past transactions. Secondly, the `compare` function compares two different transactions for equivalence. Finally, the `convert2string` function returns a string representing the contents of the transaction in printable format. The `uvm_report_error` function is one of several standard utility for message reporting.

There is more to a transaction than meets the eye. As well as containing data fields representing properties of the protocol being modeled, a transaction object...
may contain housekeeping information such as timestamps, logs, and diagnostics. This secondary information typically needs to be treated differently when performing copy, compare, or convert2string operations, and such differences need to be accounted for by the user when declaring transaction classes. For example:

```verbatim
class my_tx extends uvm_sequence_item;
    `uvm_object_utils(my_tx)
    rand bit cmd;
    rand int addr;
    rand int data;
    ...
    function string convert2string;
        return $sformatf(...);
    endfunction
    function void do_copy(uvm_object rhs);
        my_tx rhs_;
        super.do_copy(rhs);
        $cast(rhs_, rhs);
        cmd = rhs_.cmd;
        addr = rhs_.addr;
        data = rhs_.data;
    endfunction
    function bit do_compare(uvm_object rhs,
                            uvm_comparer comparer);
        my_tx rhs_;
        bit status = 1;
        status &= super.do_compare(rhs, comparer);
        $cast(rhs_, rhs);
        status &= comparer.compare_field("cmd", cmd, rhs_.cmd, $bits(cmd));
        status &= comparer.compare_field("addr", addr, rhs_.addr, $bits(addr));
        status &= comparer.compare_field("data", data, rhs_.data, $bits(data));
        return(status);
    endfunction
endclass
```

Note that the behavior of copy and compare are overridden by providing functions named do_copy and do_compare, respectively, as part of the transaction class. Each function should exclude any housekeeping or diagnostic fields.

**CONCLUSION**

UVM is a rich and capable class library that has evolved over several years from much experience with real verification projects large and small, and SystemVerilog itself is a large and complex language. As a result, although UVM offers a lot of powerful features for verification experts, it can present a daunting challenge to Verilog and VHDL designers who want to start benefiting from test bench reuse. The guidelines presented in this paper aim to ease the transition from HDL to UVM.

**REFERENCES**

    http://dx.doi.org/10.1109/IEEESTD.2009.5354441

    http://www.accellera.org/activities/vip

