UVM Express
from the UVM/OVM Online Methodology Cookbook

[NOTE: This article is extracted from the UVM/OVM Online Methodology Cookbook found on verificationacademy.com. This is the Overview article, which introduces UVM Express. Other articles are also included in the Cookbook that go into more detail on each of the steps in applying UVM Express to your verification environment.]

UVM Express is a collection of techniques, coding styles and UVM usages that are designed to increase the productivity of functional verification. The techniques include raising the abstraction level of tests, writing tests using BFM function and task calls, adding functional coverage, and adding constrained-random stimulus generation.

The UVM is a class library which provides a view of the SystemVerilog verification features. The UVM class library contains powerful constructs for verification, including a phasing machine, a reporting system, a configuration database, a factory for overriding types and the ability to write tests as sequences (or function calls). Some verification teams are successfully adopting the UVM from the beginning, but those teams usually have a background in e, Vera, VMM, AVM or OVM.

In order to be proficient with SystemVerilog and the UVM, an investment of time and money needs to be made in training. Often the adoption process can be made shorter by hiring consultants.

Most verification teams, however have time and budget restrictions and cannot adopt the UVM in whole or adopt it as quickly as they might like. These teams are usually under-staffed, under-funded and over-worked. They are exactly the kind of people that the UVM is meant to help, but the first step towards adoption is too high.

UVM Express provides a small first step toward UVM adoption. UVM Express is a way to build your testbench environment, a way to raise your abstraction level, a way to check the quality of your tests and a way to think about writing your tests. Each of the steps outlined for UVM Express is a reusable piece of verification infrastructure. These UVM Express steps are a way to progressively adopt a UVM methodology, while getting verification productivity and verification results at each step.

Using UVM Express is not a replacement for full UVM, but instead enables full UVM migration or co-existence at any time. UVM Express is UVM – just organized in a way that allows progressive adoption and a value proposition with each step.

UVM EXPRESS: WRITING TESTS USING TASKS IN A BFM

The first level of UVM Express is a BFM (Bus Functional Model). The BFM has a job to manage the signals, and to supply an interface for the testbench to access the signals.

For example, a BFM might contain all the wires of an ‘abc’ bus, and a task called ‘read(address, data)’ which when called would issue a read on the bus, and return the data read from the address (see the code snippet on the following page).

This ‘read’ task consumes time. All tests are written in terms of the BFM and the helper functions that it supplies. No direct pin wiggling is allowed.
The BFM is a Bus Functional Model. It is the way that the test interacts with the signals. The BFM is implemented as a SystemVerilog interface that has signals (wires) and tasks to manipulate the signals.

The tasks in the BFM represent the kinds of transactions that the interface supports; for example, read(), write(), burst_read() and burst_write(). In order to create tests, the BFM tasks are called, but signals are never directly accessed outside the BFM.

**UVM EXPRESS: FUNCTIONAL COVERAGE**

The second level of UVM Express adds a coverage agent to the BFM based tests. The coverage agent sits on the bus collecting coverage, creating transactions and publishing them to subscribers. A typical subscriber is a coverage collector. Each published transaction gets counted by the covergroups in the coverage agent. Usually the coverage collector is written as a UVM Agent, and placed in a package. A user of this UVM agent does not need to interact directly with UVM, nor do they need to understand how the agent is built. They only need to understand any configuration options that the UVM agent offers. The UVM Agent is simply created and connected to the bus.

```
interface abc_if(input wire CLK);
  reg RST;
  reg VALID;
  reg READY;
  reg RW;
  reg BURST;
  reg [31:0] ADDR;
  reg [31:0] DATA;
  wire [31:0] DATAO;
  task reset();
  task read( bit[31:0] addr, output bit[31:0] data);
  task write( bit[31:0] addr, input  bit[31:0] data);
  task burst_read( bit[31:0] addr, output bit[31:0] data[4]);
  task burst_write( bit[31:0] addr, input  bit[31:0] data[4]);
  task monitor(output bit transaction_ok,
              output bit rw,
              output bit[31:0] addr,
              output bit[31:0] data[4],
              output bit burst);
endinterface
```
from the lowest address and the highest address,
and a few addresses in between. Each read should
produce correct values. A read to an illegal address
should be handled correctly. Each of these inputs (READ
addresses) and outputs (values READ) are counted,
and can be used to describe correct functionality.

**UVM EXPRESS: CONSTRAINED RANDOM STIMULUS**

The third level of UVM Express replaces the user directed
test with constrained-random stimulus. User directed tests
can be combined with constrained-random stimulus, but
this third level of UVM Express doesn’t attempt this. This
level builds a UVM agent which can generate stimulus and
collect coverage. This UVM agent is used in the same way
as the UVM agent from the second level – it is created and
connected. Once it is created, connected, configured and
started, it causes transactions to occur on the bus – it is
a traffic generator. Transactions are generated from the
built-in randomization contained in the UVM agent – usually
sequences and sequence items with constraints. A user of
this UVM agent does not need to interact directly with the
UVM or the UVM agent. The UVM agent is a self-contained
stimulus generation tool. The user must understand what
configuration options are available, but does not need to
understand low level UVM details.

Constrained random stimulus is stimulus that has
been described by using SystemVerilog constraints.
The constraints are randomized to produce a new collection
of stimulus. Each randomization begins with a seed value.
A series of stimulus can start with a specific seed, and
progress. Once the stimulus progression is done, a different
constraint can be used with the same or different seeds
to produce a different progression of stimulus.