UVM Sans UVM

An approach to automating UVM testbench writing

Rich Edelman
Mentor Graphics
Fremont, CA

Shashi Bhutada
Mentor Graphics
Los Angeles, CA

I. INTRODUCTION

The SystemVerilog [1] UVM [2] promises to improve verification productivity while enabling teams to share tests and testbenches between projects and divisions. This promise can be achieved; the UVM is a powerful methodology for using constrained randomization to reach higher functional coverage goals and to explore combinations of tests without having to write each one individually. Unfortunately the UVM promise can be hard to reach without training, practice and some significant expertise.

There are many organizations that offer "UVM training", "Simple UVM", "Easy UVM", and "Easier UVM". A google search will reveal many places to find easy UVM and get help. Those proponents of easy UVM supply training, support, and the actual doing of the work. But there is a cost in time and money. A google search will also reveal many places that supply code generators – "UVM templates", "UVM generator". There are also frameworks layered on top of the UVM library – for example, "UVM Framework". Our belief is that a framework or code generated template system is the right solution to get started with the UVM. The frameworks, code generators and templates leverage the UVM experience of previous verification efforts. Additionally, success with the generated code doesn't require knowing all the details at the start. Over time the framework, generator and template can grow as the UVM verification experience on the team grows.

This paper is targeted towards a verification team without any UVM background and it describes a simple template and generator, without any extra time overhead; and without any extra budget overhead. This paper offers a simple solution to taking that first step. The generator and templates do require SystemVerilog knowledge, and a general understanding of what a UVM verification environment looks like. There will be typing – but it is limited to a few well defined sections of code that are design specific – very well known to the verification and design teams building a UVM testbench. With a few simple edits the generated code can be used on the same day. Our experience deploying the templates at customers is that within 3 to 8 hours, a new UVM based testbench is up and running.

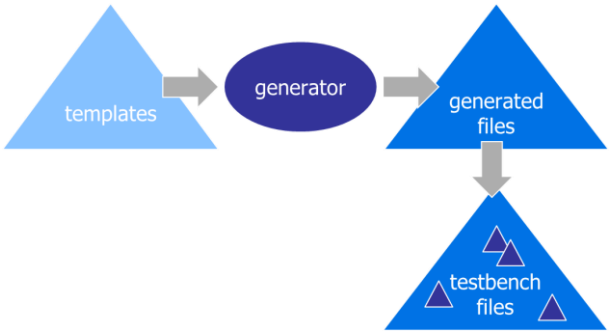

Figure 1: Idealized generator

The templates are not automatic – in the sense that you still have to do some work. You must describe your transaction. You must describe your bus. You must design the driver of the bus and the monitor of the bus. You may need to write a "golden model". You have to design how the testing will occur – for example, how to test multiple interfaces. All of the work that you must do occurs within the generated boilerplate. You must be able to write SystemVerilog code, but you don't need to know the details of the UVM.

II. A Typical UVM Testbench

A typical UVM test bench has a device-under-test (DUT), and an "agent" for each interface, an "environment" collecting agents together, and a top level "test". The interfaces to the DUT are SystemVerilog interfaces – and the virtual interface is used to connect the DUT to the class-based testbench. Don't worry – the templates will take care of most of this detail and vocabulary.

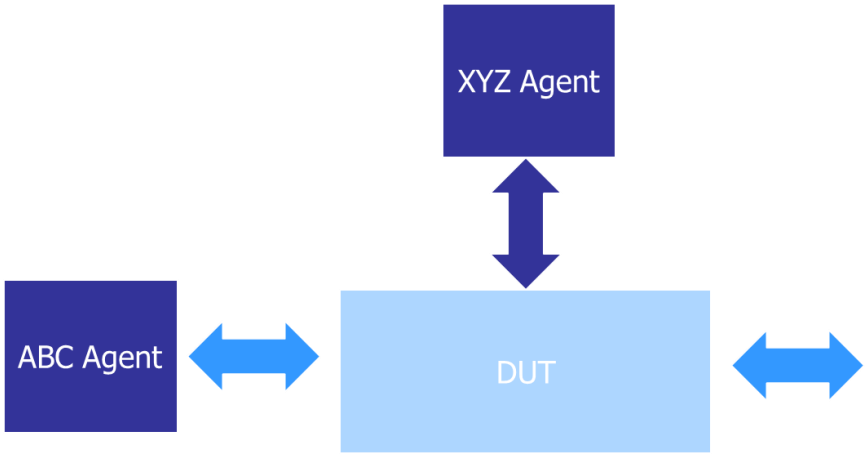

Figure 2: Simple DUT with Three Interfaces (one unconnected)

III. The Templates

Two sets of templates have been developed, each with a slightly different focus. The paper will discuss one of the templates only. The other template is available from the authors. There is no right template. Each template will reflect the priorities of the template writers (edit-in-place vs. edit-side-files), or the focus of the verification effort

(cover-all-address-ranges vs. cover-address-ranges-edges) or the general verification philosophy (read-and-write-all-memory).

The template system itself is a stand-alone runnable – meaning that the template can be compiled and simulated, and will do something. There is no DUT connected, and no useful verification will happen, but by being a compile-able and simulate-able system, it is easy to create a template that has fewer bugs to begin with. There is no pseudo SystemVerilog code involved. It is all SystemVerilog UVM.

The templates for this paper consist of about 20 files that have a total of 500 lines. The idea is that the template should be trivially simple. The generator script is below.

```
#!/bin/csh -f

mkdir -p generated

foreach agent ( ABC XYZ )

  # Create a new file tree. (ala cp -r a b)
  tar cf - \
    -C uvm-templates/agents \
           --transform=s/MY_AGENT_/${agent}_/g \
                  MY_AGENT_agent \
  | tar xvf - \
    -C generated

  # Apply 'sed'
  pushd generated/${agent}_agent

  foreach file ( `find . -type f ` )
    sed -e s/MY_AGENT_/ABC_/g < $file > $file.NEW
    mv $file.NEW $file
  end

  popd
end
```

It uses the ‘tar’ command to create a new hierarchy for the agent. Then it processes the contents of each file in the new hierarchy – using a ‘sed’ filter to change the generic template names to specific agent names. For example the generic sequencer:

```
class MY_AGENT_sequencer extends uvm_sequencer#(MY_AGENT_seq_item);
  `uvm_component_utils(MY_AGENT_sequencer)
  ...
endclass: MY_AGENT_sequencer
```

is changed to the specific code:

```
class ABC_sequencer extends uvm_sequencer#(ABC_seq_item);
  `uvm_component_utils(ABC_sequencer)
  ...
endclass: ABC_sequencer
```

IV. THE TRANSACTION

The transaction is an abstraction that describes how you want to communicate with your DUT. For example, you can have read/write transactions, or register transactions, or very simple transactions that initially represent the bus signals directly and can later be abstracted as needed.

A simple read/write transaction with address and data is below. This transaction can transfer a burst of 4 data words.

```
class ABC_seq_item extends uvm_sequence_item;
  `uvm_object_utils(ABC_seq_item)

  rand bit burst;
  rand bit rw;
  rand bit[31:0] addr;
  rand bit[31:0] data[4];
  …
endclass
```

V. THE BUS

Each of the agents generated is attached to a SystemVerilog virtual interface – a collection of signals. The template also has a second interface – it is a BFM with helper functions to drive and monitor the bus. These helper functions are not a requirement of a UVM testbench, but allow for better separation of the UVM testbench and the RTL pin wiggling required on the bus.

The Collection of Signals

```
interface ABC_if ( input CLK );

  logic RST;

  logic VALID; // Goes high when the addr/data is valid.
  logic READY; // Goes high when the DUT is ready.

  logic RW;    // READ = 1, WRITE = 0
  logic BURST; // High when a burst begins, low when it ends.

  logic [31:0] ADDR;

  logic [31:0] DATAI;
  wire[31:0] DATAO;

endinterface
```

The BFM

```
interface automatic ABC_bfm ( ABC_if bus );

  task automatic read(bit[31:0] addr, output bit[31:0] data);
    bus.RW <= 1;       // Going to be a READ.
    bus.BURST <= 0;    // Not a burst.
    bus.ADDR <= addr;  // Put out the address.
    bus.VALID <= 1;    // Signal valid...
    bus.DATAI <= 'z;
    @(negedge bus.CLK);
    while (!bus.READY) wait(bus.READY); // Wait for response ready.
    @(posedge bus.CLK);

    @(negedge bus.CLK);
    while (!bus.READY) wait(bus.READY); // Wait for response ready.
    @(posedge bus.CLK);
    $display("@ %05t: %m READ (%0x)", $time, addr);
    data = bus.DATAO;  // Get the response
    $display("@      : %m READ (%0x)", data);
    bus.VALID <= 0;    // Not valid any more.
  endtask

  task automatic write(bit[31:0] addr, input bit[31:0] data);
    bus.RW <= 0;
    bus.BURST <= 0;
    bus.ADDR <= addr;
    bus.DATAI <= data;
    bus.VALID <= 1;
    @(negedge bus.CLK);
    while (!bus.READY) wait(bus.READY);
```

```
    @(posedge bus.CLK);
    $display("@ %05t: %m WRITE(%0x, %0x)", $time, addr, data);
    bus.VALID <= 0;
  endtask

  task automatic burst_read(bit[31:0] addr,
                    output bit[31:0] data[4]);
    ...
  endtask

  task automatic burst_write(bit[31:0] addr,
                             bit[31:0] data[4]);
    ...
  endtask

  task automatic monitor(output bit transaction_ok,
                      bit rw,
                      bit[31:0]addr,
                      bit[31:0]data[4],
                      bit burst);

    ...
  endtask

endinterface: ABC_bfm
```

VI. The Agent

The agent is a simple container for the driver, monitor and sequencer. Its purpose is to build and connect those pieces. It is a box of parts that operate together on an interface to drive and monitor bus transactions.

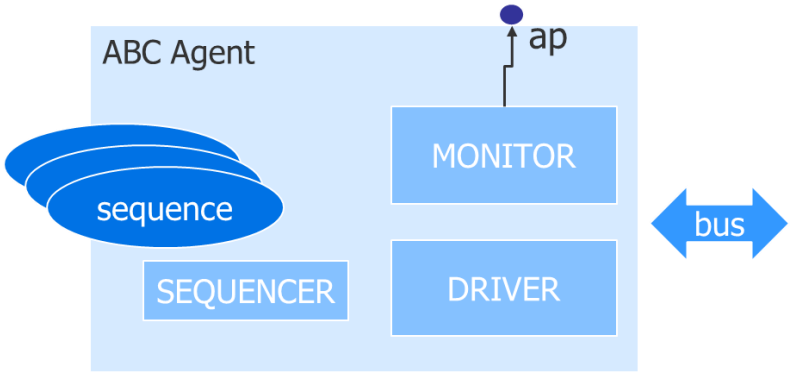

Figure 3: An ABC agent

VII. The Driver

The driver gets a REQUEST from a sequence, and sends the REQUEST out on the bus. It may also receive a RESPONSE from the bus, and return the result to the sequence. Each driver is specific to the particular protocol.

```
class ABC_driver extends uvm_driver#(ABC_seq_item);
  `uvm_component_utils(ABC_driver)

  virtual ABC_bfm bfm;

  ...

  task run_phase(uvm_phase phase);
    ABC_seq_item item;
    ABC_reset_item reset_item;
```

```
    forever begin
      seq_item_port.get_next_item(item);
      `uvm_info("DRIVER",
        $sformatf("item=%s", item.convert2string()),
        UVM_MEDIUM)

      if ($cast(reset_item, item))
        bfm.reset();
      else if (item.rw == 1) // READ
        if (item.burst == 1)
          bfm.burst_read(item.addr, item.data);
        else
          bfm.read(item.addr, item.data[0]);
      else
        if (item.burst == 1)
          bfm.burst_write(item.addr, item.data);
        else
          bfm.write(item.addr, item.data[0]);

      seq_item_port.item_done();
    end
  endtask
endclass: ABC_driver
```

VIII. THE MONITOR

The monitor is a passive component that monitors the activity on the bus, and when it has recognized a "transaction", it creates a transaction object, and publishes it on an analysis port. That analysis port is connected to a subscriber. For example a monitor might be connected to a scoreboard – when a transaction appears at the monitor it creates a transaction object and sends it to the scoreboard. The scoreboard checks correctness, or saves it for later.

```
class ABC_monitor extends uvm_monitor;
  `uvm_component_utils(ABC_monitor)

  virtual ABC_bfm bfm;
  uvm_analysis_port #(ABC_seq_item) ap;

  function void build_phase(uvm_phase phase);
    ap = new("ap", this);
  endfunction

  task run_phase(uvm_phase phase);
    ABC_seq_item t;

    ...

    t = ABC_seq_item::type_id::create("t");
    forever begin
      bit transaction_ok;

      // Each call to monitor waits for at least one clock
      bfm.monitor(transaction_ok,
        t.rw, t.addr, t.data, t.burst);
      if (transaction_ok) begin
        `uvm_info("MONITOR",
          $sformatf(" %s (%s)",
          (t.burst==1)?" Burst":"Normal", t.convert2string()),
          UVM_MEDIUM)
        ap.write(t);
        t = ABC_seq_item::type_id::create("t");
      end
    end
  endtask
endclass: ABC_monitor
```

IX. THE SEQUENCE

A UVM sequence is simple a piece of software – code – that generates transactions for the driver to execute. For example, the small body() task below creates a transaction, calls start_item() (which is really a call to get permission to run on the driver – like a call to 'wait_for_grant()'). Then, when start_item() returns, we have been granted permission to run on the driver. The transaction 'tx' is now randomized (so-called 'late-randomization'), and then finish_item() is called (which is really a call to execute the transaction on the driver – like a call to 'execute()'). So the steps are: Create, Wait for Permission, Randomize, Execute.

```
task body;
  repeat(N) begin
    ABC_seq_item tx;
    tx = ABC_seq_item::type_id::create("tx");
    start_item(tx);
    if(! tx.randomize() )
      `uvm_fatal(get_type_name(), "Randomize failed");
    finish_item(tx);
  end
endtask: body
```

For example a sequence could be used to generate 10,000 read and write transactions between two addresses with the following implementation:

```
class ABC_seq extends uvm_sequence #(ABC_seq_item);
  `uvm_object_utils(ABC_seq)


  int number_of_items = 10000;

  rand bit[31:0] addr;

  rand bit[31:0] addr_low;
  rand bit[31:0] addr_high;

  constraint rand_range {
    addr_low  >=  'h00;
    addr_high <=  'h82;
    addr_low < addr_high;
  }

  constraint addr_value {
  }

  ...

  task body();
    ABC_seq_item item;
    ABC_reset_item reset_item;

    `uvm_info("SEQUENCE", "body() running...", UVM_MEDIUM)

    // Reset the hardware
    reset_item = ABC_reset_item::type_id::create("reset_item");
    start_item(reset_item);
    finish_item(reset_item);

    // Crank out reads and writes
    for(int i = 0; i < number_of_items; i++) begin
      item = ABC_seq_item::type_id::create("item");

      start_item(item);
      if (!item.randomize() with { addr >= local::addr_low;
                                   addr <= local::addr_high; })
      begin
        `uvm_fatal("SEQUENCE", "Randomize FAILED")
      end
```

```
      finish_item(item);
    end
    `uvm_info("SEQUENCE", "body() done...", UVM_MEDIUM)
  endtask
endclass: ABC_seq
```

In order to create a different pattern of addresses, we could create a new sequence, extended from the first, with a different constraint. The constraint below ensures that all addresses are even.

```
class ABC_even_sequence extends ABC_seq;
  `uvm_object_utils(ABC_even_sequence)

  constraint addr_value {
    addr[0] == 1'h0;
  }

  function new(string name = "ABC_even_sequence");
    super.new(name);
    addr_value.constraint_mode(1);
  endfunction

endclass
```

X. THE ENVIRONMENT

The environment is another container, much like the agent. The environment normally contains the agents in use. It's responsible for creating the agents, and setting their virtual interface connections to the DUT. A test builds the environment, so a specific test might decide to "configure" an environment. One example of a configured environment in this example would be to make the ABC_1_agent_h a high-speed interface, while making the XYZ_2_agent_h a low-speed interface. Another configuration would be to make them both high-speed interfaces.

```
class env extends uvm_env;
  `uvm_component_utils(env)

  ABC_agent  ABC_1_agent_h;
  ABC_config ABC_1_config_h;

  XYZ_agent  XYZ_2_agent_h;
  XYZ_config XYZ_2_config_h;

  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    ABC_1_agent_h = ABC_agent::type_id::create("ABC_1_agent_h", this);

    // Setup the AGENT configuration.
    //  1. Construct it.
    //  2. Set the 'bfm' field in the configuration.
    //  3. Set the 'active' field in the configuration.
    //  4. Assign the configuration into the ENV.
    //

    // Construct the AGENT handle
    ABC_1_config_h  = ABC_config::type_id::create("ABC_1_config_h", this);

    if (!uvm_config_db #(virtual ABC_bfm)::get(null, "*", "ABC_1_bfm", ABC_1_config_h.bfm))
      `uvm_fatal("TEST_BASE", "Cannot find AGENT handle")


    // Set any required AGENT handle settings.
    ABC_1_config_h.active = UVM_ACTIVE;
```

```
    ABC_1_agent_h.config_h = ABC_1_config_h;


    // Build an AGENT handle. One for each declared above.
    XYZ_2_agent_h = XYZ_agent::type_id::create("XYZ_2_agent_h", this);

    // Setup the AGENT configuration.
    //  1. Construct it.
    //  2. Set the 'bfm' field in the configuration.
    //  3. Set the 'active' field in the configuration.
    //  4. Assign the configuration into the ENV.
    //

    // Construct the AGENT handle
    XYZ_2_config_h  = XYZ_config::type_id::create("XYZ_2_config_h", this);

    if (!uvm_config_db #(virtual XYZ_bfm)::get(null, "", "XYZ_2_bfm", XYZ_2_config_h.bfm))
      `uvm_fatal("TEST_BASE", "Cannot find AGENT handle")

    // Set any required AGENT handle settings.
    XYZ_2_config_h.active = UVM_ACTIVE;

    XYZ_2_agent_h.config_h = XYZ_2_config_h;

  endfunction: build_phase

endclass: env
```

XI. THE TEST

The test is the name of the running test. For example the “read/write” test or the “walking-ones” test. A test is itself a component that builds an environment, configures it, and starts various sequences. For example, in a DUT with three interfaces, the test might start 3 different sequences, one on each interface. This test would simulate the activity of a busy system. A different test might start just two sequences, with one interface being “off”. This test would simulate the activity of a system in low-power mode.

A test is simply the way that the simulation knows what it should do. It should configure the environment and start some sequences.

```
class test_0 extends uvm_test;
  `uvm_component_utils(test_0)

  env env_h;

  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    env_h = env::type_id::create("env_h", this);

  endfunction: build_phase

  task run_phase (uvm_phase phase);

    // Declare a sequence, one (or more) for each AGENT.
    ABC_seq ABC_1_seq1;
    XYZ_seq XYZ_2_seq1;

    // Construct all the AGENT sequences
    ABC_1_seq1 = ABC_seq::type_id::create("ABC_1_seq1");
    ABC_1_seq1.N = 10;

    XYZ_2_seq1 = XYZ_seq::type_id::create("XYZ_2_seq1");
    XYZ_2_seq1.N = 10;
```

```
    phase.raise_objection(this);

    fork
      // Start all the AGENT sequences (in parallel)
      ABC_1_seq1.start( env_h.ABC_1_agent_h.sequencer_h );
      XYZ_2_seq1.start( env_h.XYZ_2_agent_h.sequencer_h );
    join

    phase.drop_objection(this);

  endtask: run_phase

endclass: test_0
```

XII. THE REST

There are various other components in a UVM test bench, notably the "analysis components" and the Verilog module based connections to the actual DUT.

Analysis components are things like a predictor, a scoreboard, checkers and the golden model. Each of these components has a specific job. For our simple template generation a simple scoreboard will be used with a simple golden model. No predictor, scoreboard or special checker is needed. The checking is performed within each sequence. For a more complex DUT or a more complex set of stimulus such simple checking would not be possible – we might need an out-of-order comparator for example.

The DUT must be connected to the test bench – this is done with virtual interfaces, module instantiations and a global database called the configuration database. The 'top' module instantiates the interfaces and DUT wrapper. The DUT wrapper instantiates the DUT and connects the DUT pins to the SystemVerilog interface signals. The 'test top' instantiates the BFM and starts the UVM test. See below for example code.

The top

The top module is a simple top level that instantiates the actual bus – the signals that need to be connected to the DUT. It also instantiates the DUT wrapper and sets up any clocks that are needed.

```
module top;
  logic clock;

  // Declare the AGENT interface - the BUS (signals)
  ABC_if ABC_1_if (.CLK(clock));
  XYZ_if XYZ_2_if (.CLK(clock));

  dut_wrapper dut1 (
    // Supply the AGENT interface to the DUT.
    .ABC_1_if,
    .XYZ_2_if
  );

  // Clock and reset generator
  initial begin
    clock = 1'b0;
    forever #5 clock = ~clock;
  end

endmodule: top
```

The DUT Wrapper

The DUT wrapper is just a simple wrapper around the DUT that maps from SystemVerilog interface to pins.

```
module dut_wrapper(
  ABC_if ABC_1_if,
  XYZ_if XYZ_2_if
```

```
);

dut_mem u_dut(
  .CLK1  (ABC_1_if.CLK),  // input  wire        CLK1,
  .RST1  (ABC_1_if.RST),  // input  wire        RST1,
  .RW1   (ABC_1_if.RW),   // input  wire        RW1,
  .READY1(ABC_1_if.READY),// output reg         READY1,
  .VALID1(ABC_1_if.VALID),// input  wire        VALID1,
  .ADDR1 (ABC_1_if.ADDR), // input  reg  [31:0] ADDR1,
  .DATAI1(ABC_1_if.DATAI),// input  reg  [31:0] DATAI1,
  .DATAO1(ABC_1_if.DATAO),// output reg  [31:0] DATAO1,

  .CLK2  (XYZ_2_if.CLK),  // input  wire        CLK2,
  .RST2  (XYZ_2_if.RST),  // input  wire        RST2,
  .RW2   (XYZ_2_if.RW),   // input  wire        RW2,
  .READY2(XYZ_2_if.READY),// output reg         READY2,
  .VALID2(XYZ_2_if.VALID),// input  wire        VALID2,
  .ADDR2 (XYZ_2_if.ADDR), // input  reg  [31:0] ADDR2,
  .DATAI2(XYZ_2_if.DATAI),// input  reg  [31:0] DATAI2,
  .DATAO2(XYZ_2_if.DATAO) // output reg  [31:0] DATAO2,

  ...
 );


endmodule: dut_wrapper
```

The Test top

The test top has a simple job. It is a module that instantiates the BFM interfaces. These BFM interfaces will be used by the driver and monitor to operate the actual bus. The test top then stores the BFM interfaces into a global database, searchable by name. Finally, it starts the UVM test by calling 'run_test()'.

```
module test_top;
  import uvm_pkg::*;
  import test_pkg::*;

  // Declare the AGENT BFM - the AGENT interface
  ABC_bfm ABC_1_bfm (top.ABC_1_if);
  XYZ_bfm XYZ_2_bfm (top.XYZ_2_if);

  initial begin
    // Store the AGENT interface into the configuration database.
    uvm_config_db#(virtual ABC_bfm)::set( null, "", "ABC_1_bfm", ABC_1_bfm);
    uvm_config_db#(virtual XYZ_bfm)::set( null, "", "XYZ_2_bfm", XYZ_2_bfm);

    uvm_config_db#(int)::set( uvm_root::get(), "", "recording_detail", 1);

    run_test();
  end
endmodule: test_top
```

XIII. CONCLUSION

Using the UVM constrained random environment is a good idea. It will improve verification productivity. Coverage will improve. Unknowable tests will be run. Unfortunately, learning the UVM can be daunting. Outside resources exist to help, but can take time and money. Using a template system can allow a simple UVM testbench to be created in seconds, and completed by hand within a day. Using templates is not a panacea. A template user still needs to write SystemVerilog code, and must understand some of the details about where edits need to be made to customize the generated code for the specific protocol or DUT.

For success a template system should be small. Ours is 20 files containing about 500 lines of code. The generator should be transparent. Our generator is less than 50 lines line, and consists of invoking ‘sed’ and ‘tar’. Many template systems exist – try one to get a head start on UVM and verification productivity. Please contact the authors for the complete source code – templates, generator and generated example.

XIV. REFERENCES

[1] SystemVerilog LRM, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
[2] UVM 1.1d Reference implementation code, http://www.accellera.org/downloads/standards/uvm/uvm-1.1d.tar.gz