

# It's Not Too Late to Adopt: The Full Power of UVM

Kathleen Wittmann  
Rockwell Automation  
1201 South Second Street  
Milwaukee, WI 53204

**Abstract**—The Universal Verification Methodology (UVM) has existed for over a decade, with its predecessors existing before that. According to Siemens EDA's 2020 Wilson Research Group functional verification study, Accellera UVM is the most used standard adopted for testbenches. With use in over 70% of IC/ASIC testbenches[1] and over 40% of FPGA testbenches[2], there is still room for further adoption.

Upon completion of my group's most recent ASIC development, our retrospective identified several simulation efficiencies we wanted to adopt:

- 1) Incremental compilation, where rather than rebuilding all source files for every testcase run, only modified files are recompiled. This technique saves compilation time.
- 2) Elaboration snapshots, where the design and testbench are elaborated once and subsequent testcase runs go straight to simulation. This technique saves compilation and elaboration time.
- 3) Simulation snapshots, where a snapshot is taken at some point in a simulation and subsequent testcase runs begin at that point. This technique saves compilation, elaboration, and simulation time.

I was tasked with piloting these techniques, providing benchmark comparisons, and recommending the next steps. In exploring these techniques, I had a key realization about our testbench. While we were using elements of UVM in our testbench, we had not fully migrated away from using Verilog compiler directives to statically configure our testbench. We had not yet adopted the full power of UVM to dynamically configure our testbench. This lack of dynamic configuration was causing us to miss all three simulation efficiencies we wanted to adopt.

Between the Wilson Research Group's findings about UVM adoption and my experience with partial UVM adoption, a target audience of late UVM adopters remains. This paper targets those late adopters and those partial adopters. It will describe the statically configured UVM environment I encountered and the barriers it placed to achieving simulation efficiencies. It will quantify the productivity lost by not being able to achieve the simulation efficiencies. Finally, it will highlight the key components of UVM that were necessary for realizing its power.

## I. BARRIERS TO SIMULATION EFFICIENCIES

Elaboration and simulation snapshots require the testbench to have the same static structure across tests. When conditional compiler directives like ``ifdef` are used in the testbench environment and the defined macros are inconsistent across tests, the result is a different static testbench structure across tests.

Some use cases warrant macros and conditional compiler directives in a testbench, e.g., to choose whether to instantiate the ASIC version of the Design Under Test (DUT) or the FPGA-prototype version of the DUT. However, they can also be overused, which I encountered in my testbench.

Figure 1 demonstrates the use of conditional compiler directives that I encountered in my testbench. In testcases that define the `USB_VIP` macro, the UVM Environment will include the USB Agent. In testcases that do not define the `USB_VIP` macro, the UVM Environment will not include the agent at all. The result is a different static environment, meaning that elaboration and simulation snapshots cannot be used. Additionally, the environment must be recompiled, and the command line and/or the arguments file must be extended when the macro is used.

Later in this paper, I will present a better way to define and configure this UVM environment for using or not using the USB VIP for a particular test.

```

class dut_env extends uvm_env;
  `uvm_component_utils(dut_env)

  dut_env_config m_env_cfg;

  `ifdef USB_VIP
    usb_agent m_usb_agent;
  `endif

  extern          function      new          (string name, uvm_component parent);
  extern virtual function void build_phase   (uvm_phase phase);
  extern          function void connect_phase (uvm_phase phase);
endclass: dut_env

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

  if (!uvm_config_db #(dut_env_config)::get(this, "", "dut_env_config", m_env_cfg)) begin
    `uvm_error("CONFIG_DB", "dut_env_config not found")
  end

  `ifdef USB_VIP
    m_usb_agent = usb_agent::type_id::create("m_usb_agent", this);
  `endif

endfunction: build_phase

```

Figure 1. Environment Definition with Conditional Compilation

## II. PRODUCTIVITY LOST... AND GAINED

There are several ways in which productivity was lost due to the statically configured UVM environment presented in Figure 1:

- 1) The environment must be recompiled when USB\_VIP is defined or not.
- 2) Elaboration and simulation snapshots cannot be used across tests that define USB\_VIP and those that do not.
- 3) The command line and/or the arguments file must be extended when the macro is used.

One might suggest managing two separate compilation libraries and snapshots for the two cases of USB\_VIP being defined or not; however, the use of conditional compiler directives was more extensive than conveyed in this example. As a result, compilation, elaboration, and simulation were performed for every single testcase.

The first step to motivating the adoption of an improved verification environment was to benchmark the performance gains available in incremental compilation and elaboration and simulation snapshots. A test suite of 31 tests, with an initial average run time of 31.5 minutes per test, was identified for this benchmarking exercise. This was just one test suite of an overall ASIC regression that consists of approximately 2000 tests.

Moving to an environment that could be compiled once and the compiled library used across testcases would save 0.8 minutes per test. Multiply that savings across 2000 tests, and 1600 minutes of processing time have been saved, or 26.7 hours.

Having one common work library across testcases opens the opportunity to use elaboration snapshots. The DUT and testbench can then be compiled and elaborated once and simulated many times. Using the same 31-test suite, I found that 8.6 minutes could be saved per test. Across 2000 tests, it is nearly 12 days of processing time.

Of course, DUTs and testbenches and regression suites all differ, but these numbers give an idea of the productivity my group was losing and that we stood to gain.

## III. REALIZING THE POWER OF UVM

Instead of statically configuring a verification environment as demonstrated in Figure 1, UVM offers the ability to dynamically configure the environment.

Figure 2 demonstrates how the UVM environment can be rewritten to take advantage of this dynamic configuration. In this environment, the USB Agent is always a member of the environment, but whether it is built depends on how the environment has been configured.

```

class dut_env extends uvm_env;
  `uvm_component_utils(dut_env)

  dut_env_config m_env_cfg;

  usb_agent      m_usb_agent;

  extern          function      new          (string name, uvm_component parent);
  extern virtual function void build_phase   (uvm_phase phase);
  extern          function void connect_phase (uvm_phase phase);
endclass: dut_env

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

  if (!uvm_config_db #(dut_env_config)::get(this, "", "dut_env_config", m_env_cfg)) begin
    `uvm_error("CONFIG_DB", "dut_env_config not found")
  end

  if (m_env_cfg.has_agent_usb) begin
    uvm_config_db #(usb_agent_config)::set( this,
                                             "m_usb_agent",
                                             "usb_agent_config",
                                             m_env_cfg.m_usb_agent_cfg)
    m_usb_agent = usb_agent::type_id::create("m_usb_agent", this);
  end
end

endfunction: build_phase

```

Figure 2. Environment Definition with Dynamic Configuration

The UVM test configures the UVM environment and is where object-oriented programming's (OOP) polymorphism concept adds power to UVM. Figure 3 defines the base test for this testbench. It defines a virtual function, `configure_dut_env`, to configure the environment. This function sets the environment configuration object's member `has_agent_usb` to 1, indicating that the environment should create the USB Agent.

```

class dut_test_base extends uvm_test;
    ...
    extern          function void build_phase      (uvm_phase phase);
    extern virtual function void configure_dut_env(dut_env_config cfg);
endclass: dut_test_base

function void dut_test_base::build_phase(uvm_phase phase);
    // Create the Environment Configuration Object
    m_dut_env_cfg = dut_env_config::type_id::create("m_dut_env_cfg");
    configure_dut_env(m_dut_env_cfg);

    // Create and configure the Agent Configuration Objects
    // Assign the Agent Configuration Objects in the Environment Configuration Object
    ...

    // Put the environment configuration in the uvm_config_db
    uvm_config_db #(dut_env_config)::set(this, "*", "dut_env_config", m_dut_env_cfg);

    // Create the environment
    m_env = dut_env::type_id::create("m_env", this);

endfunction: build_phase

function void dut_test_base::configure_dut_env (dut_env_config cfg);
    cfg.has_agent_usb = 1;
endfunction: configure_dut_env

```

Figure 3. Virtual Function in Test Base Class

Tests that inherit from the test base class have two options:

- 1) A test that inherits from the test base class can use the base class's function to do a default configuration. In this case, a test that inherits from `dut_test_base` would not define an overriding `configure_dut_env` function. When this inherited test then calls `configure_dut_env` in its `build_phase`, it executes its parent's `configure_dut_env`. In this example, the `configure_dut_env` function defined in Figure 3 is executed and sets `has_agent_usb` to 1.
- 2) Alternatively, a test that inherits from the test base class can declare its own overriding function. This overriding function follows the same prototype as the function in the base class – the same name, inputs, and outputs. Because virtual methods are inheritable and overridable, there is only one implementation of a virtual method per class hierarchy, and it is always the one in the last derived class. Therefore, when this inherited test then calls `configure_dut_env` in its `build_phase`, it executes its own overriding function. In the example in Figure 4, the function in the derived `test_without_usb` class is executed. It sets `has_agent_usb` to 0, resulting in the USB Agent not being created.

```

class test_without_usb extends dut_test_base;

    `uvm_component_utils(test_without_usb)
    ...
endclass: test_without_usb

function void test_without_usb::configure_dut_env (dut_env_config cfg);
    cfg.has_agent_usb = 0;
endfunction: configure_dut_env

```

Figure 4. Overriding Function in Derived Test

#### IV. BUILDING ON THE POWER OF UVM

The previous section presented a simple example of how a virtual function and function override are used to dynamically configure the UVM environment for a given test. In the example, the environment either created or did not create the USB agent, depending on how the test class configured the environment. This same method can be used to build or not build other components, like a scoreboard. For example, it is common to elect to not build the scoreboard in register tests, where the only concern is whether the register fields reflect the proper access modes.

Sequence override is another powerful option in UVM. In the past, my organization largely used a 1:1 UVM test to sequence relationship. Each test started one main sequence, which would stimulate the DUT as specified for that testcase and possibly perform some error checking. We found that some error checking was better performed in that sequence than in the scoreboard. Figure 5 demonstrates the definition of two such tests.

```
class test_adc_interface extends dut_test_base;
    `uvm_component_utils(test_adc_interface)
    ...
    extern task run_phase(uvm_phase phase);
endclass: test_adc_interface

task test_adc_interface::run_phase(uvm_phase phase);
    seq_adc_interface_t t_seq = seq_adc_interface_t::type_id::create("t_seq");

    phase.raise_objection(this);

    assert(t_seq.randomize());
    t_seq.start(null);

    phase.drop_objection(this);
endtask: run_phase

class test_spi extends dut_test_base;
    `uvm_component_utils(test_spi)
    ...
    extern task run_phase(uvm_phase phase);
endclass: test_spi

task test_spi::run_phase(uvm_phase phase);
    seq_spi_t t_seq = seq_spi_t::type_id::create("t_seq");

    phase.raise_objection(this);

    assert(t_seq.randomize());
    t_seq.start(null);

    phase.drop_objection(this);
endtask: run_phase
```

Figure 5. One Sequence for Every Test

This approach of having each UVM test run a dedicated sequence is acceptable and may be well-suited to some projects. In this example, the two tests did not share a common stimulus, such as a common initialization sequence.

Where simulation snapshots add value is in allowing a long initialization sequence to be simulated once and subsequent simulations to pick up where it ended and spend the compute time on new and interesting stimuli rather than on initializing the DUT again.

In investigating simulation snapshots for my organization, we used the Cadence simulator. Its simulation snapshot method, called Process Based Save Restart (PBSR), requires a single UVM test from which different sequences can be started [3]. Figure 6 provides an example of this single UVM test. In this example, the test first starts an initialization sequence and then starts a stimulus sequence. The simulation snapshot is created between the two

sequences. The first invocation of this test would create the simulation snapshot and run the default 'seq\_stim\_base' sequence. Subsequent invocations would use the simulation snapshot and override 'seq\_stim\_base' with a different sequence to achieve a new stimulus.

```

class test_usb extends dut_test_base;
  `uvm_component_utils(test_usb )

  seq_init_dut init_seq;
  usb_seq_base stim_seq;

  ...

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    `uvm_info(get_type_name(),"Starting initialization",UVM_LOW)
    init_seq = seq_init_dut::type_id::create("init_seq");
    init_seq.start(null);

    // Simulation snapshot is created at this point
    // Details of creating this snapshot are out of scope for this paper

    `uvm_info(get_type_name(),"Starting stimulus, which can be overridden",UVM_LOW)
    stim_seq = usb_seq_base::type_id::create("stim_seq");
    stim_seq.start(null);

    phase.drop_objection(this);
  endtask: run_phase
endclass : test_usb

```

Figure 6. Test Used in Simulation Snapshot

Sequence override is achieved through inherited sequence classes. Three sequences are declared in Figure 7. The first, `usb_seq_base`, is the sequence that `test_usb` of Figure 6 starts by default. The other two sequences, `seq_exercise_dma` and `seq_suspend_resume`, inherit from `usb_seq_base`. Because of this inheritance, an invocation of `test_usb` after the creation of the simulation snapshot can go straight to the point where initialization has already been completed and execute a new and interesting stimulus by overriding `usb_seq_base` with a derived class.



```

class usb_seq_base extends dut_seq_base;
    `uvm_object_utils(usb_seq_base)

    // Declare USB-SPECIFIC sequencers, sequence items, sequences, and
    // register models that can be used by derived classes

    // Declare USB-SPECIFIC commonly used functions and tasks

    task body();
        // Do USB-SPECIFIC stimulus and/or create USB-SPECIFIC transactions
        // for use by derived classes
    endtask: body
endclass: usb_seq_base

class seq_exercise_dma extends usb_seq_base;
    `uvm_object_utils(seq_exercise_dma)
    ....

    task body();
        // Do some stimulus
    endtask: body
endclass: seq_exercise_dma

class seq_suspend_resume extends usb_seq_base;
    `uvm_object_utils(seq_suspend_resume)
    ....

    task body();
        // Do some stimulus
    endtask: body
endclass: seq_suspend_resume

```

Figure 7. Derived Sequences for Use in Simulation Snapshots

This paper previously reported that using elaboration snapshots saved 8.6 minutes per test in a 31-test benchmark. Simulation snapshots can save even more by removing duplicated simulation cycles from the overall regression. Across a 2000-test regression, the savings can easily amount to days of processing time.

## V. INSTITUTIONALIZING THE POWER OF UVM

Given the efficiency gained by fully adopting the dynamic configuration power of UVM, it is important to enable that adoption. To that end, my organization has taken some steps.

First, the efficiency data in this paper has been presented as the motivation for why we must make changes.

Second, a script and templates were written to automate the creation of UVM testbenches. This template-based, automated testbench generation guarantees a similar look and feel across testbenches and builds in the elements needed for dynamic configuration from the start. Some details about the script, templates, and generated testbench are presented in the next section.

Finally, periodic knowledge share, a documented methodology, and a review of the testbench at the start of each new project will help to institutionalize the full adoption of UVM.

In taking these steps, one important thing my organization formalized is a plan for sequence inheritance. Figure 7 declared `usb_seq_base` and two sequences derived from it. `usb_seq_base` itself derived from `dut_seq_base`, which is now presented in Figure 8.

```

class dut_seq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(dut_seq_base)

  dut_env_config m_env_cfg;

  // Declare sequencers, sequence items, sequences, and
  // register models that can be used by derived classes

  // Declare commonly used functions and tasks

  task body();
    // Get the environment configuration object
    if(!uvm_config_db #(dut_env_config)::get(null,
                                              {"uvm_test_top.",get_full_name()},
                                              "dut_env_config",
                                              m_env_cfg)) begin
      `uvm_fatal("body", "dut_env_config configuration object not found")
    end

    // Create transactions
    spi_txn = spi_frame::type_id::create("spi_txn");
    ...

    // Prepare the register model for use by the derived test sequences
    dut_rm = m_env_cfg.dut_rm;

    // Connect sequencer handles for each active agent in environment
    if (m_env_cfg.m_dut_reset_agent_cfg.is_active == UVM_ACTIVE) begin
      dut_reset_sqr = m_env_cfg.m_dut_reset_sqr;
    end
    m_spi_sqr = new[m_env_cfg.num_agent_spi];
    for (int i=0; i<m_env_cfg.num_agent_spi; i++) begin
      m_spi_sqr[i] = m_env_cfg.m_spi_sqr[i];
    end
  endtask: body
endclass: dut_seq_base

```

Figure 8. Sequence Base Class for the DUT

The intent is for `dut_seq_base` to serve as the sequence base class in a top-level testbench. It declares elements that might be common to all derived sequences, such as register models and tasks to reset the DUT or wait for a specified amount of time.

Within that top-level testbench, a suite of tests may require components, functions, and tasks that are unique to it and do not belong in `dut_seq_base`. A subsystem base sequence class, like `usb_seq_base` in Figure 7, derives from `dut_seq_base` and declares elements specific to its suite of tests. Other test sequences for that subsystem, for example `seq_exercise_dma` and `seq_suspend_resume` in Figure 7, would then derive from the subsystem's base sequence class. This class hierarchy is demonstrated in Figure 9.

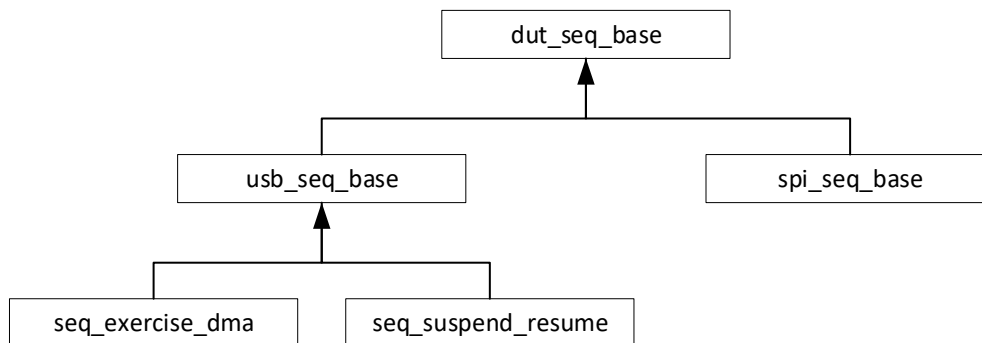


Figure 9. Sequence Class Hierarchy

## VI. THE TESTBENCH ITSELF

As mentioned in the second step outlined in the previous section, my organization wrote a script and templates to automate the creation of UVM testbenches, build in the elements needed for dynamic configuration, and lay the groundwork for the sequence inheritance needed for simulation snapshots. The perl script's only required input is the unit name, and it optionally takes the names of up to four agents, a register block name, the default address map for that register block, and a register adapter. The script invokes SystemVerilog Assistant, a tool in the HDL Designer



Series of tools from Siemens EDA, to generate a UVM environment and testbench from pre-existing templates. These templates are ultimately derived from templates provided with the Siemens EDA tool. The script then fills in component names throughout the UVM environment, based on the specified inputs.

The script generates a testbench that puts the SystemVerilog interface for each agent into the `uvm_config_db` and calls `run_test()`, where the testcase name is specified on the command line. This generated testbench is shown in Figure 10.

```

module dut_tb;

    import uvm_pkg::*;
    import dut_test_pkg::*;

    // Virtual interface for each agent
    apb_uvc_if    vif_apb    (clk, rst_l);
    spi_uvc_if    vif_spi    (clk, rst_l);

    wave_gen_if    dut_reset_if(.clock(clk), .resetn(rst_l));

    reset_gen#(.TIME(200000), .POLARITY(0))
        i_reset_gen(.reset(rst_l));
    clk_gen    #(.PERIOD(12500), .POLARITY(0))
        i_clk_gen(.clk(clk));

    dut my_dut(
        .clk    (clk),
        .rst_l  (dut_rst_l)
    );

    assign dut_rst_l = dut_reset_if.InputPin & rst_l;

    // Place the virtual interfaces in the uvm_config_db & execute run_test()
    initial begin
        uvm_config_db #(virtual wave_gen_if)::set(null, "uvm_test_top", "vif_dut_reset", dut_reset_if);
        uvm_config_db #(virtual apb_uvc_if)::set( null, "uvm_test_top", "vif_apb",    vif_apb);
        uvm_config_db #(virtual spi_uvc_if)::set( null, "uvm_test_top", "vif_spi",    vif_spi);

        run_test();
    end
endmodule: dut_tb

```

**Figure 10. Generated Testbench**

The testcase name specified on the command line is a UVM test derived from the test base class. The script and templates generated the test base class, and its members include environment and agent configuration objects. The derived test may choose to override a default environment or agent configuration through an overriding function, as shown earlier in this paper in Figure 4.

During their build\_phase, the test base class, and therefore any derived class, gets the virtual interfaces out of the `uvm_config_db` and assigns them to their agent configuration objects. Recall that the virtual interfaces were put into the `uvm_config_db` by the testbench itself, shown in Figure 10. The test then creates the environment.

During their connect\_phase, the test base class and its derived classes connect sequencers from the active agents in the environment to sequencers in the environment configuration object. Figure 11 declares a test base class, `dut_test_base`, and shows the creation of these environment and agent configuration objects and their connections to the agents in the environment. Figure 12 depicts the architecture itself and the connection of configuration objects.

```

class dut_test_base extends uvm_test;
    dut_env          m_env;
    dut_env_config    m_dut_env_cfg;

    wave_gen_agent_config m_dut_reset_agent_cfg;
    apb_agent_config      m_apb_agent_cfg;
    spi_agent_config      m_spi_agent_cfg;
    ...
    extern          function void build_phase      (uvm_phase phase);
    extern virtual function void configure_dut_env(dut_env_config cfg);
endclass: dut_test_base

function void dut_test_base::build_phase(uvm_phase phase);
    // Create the Environment Configuration Object
    m_dut_env_cfg = dut_env_config::type_id::create("m_dut_env_cfg");
    configure_dut_env(m_dut_env_cfg);

    // Create and configure the Agent Configuration Objects
    // Assign the Agent Configuration Objects in the Environment Configuration Object
    m_spi_agent_cfg = spi_agent_config::type_id::create("m_spi_agent_cfg");
    if(!uvm_config_db #(virtual spi_uvc_if)::get(this, "", "vif_spi", m_spi_agent_cfg.vif)) begin
        `uvm_error("RESOURCE_ERROR", "spi_uvc_if virtual interface not found")
    end
    m_dut_env_cfg.m_spi_agent_cfg = m_spi_agent_cfg;
    configure_spi_agent(m_spi_agent_cfg);
    uvm_config_db #(dut_env_config)::set(this, "*", "dut_env_config", m_dut_env_cfg);
    ...

    // Put the environment configuration in the uvm_config_db
    uvm_config_db #(dut_env_config)::set(this, "*", "dut_env_config", m_dut_env_cfg);

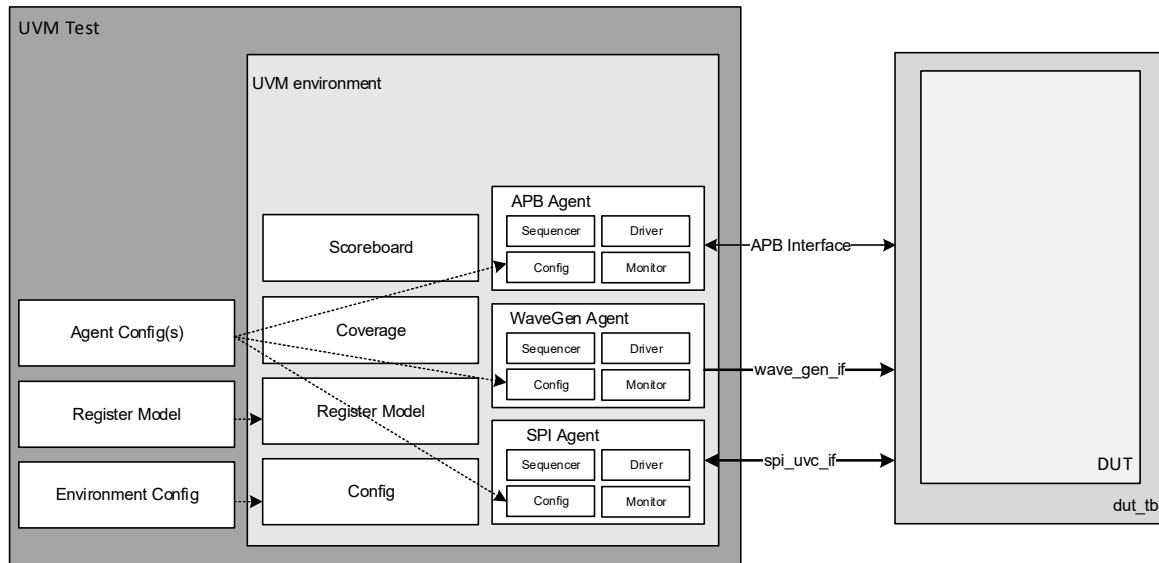
    // Create the environment
    m_env = dut_env::type_id::create("m_env", this);

endfunction: build_phase

function void dut_test_base::connect_phase(uvm_phase phase);
    // Connect sequencer handles for each active agent
    if (m_dut_env_cfg.m_spi_agent_cfg.is_active == UVM_ACTIVE) begin
        m_dut_env_cfg.m_spi_sqr = m_env.m_spi_agent.m_sequencer;
    end
    ...
endfunction: connect_phase

```

Figure 11. Test Base Class Creation of Configuration Objects



**Figure 12. UVM Test Architecture**

The test, derived from the base test class, calls sequences that are often derived from a sequence base class, like `dut_seq_base` or `usb_seq_base`. These sequence base classes get the environment configuration object from the `uvm_config_db` in order to use the sequencers that were connected by the test. This retrieval of the environment configuration object and its sequencers is demonstrated in Figure 8.

The environment, created by the test base class, also gets the environment configuration from the `uvm_config_db` and uses it to determine what agents, scoreboards, and coverage classes to create and connect. The environment's use of the environment configuration is demonstrated in Figure 2.

This description of our generated testbench demonstrates the importance of configuration objects in the overall structure of a UVM-based testbench, as well as to its dynamic configuration. The UVM test uses these configuration objects to configure and convey information to the UVM sequences and UVM environment and beyond.

## VII. SUMMARY

This paper touched on just a couple of UVM's "powers." First, productivity gains can be achieved when incremental compilation and elaboration snapshots are made possible with a dynamically configured testbench. Second, even more productivity gains can be achieved when simulation snapshots are enabled through a carefully planned approach to sequence class inheritance.

There are still more "powers" to be enjoyed than what was discussed in this paper. For example, a UVM register model can be used to abstract the registers in a design. Realizing these "powers" does take an investment in knowledge, methodology, and infrastructure, but there are rewards to be reaped.

## REFERENCES

- [1] Siemens EDA, "2020 Wilson Research Group functional verification study: IC/ASIC functional verification trend report."
- [2] Siemens EDA, "2020 Wilson Research Group functional verification study: FPGA functional verification trend report."
- [3] Cadence, "Process based save restart in UVM."