

Translation of an Existing VMM-based SystemVerilog Testbench to URM

Analog Devices, Inc.

Kelly D. Larson
kelly.larson@analog.com

Session # 1.8



Presented at

cadence designer network



Silicon Valley 2007

ABSTRACT

Many features built into the SystemVerilog language make it an extremely effective high-level verification language. Using class libraries with SystemVerilog can take this a step further by enhancing productivity, and enabling better, more efficient reuse between engineers and between projects. The *Verification Methodology Manual* (VMM) class library was one of the first SystemVerilog class libraries available, and has been widely adopted. The *Universal Reuse Methodology* (URM) class library has more recently become available, and while it is similar to VMM in many respects, there are also some important differences. This paper will describe the process of converting an existing testbench based on VMM class libraries, to one based on URM class libraries. It will summarize the similarities and differences between the two approaches, and highlight which aspects of the conversion were straight forward, and which aspects required more attention.

Table of Contents

1	Introduction.....	5
2	EBIU Design Description	5
3	EBIU VMM Testbench.....	6
4	Basic Components	7
4.1	Transaction Objects	7
4.2	Bus Functional Models	9
4.2.1	Control Loop.....	9
4.2.2	Message and Error Logging.....	11
4.2.3	urm_unit Macros.....	12
5	Building the Environment.....	12
5.1	Agent Class	15
5.2	UVC (Universal Verification Component).....	17
5.3	SVE (Simulation & Verification Environment)	18
5.4	Build() Phase.....	19
5.5	Configuration of the DUT.....	21
6	Testing the DUT	23
6.1	Sequences & Scenarios	23
6.2	urm_test Objects	25
7	Conclusion	25
8	Acknowledgements.....	26

Table of Figures

Figure 1 - EBIU Block Diagram.....	6
Figure 2 - EBIU VMM Testbench.....	7
Figure 3 - vmm_data Class for aus_data.....	8
Figure 4 - urm_sequence_item for aus_data.....	9
Figure 5 – VMM Control Loop for DEB Master.....	10
Figure 6 – URM Control Loop for DEB Master.....	11
Figure 7 – Translation of VMM Log Messages.....	11
Figure 8 – urm_unit Macros	12
Figure 9 – vmm_env Class	14
Figure 10 - EBIU URM Testbench.....	15
Figure 11 – urm_driver Class for DEB Master.....	16
Figure 12 – Agent Class for DEB Master BFM	16
Figure 13 - UVC Block Diagram for the PAB Bus	17
Figure 14 – UVC Code for the PAB.....	18
Figure 15 – SVE Class.....	19
Figure 16 – Top-level print() Output	21
Figure 17 - Simulation Phase Comparison	22
Figure 18 – VMM Test Scenario	23
Figure 19 – URM Test Sequence.....	24
Figure 20 – Simple Test.....	25

1 Introduction

Many features built into the SystemVerilog language make it an extremely effective high-level verification language. The use of class libraries can extend the benefits of SystemVerilog by enhancing productivity, and enabling better, more efficient reuse between engineers, and between various projects. The *Verification Methodology Manual* (VMM) class library was one of the first SystemVerilog class libraries available, and has been widely adopted. The *Universal Reuse Methodology* (URM) class library has more recently become available.

Within the Austin Design Center at Analog Devices we have been developing testbenches with VMM class libraries for some time. While this has been very successful for us, there was also interest in evaluating what it would take to translate an existing VMM testbench into the URM class library. One motivation behind this was simply to compare features, and ensure that we continue to use methodologies which are best in class. A second motivation is the fact that Synopsys and Cadence simulators are both used within our corporation, and within our own design center. Although both class libraries are promoted as being open, because of differences in language support, at the time of this writing VMM is only compatible with the Synopsys simulator, and URM with the Cadence simulator. Knowing the scope of effort which is required to port testbenches between the two libraries could help maximize the utilization of our tool licenses, as well as reuse with other groups.

This paper assumes a basic knowledge of the VMM class library. This paper does not assume prior knowledge of URM, but it is not intended to be a tutorial of the URM class library, nor is it intended to be an exhaustive guide for testbench conversion. The paper will give an overview of the process of converting an existing VMM testbench into URM, as well as some general observations on similarities and differences between the two approaches.

2 EBIU Design Description

The VMM testbench selected for the conversion to URM was developed for the verification of a custom memory controller for a general purpose DSP SOC. The EBIU (External Bus Interface Unit) is complex enough to require a non-trivial testbench, and the large number of system busses makes it an interesting candidate for a randomized SystemVerilog testbench approach.

The EBIU supports two different types of memory interfaces, asynchronous and synchronous. The synchronous interface supports Dual Data Rate (DDR) SDRAM. The asynchronous interface supports four banks of memories such as SRAM and FLASH. All four banks can be configured independently, and FLASH memories can be operated in normal asynchronous, as well as page and burst modes.

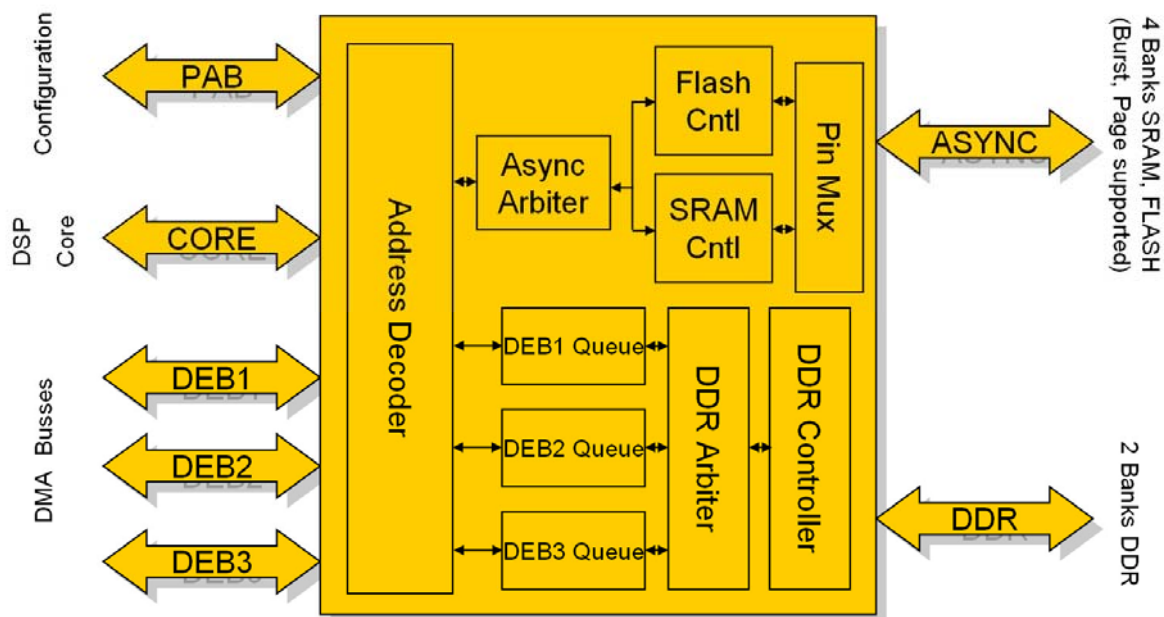


Figure 1 - EBIU Block Diagram

On the system side, the EBIU functions as a slave on five internal busses. These busses consist of a 32-bit DSP processor bus, a 16-bit DMA bus, 2 32-bit DMA busses, and a PAB bus for the programming of internal registers. Advanced features of the EBIU include programmable arbitration, error detection, data reordering for slow memories, write packing, and programmable prefetching.

3 EBIU VMM Testbench

A block diagram of the VMM testbench for the EBIU is shown in Figure 2. This testbench provides three levels of randomization:

- 1) External Environment – Randomization of the external environment is done at runtime, early in the simulation. This consists of selecting an external clock frequency, and also choosing what types of memories (size, mode, speed) should be populated in the testbench. Once a memory type has been selected for each slot, the proper memory model is dynamically muxed to the DUT in the testbench.
- 2) Configuration Registers – Configuration registers are programmed with random values. This sets up various operating modes, arbitration priorities, and timing parameters, all subject to any constraints imposed by the choices from step #1.
- 3) Random Transactions – The main part of a typical test consists of random transactions on all system busses. The testbench contains shadow values for all accessed regions of memory, which is used for the checking of read values.

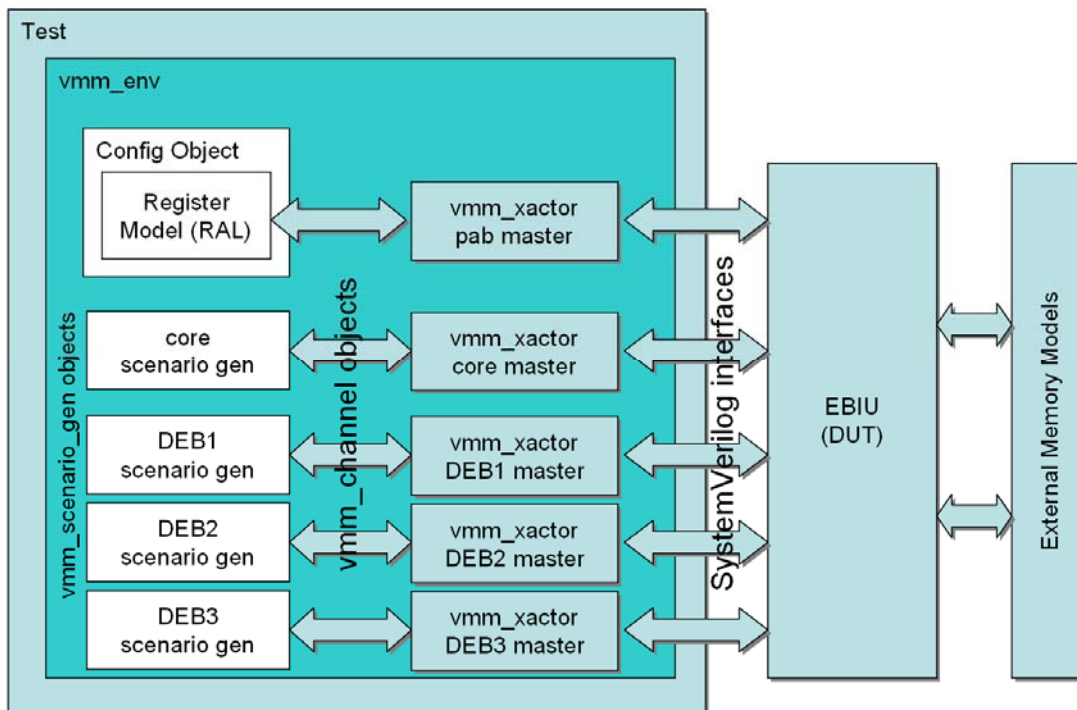


Figure 2 - EBIU VMM Testbench

Standard VMM components were used throughout the testbench, including VMM transactors, channels and scenario generators. A scoreboard for checking and coverage was developed during the verification of the EBIU, but was not included in the initial port to URM.

4 Basic Components

4.1 Transaction Objects

The first step for translating the EBIU testbench to URM was to convert the transaction objects for all three types of system busses, core, DEB & PAB. In VMM the transaction class is extended from the vmm_data class. To be VMM-compliant, a vmm_data object is required to implement several functions which govern how the data is displayed to the screen, copy and compare functions, as well as routines for byte packing and unpacking, if used. Figure 3 shows a selection of code for the VMM version of aus_data, which is the parent transaction class for each of the specific bus transaction classes used in the EBIU.

```

class aus_data extends vmm_data;

    typedef enum {INVALID, ERROR, SYNC, READ8, WRITE8, READ16,
                 WRITE16, READ32, WRITE32} transtype_e;

    typedef enum {w32, w16, w8} buswidth_e;

    static vmm_log log = new("aus_data","class");

    rand transtype_e transtype;
    rand logic [31:0] data [];
    rand logic [31:0] address;
    rand int delay_issue = 0;
    bit          error = 0;

    extern virtual function string psdisplay(string prefix = "");
    extern virtual function vmm_data copy(vmm_data copyto =
null);
    extern virtual function bit compare(input vmm_data to, output
string diff, input int kind);

    // Constraints
    <...>
    // Utility functions
    <...>

```

Figure 3 - vmm_data Class for aus_data

The URM class library eliminates much of the overhead of implementing the transaction class by providing a set of macros which automatically set up all of the required functions for the testbench developer. As with VMM transaction class, these functions govern which data members are included when the object is displayed on the screen, which are included in compare and copy functions, and which to use for packing and unpacking functions. Modifiers used along with these macros can change display formats, or even designate whether a deep or shallow copy should be used for a particular variable.

Translation of the EBIU transaction objects from VMM to URM was relatively straightforward. For each transaction object, all of the data members, constraints, and utility functions migrated directly from one version to the next. URM macros were added for each of the data members, and the redundant VMM routines were deleted (*psdisplay()*, *copy()*, *compare()*, etc.), as this functionality is provided automatically by the *urm_object_utils* macros.

```

class aus_data extends urm_sequence_item;

    typedef enum {INVALID, ERROR, SYNC, READ8, WRITE8, READ16,
                 WRITE16, READ32, WRITE32} transtype_e;

    typedef enum {w32, w16, w8} buswidth_e;
    typedef enum bit {ACTIVE = 1, PASSIVE = 0} active_passive_e;

    rand transtype_e transtype;
    rand logic [31:0] data [];
    rand logic [31:0] address;
    rand int delay_issue = 0;
    bit          error = 0;

    `urm_object_utils_begin(aus_data)
    `urm_field_int(transtype, ALL_ON + ENUM)
    `urm_field_array_int(data, ALL_ON)
    `urm_field_int(address, ALL_ON)
    `urm_field_int(darray_size, ALL_ON)
    `urm_field_int(delay_issue, ALL_ON)
    `urm_object_utils_end

    // Constraints
    <...>
    // Utility functions
    <...>

```

Figure 4 - urm_sequence_item for aus_data

Figure 4 shows a portion of the code for the URM implementation of the `aus_data` class, including the `urm_object_utils` macro used to provide all of the required class methods.

4.2 Bus Functional Models

The next step for translating the EBIU testbench to URM was to translate each of the BFM's used to convert abstract transaction objects into physical bus signals. The URM class object which corresponds to the `vmm_xactor` object is the `urm_unit`. The approach taken here was to make a minimal number of changes, leaving the majority of the BFM code untouched. This was fairly easy to accomplish, as most of the changes fell into two categories, changes to the main control loop, and changes to the message logging.

4.2.1 Control Loop

The main control loop determines how transaction objects are passed from the testbench to the BFM, and also provides synchronization between the BFM and the rest of the environment.

```

protected virtual task deb_master_xactor::main();
  deb_txn txn;
  fork
    super.main();
    monitor_readack();
  join_none
  forever begin: main_loop
    wait_if_stopped_or_empty(in_chan);
    in_chan.get(txn);

    <...>      // Do pre-transaction callbacks
    txn.notify.indicate(vmm_data::STARTED);

    // Launch transaction
    do_transaction(txn);

    <...>      // post-transaction callbacks
    txn.notify.indicate(vmm_data::ENDED);

  end: main_loop
endtask: main

```

Figure 5 – VMM Control Loop for DEB Master

The main control loop for each of the BFM's in the VMM version of the testbench made use of the `wait_if_stopped_or_empty()` task call from within a forever loop to pull the next available transaction from a `vmm_channel` object. This call will block if there isn't a transaction available and waiting, or if the transactor has been told to stop via the `stop_xactor()` call. A `vmm_notify` object within the transaction object was used to indicate to the testbench that a transaction was initiated, and when it was complete (See Figure 5.)

```

task adi_deb_master_bfm::run();
  fork
    get_and_drive();
    monitor_readack();
    reset_signals();
  join
endtask

// This task constantly pull master_deb_txns from the
// driver and responds to transactions
task adi_deb_master_bfm::get_and_drive();
  urm_sequence_item item;
  deb_txn this_master_deb_txn;
  @(negedge deb.reset);

  forever begin
    driver.get_next_item(item);
    $cast(this_master_deb_txn, item);
    do_transaction(this_master_deb_txn);
    driver.item_done_trigger(); // Indicate txn finish
  end
endtask

```

Figure 6 – URM Control Loop for DEB Master

While the URM class library supports TLM channels, which are very similar in function to the `vmm_channel` object, the recommended approach when passing transactions from a driver to a BFM within the same agent object is to pull them directly from the driver. See Figure 6 for the control loop of the DEB BFM master in URM. Transactions must be completed by calling `item_done_trigger()`, which tells the driver that the BFM is ready for the next transaction. Note that the BFM communicates directly to the driver object, which is different than the VMM approach.

4.2.2 Message and Error Logging

Message and error logging is done through macros in URM in much the same way as it's done in VMM. One difference with the URM macros is that it isn't necessary to reference a log object. See Figure 7 for some example translations of messages and error logging.

```

`vmm_note(log,$psprintf("Misaligned 32 bit access: %h",
                        address));
`urm_infol("Misaligned 32 bit access: %h", address);

`vmm_fatal(log,"Data Mismatch");
`urm_fatal("Data Mismatch");

`vmm_trace(log,"Dropped Transaction");
`message(LOW,"Dropped Transaction");

```

Figure 7 – Translation of VMM Log Messages

4.2.3 urm_unit Macros

An additional requirement for the `urm_unit` class is the inclusion of `urm_unit_utils` macros similar to the `urm_object_utils` macros seen in Figure 4 for the `urm_sequence_item` transaction class. These macros provide implementations of virtual methods, set display characteristics, and also define which variables are intended to be configurable by the user. Figure 8 shows an example of these macros for the master BFM of the DEB interface.

```
class adi_deb_master_bfm extends urm_unit;
  // The virtual interface used to drive and view HDL signals.
  virtual adi_deb_if      deb;
  adi_deb_master_driver  driver;
  aus_data::buswidth_e   width;
  bit                    allow_delay_issue = 0;

  <...>

  // Provide implementations of virtual methods
  // such as get_type_name and create
  `urm_unit_utils_begin(adi_deb_master_bfm)
    `urm_field_int(allow_delay_issue, ALL_ON)
    `urm_field_object(driver, PRINT+REFERENCE)
  `urm_unit_utils_end

  <...>
```

Figure 8 – urm_unit Macros

5 Building the Environment

Up until this point things have matched up fairly closely between the URM and the VMM environment. Transaction objects and BFMs appear very similar to one another between the two class libraries. Above this layer, however, things start to diverge slightly as the two approaches abstract the testbench components differently within the testbench environment.

In the typical VMM environment, the `vmm_env` object is the heart of the testbench. There the environment is randomized, objects are created, communication channels are established, and most of the major components of the testbench are visible at that layer. Each of the steps of the test flow is visible within the `vmm_env` framework. Figure 9 shows selections of this code (all busses but the core bus are removed for simplicity.)

```

class ebiu_env extends vmm_ral_env;
  // Test parms
  test_cfg cfg;
  // RAL model
  ral_block_EBIU ral_model = new;
  // Channels
  core_txn_channel core_channel;
  <...> // Only CORE shown for simplicity
  // Xactors
  core_master_xactor core_master;
  <...>
  // Scenario Generators
  core_txn_scenario_gen core_scenario_gen;
  <...>
  ebiu_config myconfig = new(ral_model); // EBIU configuration
  memory_map mmap; // EBIU memory map
  <...>

  virtual function void gen_cfg();
    super.gen_cfg();
    if (!cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize test config");
    end
  endfunction: gen_cfg

  virtual function void build();
    super.build();
    // Instantiate Channels
    core_channel = new("core_chan", "channel");
    <...>
    // Instantiate Xactors
    core_master = new("CORE", -1, top.core.Master,
                      null, core_channel);
    <...>
    // Scenario Transaction Generators
    core_scenario_gen = new("CORE Scen Gen", -1, core_channel);
    core_scenario_gen.stop_after_n_insts = cfg.num_core_txns;
    <...>
    // Set register models to have reset values
    // Randomize EBIU memory configuration, RAL model,
    // Set DDR refresh & other memory parms, display config
    // Initialize memory map object
    <...>
    // Go ahead and start the transactors here.
    core_master.start_xactor();
    <...>
  endfunction: build

  virtual task reset_dut();
    super.reset_dut();
  endtask: reset_dut

  virtual task hw_reset();
    // Apply hw reset to DUT. Called by super.reset_dut()
    // Control reset line, connect testbench memories
    <...>
  endtask: hw_reset

```

```

virtual task cfg_dut();
    vmm_rw::status_e status;
    super.cfg_dut();
    // Turn off AMC, reset DDR cntrllr, wait for DDR init
    // Program EBIU registers, turn AMC back on, program BURST
    // on core bus, etc.
    <...>
endtask: cfg_dut

virtual task start();
    super.start();
    core_scenario_gen.start_xactor();
    <...>
endtask: start

virtual task wait_for_end();
    super.wait_for_end();
    fork
        begin // Timeout
            repeat (timeout) @(posedge top.sclk);
            `vmm_fatal(log, "Testbench Timeout");
        end
        begin // Transaction Generation
            core_scenario_gen.notify.wait_for
                (core_txn_scenario_gen::DONE);
            <...>
        end
    join_any
endtask: wait_for_end

virtual task stop();
    super.stop();
    core_scenario_gen.stop_xactor();
    <...>
    // Let the DUT drain off all pending data
    repeat (20) @(top.sclk);
    core_master.stop_xactor();
    <...>
endtask: stop

```

Figure 9 – vmm_env Class

URM also has a simulation flow with distinct steps, similar to VMM. In VMM the top-level vmm_env objects governs this entire flow, however, in URM this can be a much more distributed process. Simulation phases are called in sequence not only at the very highest level of the environment, but within the test object itself, and continuing through all the levels of hierarchy down into the BFMs. The outcome of this is that lower level subunits can all have their own set of build and configuration steps, which can be integrated into these units and abstracted from higher levels of hierarchy. Where as in VMM, to bring in multiple subunits you would have to manually merge all build and configuration steps into the same top-level environment, with URM these steps might be abstracted into each individual sub-unit.

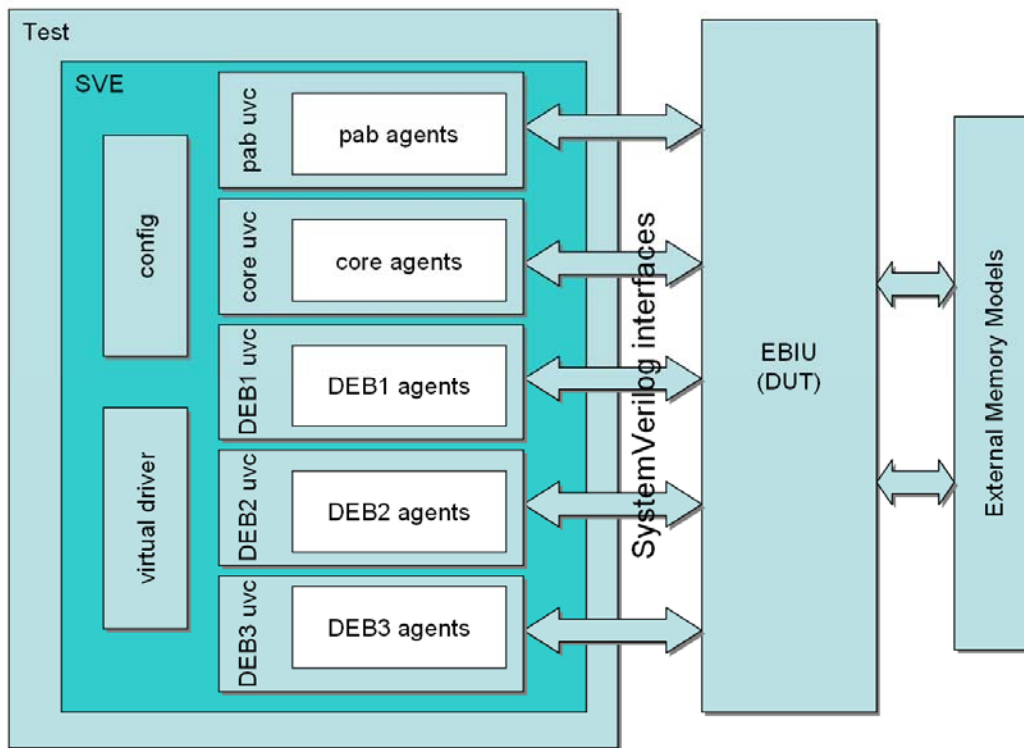


Figure 10 - EBIU URM Testbench

A comparison of the EBIU testbench implementation in both methodologies shows that the URM version has several more levels of hierarchy than the VMM version. The motivation for having additional levels of abstraction is the hope that it makes the individual pieces more flexible, and easier to reuse in the future. These additional layers, not found in the VMM implementation of the EBIU, include agents, UVCs, and the SVE. (See Figure 10 for URM block diagram & Figure 13 for UVC detail). These classes implement levels of abstraction which are recommended by the URM methodology, leveraging proven eRM techniques. The URM class library is very flexible, however, and these divisions are not a requirement of the class library itself. In the current version of the URM library, each of these levels of abstraction are implemented using a `urm_unit` class, the same class which is used for the individual BFM.

5.1 Agent Class

The agent class, extended from `urm_unit`, contains a BFM (either master or slave), a monitor, and a driver. The driver is what generates stimulus for the BFM, functionally it does the same thing as the `<classname>_atomic_gen` and `<classname>_scenario_gen` classes in VMM.

```

class adi_deb_master_driver extends urm_driver;
  // Note that the driver has a special macro. As well as
  // the create() and get_type_name() function
  // definitions it creates the infrastructure to generate
  // combinations of sequences and items
  `urm_driver_utils(adi_deb_master_driver)

  function new(string name = "", urm_unit parent = null);
    super.new(name, parent);
    // Note that this macro is required here.
    `urm_driver_set_sequences_and_item(deb_txn)
  endfunction // new
endclass

```

Figure 11 – urm_driver Class for DEB Master

Unlike the VMM transaction generators, the URM driver is not generated automatically by macros, and must be implemented manually. Most of the functionality is in the urm_driver base class, which is why the code in Figure 11 looks very basic. The parent class will handle all the real work, such as sequence selection, sending transactions to the BFM, synchronization, and more. The sequences are the common test scenarios, which will be assigned to the drivers, and direct how the transactions are generated.

```

class adi_deb_master_agent extends urm_unit;
  `urm_unit_utils(adi_deb_master_agent)
  bit is_active = 1;
  adi_deb_master_bfm bfm;
  adi_deb_master_driver driver;
  adi_deb_master_monitor monitor;
  // Constructor.
  function new(string name, urm_unit parent = null);
    super.new(name, parent);
  endfunction

  function void build();
    super.build();
    $cast(monitor, create_unit("adi_deb_master_monitor",
                              "monitor"));

    monitor.build();
    if(is_active == 1) begin
      $cast(driver, create_unit("adi_deb_master_driver",
                              "driver"));

      driver.build();
      $cast(bfm, create_unit("adi_deb_master_bfm", "bfm"));
      bfm.build();
      bfm.driver = this.driver;
    end
  endfunction : build
  function void assign_vi(virtual interface adi_deb_if vif);
    monitor.deb = vif;
    if (is_active == 1) begin
      bfm.deb = vif;
    end
  endfunction
endclass // adi_deb_master_agent

```

Figure 12 – Agent Class for DEB Master BFM

5.2 UVC (Universal Verification Component)

At Analog Devices' Austin Design Center, whenever a BFM is developed for testbench purposes, it's considered good design practice for the engineer to develop a master, slave and monitor BFM. This is true, even though the immediate requirement may be for only one of these components. One reason for this is that it's often easier to do the initial test and debug of a BFM in an environment with both a master and slave communication to each other. Another reason is that it's usually not a lot of incremental effort to design all three of these at the same time. There's a good chance that somebody will be doing verification on the piece that connects to yours, and when that happens, they'll want to make use of the other half for their own testbench.

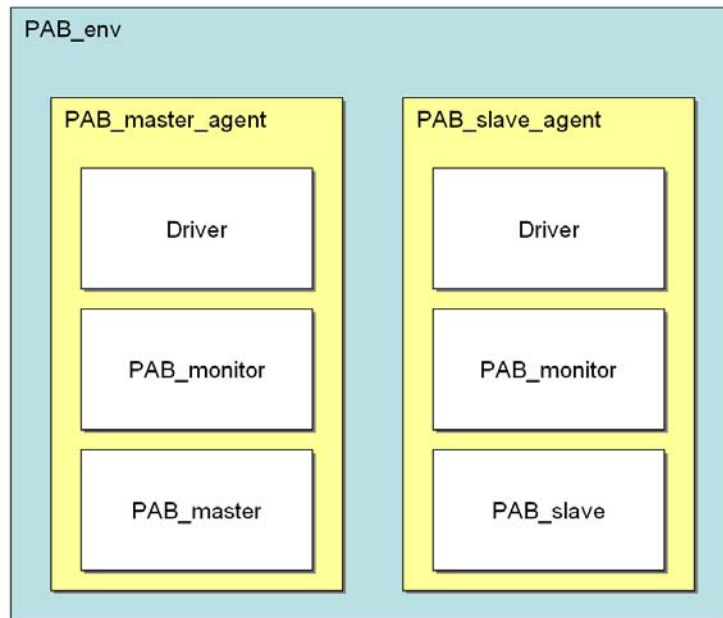


Figure 13 - UVC Block Diagram for the PAB Bus

The URM methodology formalizes this practice into part of the methodology by bundling the slave, master and monitor classes into one flexible UVC object. The idea behind this is to create a highly configurable object that can be instantiated, and then configured to play whatever role is needed on a particular bus. A UVC might even be configured to contain multiple masters or slaves, if the bus protocol allows this. See Figure 13 for a block diagram of the UVC for one of the EBIU system busses. Note that the connection between the driver (transaction generator) and the BFM is already made within this block, and does not need to be managed in the top-level environment.

```

class pab_env extends urm_unit;
  protected bit has_bus_monitor = 0;
  protected int unsigned num_masters = 0;
  protected int unsigned num_slaves = 0;
  pab_bus_monitor bus_monitor;
  pab_master_agent masters[];
  pab_slave_agent slaves[];

  // this macro provides get_type_name() and create()
  `urm_unit_utils_begin(pab_env)
    `urm_field_int(has_bus_monitor, ALL_ON)
    `urm_field_int(num_masters, ALL_ON)
    `urm_field_int(num_slaves, ALL_ON)
  `urm_unit_utils_end

  // Constructor
  function new(string name, urm_unit parent);
    super.new(name, parent);
  endfunction

  function void build();
    string inst_name;
    super.build();
    if(has_bus_monitor == 1) begin
      $cast(pab_bus_monitor, create_unit("pab_bus_monitor",
                                         "bus_monitor"));

      pab_bus_monitor.build();
    end
    masters = new[num_masters];
    for(int i = 0; i < num_masters; i++) begin
      $sformat(inst_name, "masters[%0d]", i);
      $cast(masters[i], create_unit("pab_master_agent",
                                   inst_name));

      masters[i].build();
    end
    slaves = new[num_slaves];
    for(int i = 0; i < num_slaves; i++) begin
      $sformat(inst_name, "slaves[%0d]", i);
      $cast(slaves[i], create_unit("pab_slave_agent",
                                   inst_name));

      slaves[i].build();
    end
  endfunction : build
endclass

```

Figure 14 – UVC Code for the PAB

5.3 SVE (Simulation & Verification Environment)

The SVE is the level of abstraction that most closely resembles the vmm_env class. Code at the Agent and UVC level is intended to be bus specific, but not design specific. The SVE is a convention from eRM for encapsulating all of the verification components for a specific DUT.

```

class main_sve extends urm_unit;

    // Provide implementations of virtual methods such as
    // get_type_name and create
    `urm_unit_utils(main_sve)
    adi_core_env core0;
    adi_deb_env  deb0;
    adi_deb_env  deb1;
    adi_deb_env  deb2;
    pab_env      pab0;
    //Virtual sequence driver to control PAB, CORE, and DEB UVC's
    ebiu_virtual_driver v_driver;

    // Constructor calls super.new() with appropriate parameters.
    function new (string name="", urm_unit parent=null);
        super.new(name, parent); // Super must be called first
    endfunction // new

    virtual function void build();
        super.build();
        set_config_int("core0", "num_masters", 1);
        set_config_int("deb0", "num_masters", 1);
        set_config_int("deb1", "num_masters", 1);
        set_config_int("deb2", "num_masters", 1);
        set_config_int("pab0", "num_masters", 1);
        $cast(core0, create_unit("adi_core_env", "core0"));
        $cast(deb0, create_unit("adi_deb_env", "deb0"));
        $cast(deb1, create_unit("adi_deb_env", "deb1"));
        $cast(deb2, create_unit("adi_deb_env", "deb2"));
        $cast(pab0, create_unit("pab_env", "pab0"));
        core0.build();
        deb0.build();
        deb1.build();
        deb2.build();
        pab0.build();
        core0.masters[0].assign_vi(main_sve_top.adi_core0);
        deb0.masters[0].assign_vi(main_sve_top.adi_deb0);
        deb1.masters[0].assign_vi(main_sve_top.adi_deb1);
        deb2.masters[0].assign_vi(main_sve_top.adi_deb2);
        pab0.masters[0].assign_vi(main_sve_top.pab);
    endfunction

endclass

```

Figure 15 – SVE Class

Figure 15 shows an example of the SVE class for the EBIU testbench.

5.4 Build() Phase

One of the advantages of a class-based approach to a random testbench is that the environment itself can be very dynamic. Based on parameters specified or generated at runtime, the testbench can be constructed in vastly different ways depending on the numbers and types of dynamic objects which are created, configured, and connected.

In a VMM environment, the purpose of the `gen_cfg` step of the `vmm_env` object is to generate the parameters which will determine the configuration. The next step in the VMM flow is `build()`, where objects are created and configured based on the desired configuration. With VMM, all of the configuration and build steps are visible together in one place. The parameters determined in the configuration phase can be used to direct the build phase. The disadvantage becomes apparent when merging together multiple subunits into a larger testbench, as the user is required to merge each of the steps from separate testbenches into a new `vmm_env` object.

URM provides a method to bundle the configuration and build steps into the lower-level class objects. This provides a hierarchical build process which should lend itself well to being portable, since UVCs could potentially contain other UVCs. In URM, a `urm_unit` object should never be created with a call to `new()`. Instead, a `urm_unit` object should be created by first calling `create_unit()`, followed by a call to the objects' `build()` method. (See Figure 12 & Figure 15 for examples.) If a `urm_unit` contains other `urm_unit` classes, it is the responsibility of the parent class to make sure that each of the children's `build()` commands are called within the parent `build()` routine using the above procedure.

One of the reasons `build()` is used rather than `new()` is because this mechanism allows configuration parameters to be specified prior to the creation of the objects. See Figure 15 for an example where the `set_config_int` macro is used to specify that each of the UVC's used in the EBIU testbench environment will be created with exactly 1 master BFM. Testbench developers can specify their own configuration parameters through the use of the `urm_unit_utils` macros (see Figure 14 for an example of this).

One of the side benefits of going to the trouble of using all of these macros to define testbench objects and configuration parameters is that URM keeps track of all of these relationships within the testbench environment. The entire testbench hierarchy, with all of its configuration parameters, can be displayed by a single call to the `print()` function of the top-level SVE. See Figure 16 for a partial listing of the EBIU testbench hierarchy.

Name	Type	Size	Value
sve0	main_sve	-	@1221
deb0	adi_deb_env	-	@1007
masters[0]	adi_deb_master_age+	-	@972
monitor	adi_deb_master_mon+	-	@999
logging	integral	1	'h0
driver	adi_deb_master_dri+	-	@991
default_sequence	string	17	urm_main_sequence
pull_mode	integral	1	'h1
sequences	da(string)	6	-
[0]	string	17	urm_main_sequence
[1]	string	19	urm_random_sequence
[2]	string	19	urm_simple_sequence
[3]	string	24	adi_deb_master_use+
[4]	string	12	deb_do_write
[5]	string	16	read_after_write
count	integral	32	-1
max_random_count	integral	32	'd10
max_random_depth	integral	32	'd4
bfm	adi_deb_master_bfm	-	@980
allow_delay_iss+	integral	1	'h0
driver	adi_deb_master_dri+	-	@991
has_bus_monitor	integral	1	'h0
num_masters	integral	32	'h1
num_slaves	integral	32	'h0
core0	adi_core_env	-	@895
masters[0]	adi_core_master_ag+	-	@860
monitor	adi_core_master_mo+	-	@887
driver	adi_core_master_dr+	-	@879
default_sequence	string	17	urm_main_sequence
pull_mode	integral	1	'h1

<...>

Figure 16 – Top-level print() Output

This is a very helpful feature for understanding the current testbench configuration.

5.5 Configuration of the DUT

Within the top-level VMM environment, each step in the overall simulation flow is run in sequence. Many of these simulation phases are tasks, which can consume time, rather than functions which cannot. The EBIU VMM testbench uses the `reset_dut()` step to control the reset process, and make connections in the testbench from the DUT to the proper memory models during reset. The `cfg_dut()` step is used to drive transactions on the system busses to program configuration registers within the DUT, as well as configuration commands to any external flash memories which may have been selected. Once this is taken care of, the scenario generators are turned on in the `start()` phase, and the test progresses as normal.

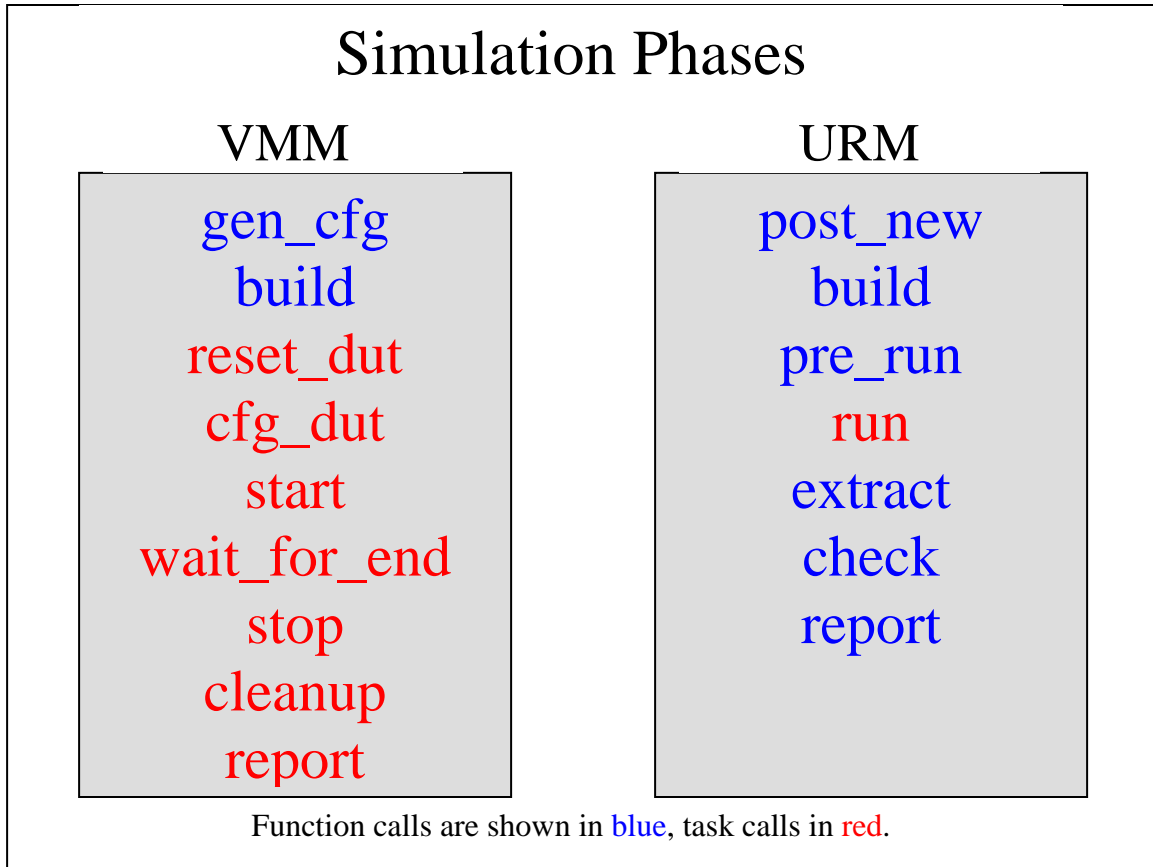


Figure 17 - Simulation Phase Comparison

While the top-level SVE block has similar simulation flow steps, the only step which is a task and not a function is the run() task. Because this is the only simulation phase which is implemented using a task, anything which consumes simulation time must be done during this step. One of the issues with simply putting everything into the run() task, however, is that by default all of the sequence drivers have already been turned on before the run() task in the SVE is called. This caused a problem in our testbench, since the EBIU had not been configured yet. VMM includes standard routines such as start_transactor() and stop_transactor() to control the behavior of BFM's and transaction generators, while the current version of URM does not..

To get around this initially, an extra layer of synchronization had to be added to all of the drivers so that their default activity could be suspended until the configuration process was complete. The concern with using this type of approach is that this is a non-standard method to control the drivers, and it negates some of the advantages of using class libraries as different engineers are sure to use different approaches.

A later release of the URM library introduced a better solution by providing support for virtual sequences. A virtual sequence is a sequence without its own data item, and is used to control other sequences and sequence drivers. This allows a virtual driver to be instantiated in the top-level SVE which acts as a master controller for the test. This driver is able to execute a series of sequences which drive the initial reset, do all of the

transactions to configure the DUT, and then initiate simultaneous random transactions on the system busses for the duration of the test.

Converting all of the configuration routines to a sequence still presented some challenges. Following recommended methodology, the drivers and the BFM within an agent are connected to one another directly. Within the VMM testbench, various configuration routines were written to pass transaction objects directly to BFMs through `vmm_channel` objects, which are easily accessible within the top-level `vmm_env` environment. This type of behavior can be more difficult to recreate in the URM testbench as the only way to access the BFM is indirectly through sequence objects assigned to a driver.

6 Testing the DUT

6.1 Sequences & Scenarios

In VMM groups of related random transactions are referred to as scenarios, and are generated by `scenario_generator` objects. The default VMM scenario generator for a `vmm_data` transaction class is usually defined automatically through the use of a macro. This default scenario generator comes pre-populated with a single scenario type which generates a stream of single unconstrained randomized transactions. Testbench writers can develop more sophisticated scenario modules, and add these to the scenario generator. By default, the VMM scenario generator will step through each of its configured scenarios in a round-robin fashion, though this behavior can be overridden.

```
class read_after_write extends deb_txn_scenario;
  int unsigned scenario_id;

  function new();
    scenario_id = define_scenario(.name("Read After Write"),
                                 .max_len(2));
  endfunction: new

  constraint c_valid {
    repeated == 5;
    length == 2;
    items[0].transtype == aus_data::WRITE32;
    items[1].transtype == aus_data::READ32;
    items[0].address == items[1].address;
  }

endclass: read_after_write
```

Figure 18 – VMM Test Scenario

```

class read_after_write extends urm_sequence;
  //READ after WRITE (same address)

  `urm_sequence_utils(read_after_write, adi_deb_master_driver)

  function new(string name="read_after_write",
               urm_object parent=null);
    super.new(name);
  endfunction

  deb_txn read_txn;
  rand int read_count;

  virtual task body();
    logic [31:0] first_address, next_address;
    read_count = 5;

    for (int i=0; i<read_count; i++) begin
      `urm_do_with(read_txn,
                  {read_txn.transtype == aus_data::WRITE32;})
      first_address = read_txn.address;
      `urm_do_with(read_txn,
                  {read_txn.address == first_address &&
                   read_txn.transtype == aus_data::READ32;})
    end
  endtask
endclass: read_after_write

```

Figure 19 – URM Test Sequence

The URM version of a scenario is called a sequence, derived from a `urm_sequence` object. The URM version of the `scenario_generator` is a driver object which is extended from `urm_driver`. In the EBIU testbench, one driver object was embedded into each of the UVCs, and another virtual driver was embedded into the top-level SVE. Similar to the VMM scenario generator, the default driver object contains a pre-defined sequence which will generate a stream of unconstrained random transactions. More elaborate sequences can be developed by creating classes derived from the `urm_sequence` object.

Figure 18 & Figure 19 show a VMM scenario and a URM sequence which accomplish roughly the same thing. One of the biggest differences between the two approaches is that the VMM scenario returns an array populated with all of the random transactions from the scenario. This happens all at once in zero time, where as the URM sequence generator is allowed to consume time while it generates its random transactions one at a time. This gives the URM sequences the flexibility to be extremely long, without potential storage constraint issues. The URM sequence object can also easily embed other sequences within the body of its own sequence, allowing the capability to produce very complex nested sequences. Because the sequence generator can consume time, it would also be conceivable to incorporate feedback from the simulation to direct the transaction generation. Sequences are used by the virtual driver in the top-level SVE to do “system” things, such as drive reset, configure the DUT, and synchronize simultaneous events on multiple busses.

6.2 urm_test Objects

In a VMM testbench, a test is usually a SystemVerilog program block which is instantiated in the top-level module. Within the program block the top-level environment is created, and the simulation flow initiated. URM behaves in much the same way, but rather than a program block, a test is simply a class which is extended from the urm_test object. Somewhere in the top-level module, a call to the global run_test() task should be made from within an initial block. The test class itself should include a build() function to build the SVE environment class. The run() task of the test class should be used to determine when the end of the test occurs, and call global_stop_request() to end normal simulation.

```
class test1 extends urm_test;
  `urm_unit_utils(test1)

  main_sve sve0;

  function new(input string name = "test1",
              input urm_unit parent=null);
    super.new(name, parent);
  endfunction

  virtual function void build();
    super.build();
    $cast(sve0, create_unit("main_sve", "sve0"));
    sve0.build();
  endfunction

  task run();
    #10000;
    `message(LOW, ("User activated end of simulation"))
    global_stop_request();
  endtask

endclass
```

Figure 20 – Simple Test

A big advantage that the URM methodology gains by defining tests as classes, is that multiple tests can all be compiled into the simulation model at the same time. Individual tests can then be selected at runtime through the use of a simple command-line argument. In addition to the convenience of having multiple tests compiled into the same simulation, this could be a huge time saver in a large environment which requires long compilation times.

7 Conclusion

Some aspects of the testbench translation were very straight-forward, and proceeded very quickly. Translating the transaction objects from vmm_data to urm_sequence_item objects is a good example of this, where code was used directly, or even removed because of the ability of the URM macros to generate required routines automatically.

While there was extra work in setting up the macros for many of the classes, the payoff came with nicely formatted tables and the built-in ability to print testbench hierarchies.

Translation of BFMs for each of the system busses was surprisingly straight-forward, with very few changes to the majority of the code. Most of the work was centered around changing the logging messages, and adapting the main control loop to work with a `urm_driver` rather than a `vmm_channel`.

Development of the agents and the UVCs was more time consuming. This was not because there was a lot of code which had to be written, but because it introduced new layers of abstraction which did not exist previously in the VMM version of the testbench. The EBIU testbench used only master BFMs on all five system busses, and creating UVCs for these busses might have taken even longer if slave and monitor agents hadn't already been created. Bundling all of these components into a UVC worked well as it integrated all of the modules developed for a particular bus into a single package.

Transaction generation also ported quite readily from VMM to URM, as URM appears to have a much more flexible approach in this area.

The biggest effort in porting the EBIU testbench was in porting all of the routines to do the initial configuration of the DUT. All of the procedures which consumed time during the configuration process had to be converted to sequences for the top-level virtual driver. Testbenches such as the EBIU, which rely on using `vmm_channel` objects directly, require additional work because of the relative difficulty of driving single transactions directly.

The majority of the testbench conversion was very straight-forward. Some styles of testbenches, especially those that require little configuration, would port very quickly. Other aspects of the testbench conversion required more effort, but in each case a suitable solution was found. Some difficulties could also be avoided in the future by partitioning the testbench in a manner more suitable to the URM methodology.

8 Acknowledgements

I would like to acknowledge both Phu Huynh and Terry Lyons from Cadence, without whose help in coding and answering countless questions, this paper would not have been possible.