2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Having Your Cake and Eating It Too

Programming UVM Sequences with DPI-C

Rich Edelman, Siemens, Fremont, CA, US (rich.edelman@siemens.com)

Tomoki Watanabe, Siemens Japan, Tokyo, Japan (tomoki.watanabe@siemens.com)

***Abstract*—Blending SystemVerilog UVM and SystemVerilog DPI-C is a powerful way to create or reuse pre-existing verification environments. This paper describes the mechanisms and methods and syntax needed, including writing tasks and functions in both the SystemVerilog interface and the UVM sequences.**

Keywords—SystemVerilog UVM, DPI-C, UVM sequences, functional verification

I. INTRODUCTION

Many verification environments today are developed using SystemVerilog [1] UVM [2]. They include UVM sequences, sequence items, environments, and tests. Many other verification environments today are developed using C. Often the C environments are used initially for architectural development and performance modeling. Unfortunately, those C verification environments – really "workloads" must be redesigned or rewritten for an RTL or gate-level simulation testbench. This paper explains a way to have both UVM SystemVerilog and C – and enjoy the benefits of both.

In this paper, we take an existing UVM verification environment [3] and add code in various places to achieve our C test programs. The added code is denoted by being in a "box" in the source code listings.

II. THE UVM

Using the UVM involves many steps. Normally a verification architect designs the testbench architecture and devises the interfaces. Then a second, larger team writes tests that exercise the DUT by using the testbench architecture and the various interfaces. Additionally, the testbench checks for correctness using coverage and checkers.

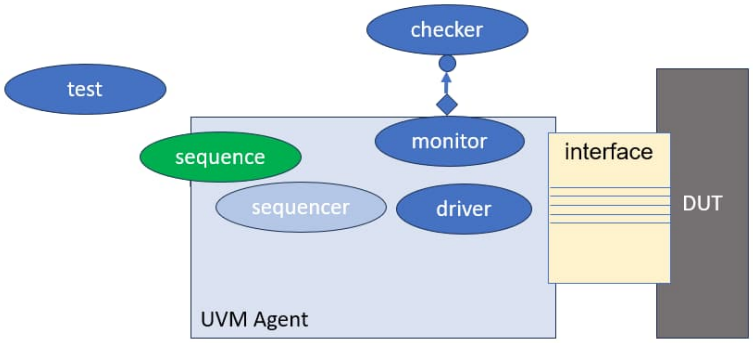

Figure 1: Typical UVM Agent Used in Verification on an Interface

In this paper, an example UVM testbench will be designed with multiple interfaces. Those interfaces will have APIs – like read() and write(). Any API that is supported could be used - postedwrite(), read_register(), write_register(), etc. These APIs are really helper tasks or functions or transactions that the verification system can use to encapsulate larger or more complex sets of functionality.

A randomized testbench can be written strictly in SystemVerilog that exercises these interfaces (APIs) and the DUT underneath, but two things make the approach of this paper more helpful. First, a random testbench isn't

always exactly what is needed. Often a "program" needs to be written that programs the registers and loads memory a specific way. Second, there are many more C programmers available than SystemVerilog UVM programmers. Randomization is one key to effective tests – the ability to create more tests simply by changing the seed value of the randomization. Our SystemVerilog UVM plus SystemVerilog DPI-C testbench will have randomization and will also reuse the pre-existing C tests. One way to think about how it might be useful is to let the UVM randomized environment generate "random traffic". Then the C test program runs at the same time – testing deadlocks and delays due to the random traffic.

III. USING DPI-C

SystemVerilog DPI-C [4] is straightforward and used widely to interface C code SystemVerilog. The C code can call SystemVerilog tasks and functions, passing arguments and getting return values. And the reverse is true – SystemVerilog code can call C functions, passing arguments and getting return values.

But when used with classes there's one problem. The DPI-C calls must be rooted in either a module, an interface or a file scope. But our approach considers marrying the class based UVM with DPI-C.

In this paper, the approach to using DPI-C is to define an interface which hosts the DPI-C calls. It serves no other purpose. Then, that interface is instantiated and referenced as a "virtual interface" handle in the UVM class code. The interface operates as the required DPI-C "location" and the C code can call sequence tasks and functions and return values. Additionally, the sequence code can call C code (again through the interface).

IV. THE SYSTEMVERILOG INTERFACE

The SystemVerilog interface design is relatively simple. It will contain the tasks and functions needed to interact with the driver. This C interface will be used from the sequence to call C code and used from C code to call tasks and functions in the sequence.

A sequence is a program that issues transactions to the driver. Using C is the same – it is a program that will call tasks or functions in the SV interface, which in turn call tasks and functions in the sequence. Think of the SV interface as "bonded" to the UVM sequence class.

In the code block below the 'zinterface' is defined. It is a SystemVerilog interface, but it has no "pins", no "ports", no "wires" or "bus". It exists for one purpose – it exists to define DPI-C tasks and functions and to register sequences who are using the interface to call C code or be called by C code.

The interface must have an associative array, indexed by a simple integer. Each element in this array holds a sequence handle (derived from uvm_sequence).

In this interface, there are two API calls that are "exported" or implemented on the SystemVerilog side. They are 'read()' and 'write()'. On the C side, there is one task named zinterface_start_test_program1 in C. Since there may be many interfaces of different types loaded, and since C does not build a namespace, please define these kinds of calls with a unique prefix. (zinterface). From the SystemVerilog side that same routine is known as start_test_program1. Since it is being used inside the zinterface, that suffices as a namespace.

The register() routine takes a simple integer – the thread id and a sequence handle. It loads the associative memory and then it assigned this interface into the handle in the sequence. The sequence uses this to call routines in the interface – to start the test program for example.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

import ip_pkg::*;

interface zinterface();
  uvm_object registered_seq[int];

  export "DPI-C" task read;
  export "DPI-C" task write;
```

```
  import "DPI-C" context zinterface_start_test_program1 =
    task start_test_program1(int index, string name, int start_addr);

  function void register(int index, uvm_object seq);
    zinterface_zombieseq zsq;
    registered_seq[index] = seq;
    $cast(zsq, seq);
    zsq.vif = interface::self(); // Extension to the LRM
  endfunction

  task read(int index, int addr, output int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.read(addr, data);
  endtask

  task write(int index, int addr, int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.write(addr, data);
  endtask
endinterface
```

The read() and write() routines in the interface are nothing more than shells. They "trampoline" over to the SystemVerilog side by looking up the sequence handle for this thread and calling the read() or write() task or function implemented in the sequence.

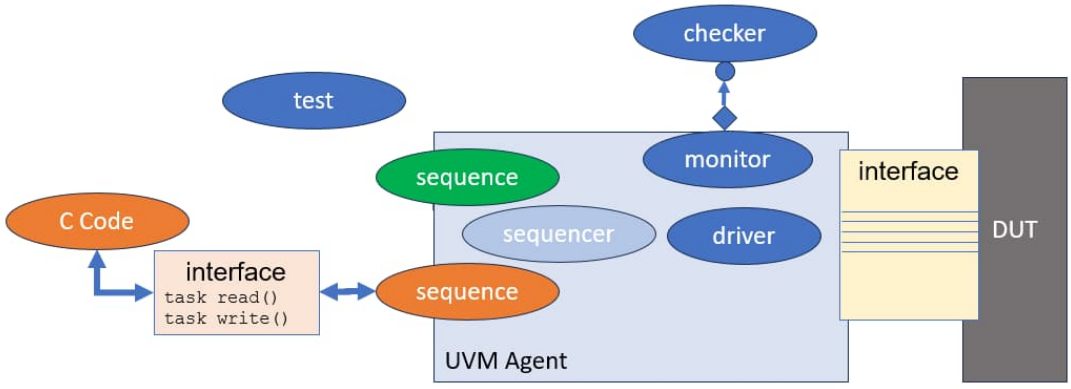

Figure 2: Adding the Interface and C code

V. The UVM Zombie Sequence

The basic UVM sequence used in this paper is a normal UVM sequence, but it runs in a very special way. The sequence defines tasks and functions that eventually call start_item() and finish_item() – generating transactions for the driver. But the body() of this sequence doesn't necessarily call those tasks and functions, nor does it call start_item() and finish_item(). Instead, it operates as a "zombie" – the body() call starts, and never ends – it waits forever. This leaves the connection to the sequencer and the driver open and available for calls to start_item() and finish_item().

This behavior is "normal UVM" – just a unique way to use a sequence – not used that frequently in an exclusively SystemVerilog based testbench – but very effective for our C based tests to interact with SystemVerilog UVM.

VI. The Sequence Code

This sequence zinterface_zombieseq has a prefix zinterface for the interface type – it is a "z" interface. And also, a prefix 'zombie' to remind that it is not a regular sequence. It is to be used in a special way.

It has a virtual interface handle to the zinterface – 'vif'.

The body() is defined and has a convenience setting called 'running' which allows a potential caller to check if the sequence is "running" – and ready to have transactions generated from it. The body() function does its waiting by waiting for done to be 1. This typically would never happen, but it could be used to shut the sequence done – end – by setting done to 1.

The next three tasks are the API that this sequence implements in support of the z interface and the test program. First the routine 'start_test_program' is defined. The first argument is the thread number. Then a name and a start address. There could be any arguments required by the testprogram. Then the C code is called by bouncing through the interface with vif.start_test_program1().

The read and write routines are accessed from C – they will be called by C through the interface. They do typical 'sequence' things. They call new() to create a transaction and fill in the attributes. Then they call start_item() and finish_item(). Read returns the data value read through an output port on the task.

```
class zinterface_zombieseq extends seq;
  `uvm_object_utils(zinterface_zombieseq)

  virtual zinterface vif;

  function new(string name = "zinterface_zombieseq");
    super.new(name);
  endfunction

  bit done;    // Kill the zombie sequence by setting to 1;
  bit running; // Let's the tasks and function know that it is safe to run

  task body();
    running = 1;
    wait(done == 1);
  endtask

  task start_test_program1(int index, string name, int start_addr);
    vif.start_test_program1(index, name, start_addr);
  endtask

  task read(bit [31:0] addr, output bit [31:0]data);
    if (running == 0) wait (running == 1); // Quick test to make sure it is safe to run
    t = transaction::type_id::create("read");
    t.rw = READ;
    t.addr = addr;
    t.data = 0;

    start_item(t);
    finish_item(t);

    data = t.data;
  endtask

  task write(bit [31:0] addr, bit [31:0]data);
    if (running == 0) wait (running == 1); // Quick test to make sure it is safe to run
    t = transaction::type_id::create("write");
    t.rw = WRITE;
    t.addr = addr;
    t.data = data;

    start_item(t);
    finish_item(t);
  endtask
endclass
```

VII. The C Code – The Test Program

This is the C test program. For this paper it is simple. It is a collection of loops calling read() and write(). Those reads and writes are the read and write exported in the interface, and implemented there to call our UVM sequence code. This test program takes a start address, and writes 10 values, then reads them back checking that the expected and actual match. Then it repeats that for the next set of 10 addresses.

This is a self-checking test, and quite simple. But it will be invoked 4 different times in this example. One time for each interface. There will be 4 SystemVerilog threads running calling through this code at one time. This means this code must be "thread-safe". Generally, it is a good idea to write such code, but in this case it is required. Treat this C code like it is threaded and running on a parallel computer. SystemVerilog models parallel computation, but there is really on one thing executing at a time. (like two initial blocks running "in parallel"). But this C code must be written in a thread-safe way.

```
#include <stdio.h>
#include "dpiheader.h"

int
zinterface_start_test_program1(int index, const char *name, int start_addr) {
    int addr;
    int data;
    int original_start_addr;
    int dataloops;
    int loops;

    original_start_addr = start_addr;

    // Repeat 10 times, changing the data
    for (dataloops = 0; dataloops < 10; dataloops++) {
      // Repeat 10 times - writing 10, and reading 10
      start_addr = original_start_addr;
      for (loops = 0; loops < 10; loops++) {
        for (addr = start_addr; addr < start_addr+10; addr++) {
          data = addr + 1000 + dataloops;
          write(index, addr, data);
        }

        for (addr = start_addr; addr < start_addr+10; addr++) {
          read(index, addr, &data);

          if (data != addr + 1000 + dataloops) {
            printf("C: ...ERROR  READ (%0d, %0d) <%s> [wrote: %d, read %d]  \n",
              addr, data, name, data, addr + 1000 + dataloops);
          }
        }

        start_addr += 10;
      }
    }

    return 0;
}
```

VIII. The Test

The test has a few updates.

The first is declaring 4 of the special sequences – they will each by started on a different agent and run a different test program thread.

The second update is a little paperwork – the interface – the zinterface instance must be retrieved from the config database. There is only one z interface needed for N instances on the DUT. This one interface defines the

programming interface and is thread-safe and simple to inspect for problems. It manages multiple connections or threads with the associative array of registered sequences.

The third and fourth update construct the zinterface zombie sequences. And then the zinterface registration is called so that the zinterface instance has its sequences and the sequences have a pointer to the interface.

The fifth update starts those new zombie sequences on the appropriate sequencers.

The sixth and final update calls the C code to start the C test program. This is done in a fork/join just like the regular sequences and the zombie sequences. This is all running in "parallel" – generating traffic and executing a test program.

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

  env e1, e2, e3, e4;
  seq s1, s2, s3, s4;

  zinterface_zombieseq zs1, zs2, zs3, zs4;

  virtual zinterface zif;

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
    e2 = env::type_id::create("e2", this);
    e3 = env::type_id::create("e3", this);
    e4 = env::type_id::create("e4", this);
  endfunction

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

    pretty_print();
    factory.print();

    s1 = seq::type_id::create("s1");
    s2 = seq::type_id::create("s2");
    s3 = seq::type_id::create("s3");
    s4 = seq::type_id::create("s4");

    if (!uvm_config_db#(virtual zinterface)::get(this, "*", "zinterface", zif))
      `uvm_fatal(get_type_name(), "cannot find ZINTERFACE_INSTANCE")

    zs1 = zinterface_zombieseq::type_id::create("zs1");
    zs2 = zinterface_zombieseq::type_id::create("zs2");
    zs3 = zinterface_zombieseq::type_id::create("zs3");
    zs4 = zinterface_zombieseq::type_id::create("zs4");

    zif.register(1, zs1);
    zif.register(2, zs2);
    zif.register(3, zs3);
    zif.register(4, zs4);

    fork
      zs1.start(e1.a.sqr);
      zs2.start(e2.a.sqr);
      zs3.start(e3.a.sqr);
      zs4.start(e4.a.sqr);
    join_none

    fork
      s1.start(e1.a.sqr);
      s2.start(e2.a.sqr);
```

```
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join
```

```
    fork
      zs1.start_test_program1(1, "thread1", 100);
      zs2.start_test_program1(2, "thread2", 200);
      zs3.start_test_program1(3, "thread3", 300);
      zs4.start_test_program1(4, "thread4", 400);
    join
```

```
    phase.drop_objection(this);
  endtask
endclass
```

IX. The Top

The top is simple for this example. The zinterface must be instantiated.

Then the top module puts a virtual interface handle to that instantiation into the config database.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

import ip_pkg::*;

module top();

  memory_interface memory_interface_instance();
```

```
  zinterface I_zinterface1();
```

```
  initial begin
    uvm_config_db#(virtual memory_interface)::set(
      uvm_root::get(), "*", "memory_interface", memory_interface_instance);
```

```
    uvm_config_db#(virtual zinterface)::set(
      uvm_root::get(), "*", "zinterface", I_zinterface1);
```

```
    run_test();
  end
endmodule
```

X. Conclusion

The reader of this paper will have all the knowledge and a working example describing how to design and build a verification environment that allows for reuse of C modeling and performance tests as part of a SystemVerilog UVM testbench. This knowledge can be applied immediately to an existing testbench – perhaps to augment a pure UVM testbench with some C programs from the performance team or perhaps as a framework for a brand new testbench and architecture.

The source code will be available as open-source download from the author.

References

[1] SystemVerilog - 800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, *https://ieeexplore.ieee.org/document/8299595*

[2] UVM - 1800.2-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual, *https://ieeexplore.ieee.org/document/9195920* , source code: *https://www.accellera.org/downloads/standards/uvm*

[3] "Easy Testbench Evolution – Styling Sequences and Drivers", Rich Edelman, DVCON Japan 2023

[4] "Using SystemVerilog Now with DPI", Rich Edelman, DVCON US 2005

XI. APPENDIX – THE CHANGED OR ADDED SOURCE CODE

```
// ================================================
// FILE: t.sv
// ================================================

import uvm_pkg::*;
`include "uvm_macros.svh"

import ip_pkg::*;

module top();

  memory_interface memory_interface_instance();

  zinterface I_zinterface1();

  initial begin
    uvm_config_db#(virtual memory_interface)::set(
      uvm_root::get(), "*", "memory_interface",
                           memory_interface_instance);

    uvm_config_db#(virtual zinterface)::set(
      uvm_root::get(), "*", "zinterface", I_zinterface1);

    run_test();
  end
endmodule


// ================================================
// FILE: zinterface.sv
// ================================================

import uvm_pkg::*;
`include "uvm_macros.svh"

import ip_pkg::*;

interface zinterface();
  uvm_object registered_seq[int];

  export "DPI-C" task read;
  export "DPI-C" task write;
  import "DPI-C" context zinterface_start_test_program1 =
    task start_test_program1(
      int index, string name, int start_addr);

  function void register(int index, uvm_object seq);
    zinterface_zombieseq zsq;
    registered_seq[index] = seq;
    $cast(zsq, seq);
    zsq.vif = interface::self(); // Extension to the LRM
  endfunction


  task read(int index, int addr, output int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.read(addr, data);
  endtask

  task write(int index, int addr, int data);
    zinterface_zombieseq zsq;
    $cast(zsq, registered_seq[index]);
    zsq.write(addr, data);
  endtask
endinterface
```

```
// ================================================
// FILE: test.svh
// ================================================

class test extends uvm_test;
  `uvm_component_utils(test)

  env e1, e2, e3, e4;
  seq s1, s2, s3, s4;

  zinterface_zombieseq zs1, zs2, zs3, zs4;

  virtual zinterface zif;

  function new(string name = "test",
      uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
    e2 = env::type_id::create("e2", this);
    e3 = env::type_id::create("e3", this);
    e4 = env::type_id::create("e4", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)

    pretty_print();
    factory.print();

    s1 = seq::type_id::create("s1");
    s2 = seq::type_id::create("s2");
    s3 = seq::type_id::create("s3");
    s4 = seq::type_id::create("s4");

    if (!uvm_config_db#(virtual zinterface)::get(
         this, "*", "zinterface", zif))
      `uvm_fatal(get_type_name(),
        "Cannot find ZINTERFACE_INSTANCE")

    zs1 = zinterface_zombieseq::type_id::create("zs1");
    zs2 = zinterface_zombieseq::type_id::create("zs2");
    zs3 = zinterface_zombieseq::type_id::create("zs3");
    zs4 = zinterface_zombieseq::type_id::create("zs4");

    zif.register(1, zs1);
    zif.register(2, zs2);
    zif.register(3, zs3);
    zif.register(4, zs4);

    fork
      zs1.start(e1.a.sqr);
      zs2.start(e2.a.sqr);
      zs3.start(e3.a.sqr);
      zs4.start(e4.a.sqr);
    join_none

    fork
      s1.start(e1.a.sqr);
      s2.start(e2.a.sqr);
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join

    fork
      zs1.start_test_program1(1, "thread1", 100);
      zs2.start_test_program1(2, "thread2", 200);
      zs3.start_test_program1(3, "thread3", 300);
      zs4.start_test_program1(4, "thread4", 400);
    join

    phase.drop_objection(this);
  endtask
endclass
```

```
// ====================================================
// FILE: sequence.svh
// ====================================================

class seq extends uvm_sequence#(transaction);
  `uvm_object_utils(seq)
  function new(string name = "seq");
    super.new(name);
  endfunction

  transaction t;

  task body();
    string name;
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)

    for (int i = 0; i < 1000; i++) begin
      name = $sformatf("t%0d", i);
      t = transaction::type_id::create(name);
      start_item(t);
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize Failed")
      finish_item(t);
    end
  endtask
endclass
```

```
class zinterface_zombieseq extends seq;
  `uvm_object_utils(zinterface_zombieseq)

  virtual zinterface vif;

  function new(string name = "zinterface_zombieseq");
    super.new(name);
  endfunction

  bit done;   // Kill the zombie sequence by setting to 1;
  bit running;// Lets the tasks and function know
              //   that it is safe to run

  task body();
    running = 1;
    wait(done == 1);
  endtask

  task start_test_program1(
      int index, string name, int start_addr);
    vif.start_test_program1(index, name, start_addr);
  endtask

  task read(bit [31:0] addr, output bit [31:0]data);
    // Quick test to make sure it is safe to run
    if (running == 0) wait (running == 1);
    t = transaction::type_id::create("read");
    t.rw = READ;
    t.addr = addr;
    t.data = 0;

    start_item(t);
    finish_item(t);

    data = t.data;
  endtask

  task write(bit [31:0] addr, bit [31:0]data);
    // Quick test to make sure it is safe to run
    if (running == 0) wait (running == 1);
    t = transaction::type_id::create("write");
    t.rw = WRITE;
    t.addr = addr;
    t.data = data;

    start_item(t);
    finish_item(t);
  endtask
endclass
```

```
// ====================================================
// FILE: testprogram.c
// ====================================================
#include <stdio.h>
#include "dpiheader.h"

int
zinterface_start_test_program1(
    int index, const char *name, int start_addr) {
  int addr;
  int data;
  int original_start_addr;
  int dataloops;
  int loops;

  original_start_addr = start_addr;

  // Repeat 10 times, changing the data
  for (dataloops = 0; dataloops < 10; dataloops++) {

    // Repeat 10 times - writing 10, and reading 10
    start_addr = original_start_addr;
    for (loops = 0; loops < 10; loops++) {
      for (addr = start_addr; addr < start_addr+10;
          addr++) {

        data = addr + 1000 + dataloops;

        write(index, addr, data);
      }

      for (addr = start_addr; addr < start_addr+10;
          addr++) {

        read(index, addr, &data);

        if (data != addr + 1000 + dataloops) {
          printf(
           "C: ...ERROR  READ (%0d, %0d) ...\
             <%s> [wrote: %d, read %d]  \n",
           addr, data, name,
           data, addr + 1000 + dataloops);
        }
      }

      start_addr += 10;
    }
  }
  return 0;
}
```