UVM Based Testbench Architecture for Unit Verification

Juan Francesconi, J. Agustin Rodriguez, Pedro M. Julián
Instituto de Investigaciones en Ing. Electrica (IIIE) Alfredo Desages (UNS-CONICET)
Depto. de Ing. Eléctrica y de Computadoras, Universidad Nacional del Sur
Av. Alem 1252, (8000) Bahía Blanca, Argentina
Email: {juan.francesconi, jrodriguez}@uns.edu.ar

***Abstract*—In this work, the *Universal Verification Methodology* (UVM) is analyzed through its application in the development of two testbenches for unit verification. The first one targets a *First Input-First Output (FIFO) buffer* module and employs all the basic UVM components; a scoreboard with a Reference Model and a Functional Coverage collector are also implemented. The second one verifies an *I2C EEPROM* slave module; a bus functional model for the *I2C* protocol is defined to facilitate the driver implementation, rising the level of abstraction and allowing the reuse of the verification component for other I2C devices.**

I. Introduction

The *Universal Verification Methodology* (UVM) [1] represents the latest member of a family of methodologies and their associated base class libraries for using *SystemVerilog* (SV) [2] for functional verification [3] of digital hardware.

SV based methodologies play a valuable role in the development of verification environments. They define conventions for the structure and dynamics of a testbench, capturing best practices and avoiding the need to reinvent the mechanisms needed to use SV classes to build verification environments.

This standardization allows users to implement verification modules that are portable and highly compatible, and helps to exploit engineers' domain specific skills from one project to another. Widely promoted in the industry, SV methodologies catalyze the rapid sharing of expertise and proven techniques among the user base and encourage the growth of a *Verification Component* (VC) intellectual property (VIP) marketplace.

VCs are encapsulated, ready-to-use configurable verification environments for an interface protocol, a design submodule or a full system. Each VC follows a consistent architecture and consists of a complete set of elements for stimulating, checking, and collecting *Functional Coverage* [3] information for a specific protocol or design. The same VCs can be reused horizontally across projects or vertically from block-level verification to integration or system level verification.

UVM provides a comprehensive base class library supporting the construction and deployment of testbenches composed of reusable VCs. It is explicitly simulation-oriented, and intended to perform coverage-driven constrained random verification, but can also be used alongside assertion-based verification, hardware acceleration or emulation. In addition to a rich base class library, it provides a reference best-practice verification methodology. It is fully supported by the major tool vendors and is maintained by the industry-recognized body Accellera.

UVM heritage includes *Mentor*'s AVM, *Mentor & Cadence*'s OVM, *Verisity*'s eRM, and *Synopsys*'s VMM-RAL. These previous methodology libraries provided a rich legacy upon which UVM was built. Most notably, OVM-2.1.1 was the starting point for UVM, the code base that seeded the development effort.

This work introduces UVM from a user perspective and presents the experience gained in several of its fundamental aspects by its application in the verification of two basic units. The complete verification process involved in the two case studies is not detailed, the focus is set in the analysis of UVM and SV employed features instead. Similar work has been done analyzing previous methodologies like OVM [4].

The remainder of the paper is organized as follows. Section II introduces the key concepts implemented in the *UVM Class Library* which gives support to the methodology. It explains the main features and testbench elements provided by the *UVM Component Class* and the *UVM Transaction Class*. Section III y IV analyses how this features were applied to the verification of a *First Input-First Output (FIFO) buffer* module and an *I2C EEPROM* module respectively. Finally V discusses the main advantages and difficulties encountered in the application of the methodology and concludes the paper.

II. The UVM Class Library

The *UVM Class Library* [5] provides the building blocks needed to develop well-constructed and reusable VCs and test environments. The library consists of base classes and infrastructure facilities. Base classes in the UVM hierarchy largely fall into two distinct categories: components and data. The component class hierarchy derives from *uvm_component* and is intended to model permanent structural parts of the testbench such as monitors and drivers. Data classes are derived from *uvm_sequence_item* and are intended to model stimulus and transactions. The infrastructure facilities provide various utilities to simplify the development, management and use of verification environments, such as phasing and execution control, configuration methods, factory convenience methods, interconnection of components facilities and hierarchical reporting control.

ISBN: 978-987-1907-86-1

IEEE Catalog Number CFP1454E-CDR
Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.

A. The UVM Component Class

All the infrastructure components in a UVM environment derive from the *uvm_component* class and build a hierarchy which includes: sequencers, drivers, monitors, coverage collectors, scoreboards, environments and tests. In the following subsections the components that conform a UVM testbench are introduced:

1) Sequencer, Driver and Monitor: the sequencer is the entity that runs stimulus generation code and sends sequence items downstream to a driver. The driver is in charge of stimulating the *Design Under Verification* (DUV) through its interface. It always sits downstream of a sequencer from which it pulls transaction sequence items through a standard communication port. It maps the sequence items to the signal level format required by the DUV interface. The monitor scans signal level traffic going to and from the DUV from which it assembles sequence items that it distributes to the rest of the verification environment through one or more standard communication ports.

2) Coverage and Scoreboard: the coverage collector entity measures the advance of the verification process by registering the kind of tests and results that occur over a *Functional Coverage Model*. The scoreboard analyzes the functional correctness of the test outputs comparing them to a *Reference Model*. Both coverage collector and scoreboard depend on data provided by a monitor, which is best kept as a separate entity for re-usability. Monitor components are usually specific to particular protocols and sensible to signal level parameters, but independent of the application. In contrast, coverage collectors and scoreboards code is usually highly application-specific and less affected by the interface protocols and timing.

3) Agent: is an abstract container that typically encapsulates a driver, a sequencer and a monitor. They are usually configurable in active or passive modes; in active mode they are used to emulate devices and drive transactions according to test directives, while in passive mode they are only used to monitor DUV activity.

4) Environment: is the entity that assembles the testbench structure. It instantiates one or more agents, a gloabal scoreboard, etc., as well as other components such as a bus-level monitor to perform system level coverage measurement and checking. It has configuration parameters that allow to restructure and reuse it for different scenarios.

5) Test: is the top-level of the component hierarchy. It is used to:

- Control the generation of the environment configuring the behavior of its VCs.
- Determine the dynamic behavior of the verification process by starting sequences in specific sequencers.
- Extent and adapt the tesbench by overriding the components structure or by overriding data items and sequences.

Each test only need to contain the few specific modifications to the verification environment that distinguish the user's test from the default situation.

B. The UVM Transaction Base Class

The dynamic data domain within the verification environment is represented by:

1) Sequence items: are the basic data objects that are passed between components. In contrast to VHDL signals and *Verilog* wires, sequence items represent communication at an abstract level.

2) Sequences: are assembled from sequence items and are used to build realistic sets of stimuli. A sequence could generate a specific pre-determined set of transactions, a set of randomized transactions or anything in between. Sequences can run other sequences and can be layered such that higher-level sequences send transactions to lower-level sequences in a protocol stack.

III. FIFO Buffer Testbench

UVM was applied following the guidelines introduced in [6] for the development of a reusable testbench for the unit level verification of a *FIFO buffer* module. It was implemented and debugged using *Synopsys VCSMx* and *DVE* version H-2013.06 with its pre compiled UVM-1.1d library. The verification environment architecture is shown in Fig.1 and the following sections describe each of its main elements.

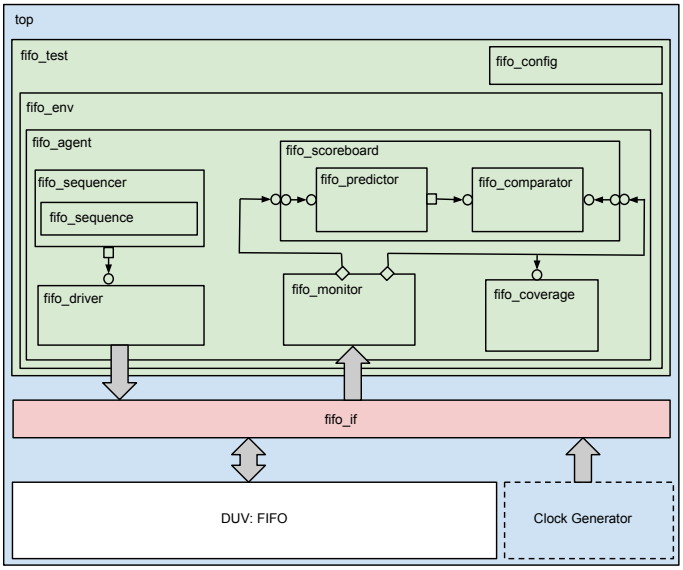

Fig. 1. *FIFO* verification environment architecture.

A. DUV Specifications

The DUV is an in house designed synchronous *FIFO buffer* module in which data items are stored in a RAM memory.

The operation of the module is controlled by *rst*, wr_{en} and rd_{en} signals, while *empty* and *full* define its state. Ports *din* and *dout* are used with wr_{en}, rd_{en} to put data into the back of the queue and to get data from the front respectively. Simultaneous write and read operations are allowed. Figure 2 shows the timing specifications for the module.

ISBN: 978-987-1907-86-1

IEEE Catalog Number CFP1454E-CDR
Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.

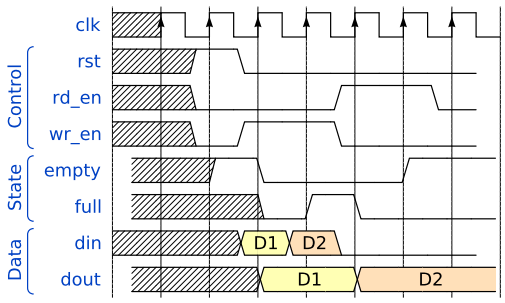

Fig. 2. Timing specifications for *Reset, Write(D1), Write(D2), Read, Read*

B. Module top

The SV verification program main entity is *top*, a SV static module which instantiates the DUV and the interface, connects them and defines the clock signal. In order to use UVM it must also import the UVM *Package* which provides the basic functionalities and resources of this framework.

UVM *Package* supplies the *run_test* method that is used to start the selected UVM tesbench execution. It also gives access to the UVM *Configuration Database* that provides a shared memory location where configurations and shared resources are stored. The *Top* module registers the interface into this database in order to allow the class based testbench to connect to the DUV.

C. Interface fifo_if

The interface facilitates the communication between the SV class based tesbench components and the DUV often implemented in *VHDL* like this *FIFO buffer* module. Besides defining the input-output signals format, it can be used to implement a *Bus Functional Model* (BFM).

A BFM provides an *Application Programming Interface* (API) that maps a bus or input-output ports signal level behavior to tasks at a higher level of abstraction. The implemented BFM in the interface provides *no_op, write, read, read&write, reset* and *monitor* operations. This approach proved to enforce a clear interaction with the DUV hiding low level communication, synchronization and timing details. The SV *Clocking Block* construct was used to implement signal level timing manipulation and avoid race conditions when reading and writing the interface signals.

D. Component class fifo_test

The *fifo_test* component initializes the testbench infrastructure. On its UVM *build phase* it sets a configuration object that parametrizes the overall testbench structure and stores it into the UVM *Configuration Database* to allow the rest of the components to access it and auto configure themselves. These configurations involve: enabling or disabling the debug messages of each component, setting the agents' active or passive modes and enabling the scoreboard creation and coverage collection. In this phase the test also instantiates the environment component which in turn instantiates the agent. On its UVM *run phase* the test selects and starts a test sequence on the agent's sequencer as seen on Listing 1.

```
task run_phase(uvm_phase phase);
 fifo_sequence seq;
 phase.raise_objection(this, "Start base_fifo_sequence");
 seq = fifo_sequence::type_id::create("seq");
 // Randomize the sequence.
 assert( seq.randomize() );
 // Start the sequence on the agent's sequencer.
 seq.start(env.agent1.sequencer);
 phase.drop_objection(this, "Finished base_fifo_sequence");
endtask
```

Listing 1. Test *run phase.*

E. Component class fifo_agent

On its UVM *build phase* (Listing 2) and according to the configuration object stored in the UVM *Configuration Database*, the agent creates its blocks: sequencer, driver, monitor, scoreboard and coverage collector. If the agent is active it should create the sequencer and the driver, otherwise if it is passive it only instantiates the monitor. In the UVM *connect phase* it binds its components' ports according the agent's structure, for example, it connects the driver to the sequencer.

```
function void build_phase(uvm_phase phase);
 if (settings.active == UVM_ACTIVE) begin
  driver=fifo_driver::type_id::create("driver",this);
  sequencer=fifo_sequencer::type_id::create("sequencer",
       this);
 end
 if (settings.has_functional_coverage == 1) begin
  coverage = fifo_coverage::type_id::create("coverage",this
       );
 end
 monitor=fifo_monitor::type_id::create("monitor",this);
 if (settings.has_checker == 1) begin
  scoreboard=fifo_scoreboard::type_id::create("scoreboard",
       this);
 end;
endfunction
```

Listing 2. Agent build phase.

F. Data class fifo_item

This class models the basic item for test sequences. Each item contain a set of attributes used to register a *FIFO buffer* operation, its associated data and its state as defined in the DUV specification. Every of its attributes that models commands or input data are enabled for randomization. This can be see in Listing 3.

```
// Commands:
rand logic read;
rand logic write;
logic reset;
// State:
logic full;
logic empty;
// Data
rand logic [dw-1:0] data_in;
logic [dw-1:0] data_out;
```

Listing 3. Item attributes.

This class implements a set of methods to directly define items as a *no_op, write, read, read&write* or *reset* operation for the purpose of using them to create directed tests. The *set_write* operation is shown in Listing 4.

ISBN: 978-987-1907-86-1

IEEE Catalog Number CFP1454E-CDR
Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.

```
function void set_write(logic [dw-1:0] data_in);
  this.write = 1;
  this.read = 0;
  this.reset = 0;
  this.data_in = data_in;
endfunction
```

Listing 4. Set method for configuring the item as a write operation.

G. Data class fifo_sequence

Sequences are high level test vectors. Directed sequences are defined using the methods in *fifo_item* class; this organization simplifies the creation of sequences, improves its readability and eases modifications, although random test can also be easily defined. A simple *reset, write(din), read* directed test sequence is showed in Listing. 5.

```
task body();
 fifo_item i_rst1, i_wr1, i_rd1;
 // Reset
 i_rst1 = fifo_item::type_id::create("i_rst1");
 start_item(i_rst1);
 i_rst1.set_reset();
 finish_item(i_rst1);
 // Write
 i_wr1 = fifo_item::type_id::create("i_wr1");
 start_item(i_wr1);
 i_wr1.set_write(din);
 finish_item(i_wr1);
 // Read
 i_rd1 = fifo_item::type_id::create("i_rd1");
 start_item(i_rd1);
 i_rd1.set_read();
 finish_item(i_rd1);
endtask
```

Listing 5. Basic directed test sequence (reset, write(din), read).

H. Component class fifo_driver

The driver component is in charge of finally stimulating the DUV. Its main task, the *run_phase* (Listing 6), executes a loop that iteratively gets an item from the sequencer and calls the corresponding interface BFM task to generate the proper signal level vectors.

```
task run_phase(uvm_phase phase);
 fifo_item item;
 forever begin
  // Gets the next item from the sequencer.
  seq_item_port.get_next_item(item);
  case (item.get_item_type())
   "RESET"          : bfm.reset();
   "READ"           : bfm.read();
   "WRITE"          : bfm.write(item.data_in);
   "WRITE_AND_READ": bfm.read_and_write(item.data_in);
   "NO_OP"          : bfm.no_op();
  endcase
  seq_item_port.item_done();
 end
endtask
```

Listing 6. Driver's main task.

I. Component class fifo_monitor

The monitor senses the DUV signals through a special task provided by the BFM. When the monitor detects an operation on the interface (*read, write, read&write, reset*) it creates an item. Given the timing specifications of the DUV, it has to wait one clock cycle to be able to observe the results. In this way the item is assembled with the operation captured in cycle *n* and its result captured in cycle *n+1*. Finally, it sends the item to the predictor in the scoreboard and to the coverage collector.

J. Component classes fifo_scoreboard and fifo_coverage

The scoreboard component is formed by a predictor and a comparator. The predictor implements the *Reference Model* of the *FIFO buffer* and as the monitor produces and send items to it, it takes the operation fields (discarding the sensed result), evaluates the input and produces a new predicted item. The comparator synchronizes the items with the DUV's output coming from the monitor with the predicted items, compares them and generates the corresponding error messages it the item values are different.

The coverage collector also receives the items from the monitor and registers them in *SV Cover Points* and *Cover Groups* which implement the corresponding *Functional Coverage Model* [3]. This kind of coverage is a *Black Box* metric because there is no access to the internal signals of the DUV.

IV. I2C EEPROM Testbench

The experience acquired with the *FIFO buffer* testbench was exploited in the development of a UVM testbench for unit verification of a *I2C EEPROM* slave module. The architecture of the testbench is similar to the previous case study, enabling the reuse of most of the structure and the already implemented VCs with minimal modifications.

Only the specific and relevant components are presented in the following sub sections, detailing design and implementation decisions targeting the I2C protocol. The most relevant modifications involve the BFM which now implements the I2C protocol. The testbench was implemented and debugged using *Mentor Graphics ModelSim SE-64 10.1c* with its pre compiled UVM-1.1b library.

A. DUV Specifications

The DUV is a 24C01 [7] 1 K-bit memory, 8 bit data width. It is a slave I2C module [8] implemented as an *Open Source* VHDL entity with device address 1010000 and 7 bit I2C address mode.

In order to write a byte to the slave *I2C EEPROM* the I2C messages shown in Fig. 3 must be sent and received by the master I2C device (testbench).

To read a byte from a memory address in the EEPROM the following I2C messages must sent to it: **(1)** I2C Start Condition, **(2)** I2C Device Address and I2C Write Bit, **(3)** EEPROM Address Byte, **(4)** I2C Start Condition and **(5)** I2C Device Address and I2C Read Bit. First the memory address must be written to the EEPROM trough an I2C message (step 2 and 3), then an I2C read message (step 5) is sent to read a byte from the previously set memory address.

B. Interface i2c_m_if

In the I2C protocol, both *SDA* and *SCL* are bidirectional lines, connected to a positive supply voltage via a current source or pull-up resistor. When the bus is free, both lines are in high impedance. This behavior is modeled in the interface using SV *str1* (tristate) data type and *scl_oe* and *sda_oe* as its respective control variables as shown in Listing 7.

ISBN: 978-987-1907-86-1 IEEE Catalog Number CFP1454E-CDR
Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.

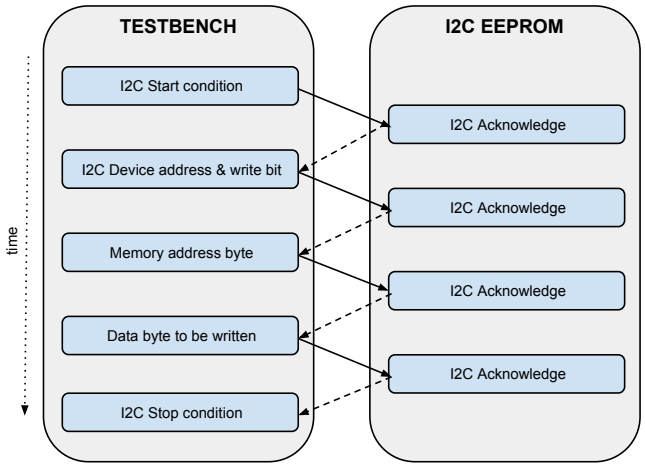

Fig. 3. The protocol communication flow for writing a byte in a desired address of the EEPROM.

```
interface i2c_m_if(inout tri1 sda);
 // I2C bus signals
 tri1  scl;
 // Tri-state buffer control
 logic scl_oe;
 logic sda_oe;
 // Tri-state buffers
 assign scl = scl_oe ? 1'bz : 1'b0;
 assign sda = sda_oe ? 1'bz : 1'b0;
endinterface
```

Listing 7. *SDA* and *SCL* lines modeled with pull-up resistors in the interface. If *scl_oe* is high then *scl* is in high impedance. If *scl_oe* is low then *scl* is low.

C. Generic class i2c_m_bfm

This class is the BFM used by the driver. It implements the low level details of the I2C protocol trough three main tasks that interact with the interface according to the information of the item they receive:

- *busIdle(i2_m_item tr):* It hold the bus idle for the corresponding clock cycles.
- *initialValues():* it sets the *SDA* and *SCL* lines to high impedance.
- *burstReadWrite(i2c_m_item tr):* It follows the I2C protocol in order to write or read data from a desire I2C device. The pseudo algorithm in Listing 8 explains how it works.

```
// Generate I2C start condition.
i2cStartCondition();
// Send address and read/write bit.
i2cAddressRWbit();
if (item.OperationType = WRITE)
 // Send data buffer.
 then i2cByteWrite(item)
 // Receive data buffer.
 else i2cByteRead()
if (item.genStop = TRUE)
 // Generate I2C stop condition.
 then i2cStopCondition()
```

Listing 8. The pseudo algorithm for the *burstReadWrite* task.

These tasks use the timing parameters from the *i2c_m_bfm_config* object (explained at the end of this section) to establish the correct timing behaviour of the signals.

D. Data class i2c_m_item

These items represent the transactions that can be performed over an I2C protocol bus by a master device.

The class has a field for the I2C address of the device to which the transaction is intended to interact with, a field that specifies the type of transaction (*READ, WRITE, RESET, IDLE*) it represents, a field to model the amount of idle time on the bus and a field to decide if the I2C stop signal should be generated at the end of the transaction.

Three set methods are implemented to easily define the items in order to define *Directed Tests* sequences:

- *setBusIdle(time idleTime):* it makes the item represent an idle time on the bus.
- *setReadData(int address, int lenght, int genStop):* it makes the item represent an I2C read operation of *length* number of bytes over the device with the specified address. *genStop* decides if an I2C stop condition is to be generated after the last read byte.
- *setWriteData(int address, bit8 dataInBuff[], int genStop):* it makes the item represent an I2C write operation of *length* number of bytes in the I2C device with the specified address. The *genStop* decides if an I2C stop condition is to be generated after the last written byte.

E. Data class i2c_m_sequence

Directed Test sequences can be easily implemented using the previously described item's set methods. A simple example in showed in Listing 9. First an item is created trough the UVM *Factory Method*, then the *I2C EEPROM* 7 bit address (1010000) is loaded into a variable, then a list of bytes is created (0,1,2,3,4), where the first element (0) represents the word address inside the EEPROM and the rest of the elements the bytes to be written. Also a variable to decide if an I2C stop signal will be generated is set. After that, the item is set trough the *setWriteData* to model a write transaction with that information.

```
i2c_m_item i0;
bit8 dataInBuff[\$];
int device_address;
int genStop;
// UVM Factory Method.
i0 = i2c_m_item::type_id::create("i0");
start_item(i0);
device_address = 7'b1010000;
genStop = 1;
// The first byte(0) is the address inside the EEPROM.
dataInBuff = {0,0,1,2,3,4};
i0.setWriteData(device_address, dataInBuff, genStop);
finish_item(i0);
```

Listing 9. A *Directed Test* sequence that writes a stream of 5 consecutive bytes staring from address 0.

F. Component class i2c_m_driver

The driver development was simplified by the usage of the BFM implemented by the *i2c_m_bfm* class. The driver gets the item from the sequencer and depending on the type of the item (*READ, WRITE, IDLE*) it calls the appropriated BFM method to take care of the low level timing and synchronization details.

ISBN: 978-987-1907-86-1

IEEE Catalog Number CFP1454E-CDR
Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.

G. Generic class i2c_m_bfm_config

This object is used to configure the BFM. It sets the I2C timing configuration like clock frequency, set-up and hold times for Start and Stop conditions, data set-up and hold times, etc. in one central place. These parameters values can be obtained from page 48 of [8] for all the *I2C Bus Speed Modes*.

This object is created and initialized by the test component. It is is set by defining the *I2C Bus Speed Mode* (*Standart, Fast* or *FastPlus)* and from that information the object derives the timing configuration that will be used by the BFM. Then this object is placed in the UVM *Configuration Database* to allow the BFM to acquire it.

V. Conclusion and Discussion

The *Universal Verification Methodology* was analyzed through the development of two testbenches for unit verification. The first one targeted a *FIFO buffer* module and employed all the basic UVM components; a scoreboard with a *Reference Model* and a *Functional Coverage* collector were also implemented. The second testbench was designed to verify an *I2C EEPROM* slave module leveraging the experience acquired in the development of the previous testbench. A configurable BFM for the I2C protocol was developed to facilitate the driver's implementation, rising the level of abstraction and allowing the reuse of the VC for other I2C devices.

UVM provided a complete framework that facilitated the development of the testbenches. There were many implemented and ready to use OOP features provided by the library, like the VC *Factory Methods* and the VC communication mechanism which facilitated the development. However previous knowledge in SV and OOP is mandatory and some of the UVM utilities, like factory registering macros, are not simple to understand and correctly use at start up.

One of the most challenging tasks was building the first UVM working testbench. The different components could not be tested individually, so all the environment had to be assembled in order to test the components; making it hard to trace the source of errors. For this issue it was helpful to control the way in which components reported debug messages. It is also recommended to use some third party unit testing tool like [9] or [10] in order to test each component individually before integrating them to the rest of the environment.

The most cumbersome part to implement were the code sections which involved communication and synchronization between the class based testbench and the *Hardware Definition Language* (HDL) DUV modules. Several SV constructs like *Interfaces*, *Clocking Blocks* and *Modports* had to be used in order to properly synchronize the writing and reading of signals. The monitor was also complicated to develop due to the lack of information or examples on how to create data items when the DUV does not have a synchronization signal.

Once the level of abstraction was raised trough the development of a driver or BFM, the test sequence generation was direct and fast, increasing the productivity by allowing us to focus on the development of good test sequences avoiding its low level details. It was also useful to put all the protocol timing parameters in one object in order for the driver or BFM to use, making them reusable for different protocol configurations.

After the first testbench was built and correctly executed it was easy and fast to re use it in other projects thanks to the re usability provided by the framework based on the OOP paradigm, the standard connection of the components and the architecture mandated by the methodology. This was experienced in the development of the *I2C EEPROM* testbench based on the *FIFO buffer* testbench, where the previously acquired knowledge allowed us to build the new testbench much faster by reusing many of the already working components and UVM utilities. It was clear that the productivity increased dramatically after the development of the first UVM testbench. Although there was much re usability from project to project the UVM/SV code was not 100% portable from one EDA provider to the other. Minor code modifications had to be done in order for the VC to work in the different simulators.

Acknowledgment

This work was partially funded by FSTICS 001 TEAC, PICT 2010 2657 and PAE 37079. EDA tools were provided by *Mentor Graphics Corp.* and *Synopsys* through its respective University Program agreements.

References

[1] Accellera, *UVM 1.1 User Guide*, 2011.
[2] C. Spear, *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*, 2nd ed. Springer Publishing Company, Incorporated, 2008.
[3] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, 1st ed. Springer Publishing Company, Incorporated, 2007.
[4] O. Cadenas and E. Todorovich, "Experiences applying ovm 2.0 to an 8b/10b rtl design," in *5th Southern Conference on Programmable Logic (SPL), 2009*, April 2009, pp. 1–8.
[5] Accellera, *UVM 1.1 Class Reference*, 2011.
[6] Mentor Graphics Corp, *UVM Cookbook*, 2014. [Online]. Available: https://verificationacademy.com/cookbook/uvm
[7] STMicroelectronics, *M24C01-R 1 Kbit serial I2C bus EEPROM Datasheet*, 2013. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00071904.pdf
[8] NXP Semiconductors, *I2C-bus specification and user manual*, 2012.
[9] R. S. Bryan Morris, "Svunit: Introducing a tdd framework for systemverilog," in *SNUG San Jose*, 2009.
[10] N. Johnson. (2013) Uvm-utest. [Online]. Available: http://www.agilesoc.com/open-source-projects/uvm-utest/uvm-utest-getting-started/

Authorized licensed use limited to: University of Tasmania. Downloaded on April 02,2024 at 05:52:20 UTC from IEEE Xplore. Restrictions apply.