FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of a reconfigurable multi-protocol verification environment using UVM methodology

Pedro Araujo

PREPARAÇÃO PARA A DISSERTAÇÃO
PRELIMINARY REPORT

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Teacher supervisor: José Carlos Alves

Company supervisor: Luis Cruz

February 22, 2014

© Pedro Araujo, 2014

Contents

1 Introduction **1**
1.0.1 Motivation and goals 1

2 State of the art **3**
2.0.2 Hardware Description Languages 3
2.0.3 Verification 4
2.0.4 Hardware Verification Languages 6
2.0.5 Verification Methodologies 7
2.0.6 The Universal Verification Methodology 8
2.0.7 Future work of the chapter 11

3 Work Plan **13**
3.0.8 Methodology 13

List of Figures

2.1 Generic testbench 4
2.2 Direct testing progress [7, p.6] 5
2.3 Random testing progress [7, p.8] 5
2.4 Layered testbench 6
2.5 Evolution of verification methodologies 8
2.6 Typical UVM testbench 9
2.7 UVM testbench with multiple tests 10
2.8 UVM testbench with multiple tests 10

3.1 Gantt chart of the work plan 13

List of Tables

3.1 Work plan 13

Abreviaturas e Símbolos

DUT	Device Under Test
EDA	Electronic Design Automation
HDL	Hardware Description Language
HVL	Hardware Verification Language
IEEE	Institute of Electrical and Electronics Engineers
OVM	Original Verification Methodology
PSL	Property Specification Language
UVM	Universal Verification Methodology

Chapter 1

Introduction

This thesis' project was proposed by Synopsys Portugal and it's going to be carried out under the Integrated Master's Degree in Electrical and Computer Engineering of the Faculdade de Engenharia da Universidade do Porto. The current chapter will introduce the thesis' subject and the motivation behind the work.

1.0.1 Motivation and goals

During the last decades, electronic circuits have grown in complexity and in production costs which compelled engineers to research and develop new methods to verify the electronic design in more comprehensive, detailed and efficient ways.

The UVM methodology is one of the results of the increasing need of digital verification. It's is designed in a way that allows to structure the verification environment in a reconfigurable architecture, so it can be possible to reuse the same environment across multiple technologies. This methodology is an industry standard recognized by Accellera System Initiative and it's comprised of a library for the SystemVerilog language (IEEE 1800) and a set of verification guidelines.

The purpose of this work is to take advantage of the best features of UVM and develop a reconfigurable verification environment that supports multiple protocols with minimal development effort. The project will start with the analysis of an existing environment used in a specific technology and then followed by an analysis of the verification techniques that could be used across different protocols.

The goals defined for this project are:

- Analysis of an existing verification environment and removal of all design logic specific to the original protocol
- Revision of the verification environment in order to support multiple protocols
- Creation of generic blocks to support the revised environment
- Configuration and application of the generic environment to another existent protocol

Chapter 2

State of the art

Technology has advanced a long way and become increasingly complex. Its foundations started with computers whose logic was maintained by valves and that eventually moved to microscopic devices, like transistors.

In the early beginnings, electronic systems were designed directly at the transistor level by hand, but due to the increasingly complexity of electronic circuits since the 1970s [4], it became unpractical to design the core logic directly at the transistor level, so circuit designers started to develop new ways to describe circuit functionality independently of the electronic technology used. The result was the Hardware Description Languages and the era of Electronic Design Automation was born.

Hardware description languages are design languages that are used to program the behavior and the structure of integrated circuits before they are translated into their own architecture. These kind of languages enable the modeling of a circuit for posterior simulation and, most importantly, translation into a lower level specification of physical electronic components.

A hardware description language resembles a typical programming language, it consists in a textual description of expressions and control structures. Although they both share some similarities, they are not the same. One main difference is that HDL code is executed concurrently, which is required in order to mimic hardware, and while programming languages, after compilation, are translated into low level instructions for the CPU to interpret, HDL is synthesized into real hardware, so using hardware description languages requires a different mindset than using programming languages.

Nowadays, hardware description languages are the prevailing way of designing an integrated circuit, having superseded schematic capture programs in the early 1990s, and became the core of EDA. [6, p.15-1]

2.0.2 Hardware Description Languages

VHDL and Verilog are the two most popular industry standards. [2] Verilog was created as a proprietary language by Phil Moorby and Prabhu Goel between 1983 and 1984 and it's formally

based in the C language. Cadence bought the rights of the language in 1989 and made it public in 1990. Eventually, IEEE adopted it as a standard in 1995 (IEEE 1364).

On the other hand, unlike Verilog which was originally designed to be used as a proprietary tool, VHDL was intentionally designed to be a standard HDL. It was originally developed on behalf of the U. S. Department of Defense between 1983 and 1984 but only released in 1985. It's based on Ada programming language and it was adopted as a standard IEEE 1076 in 1978. [6, p.15-3]

2.0.3 Verification

Hardware description languages are tools that help engineers to easily translate circuit logic into real hardware, but after the design is complete another issue becomes noticeable: how can a designer know that the design works as intended?

This brings up the topic of verification. Verification is defined as a process to demonstrate the functional correctness of a design. [1, p.1] This process is done by the means of a testbench, a system that connects to the input and outputs of the design under verification (DUV) and analyses its behavior. A generic testbench is represented in Figure 2.1.

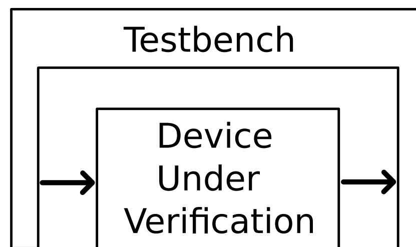

Figure 2.1: Generic testbench

One of the most common uses of a testbench is to show that a certain design implements the functionality defined in the specification. This task is denominated of functional verification. But it's important to take in account that functional verification can show that a design meets the specification but it cannot prove it. [1, p.2]

Traditionally, the approach to verification is with the use of directed tests: verification engineers conceive a series of critical stimulus, apply them directly to the device under test (DUT) and check if the result is the expected one. This approach makes steady progress and produces quick results because it requires little infrastructure. So given enough time, it's possible to write all the tests needed to cover 100% of the design. This scenario is represented in Figure 2.2.

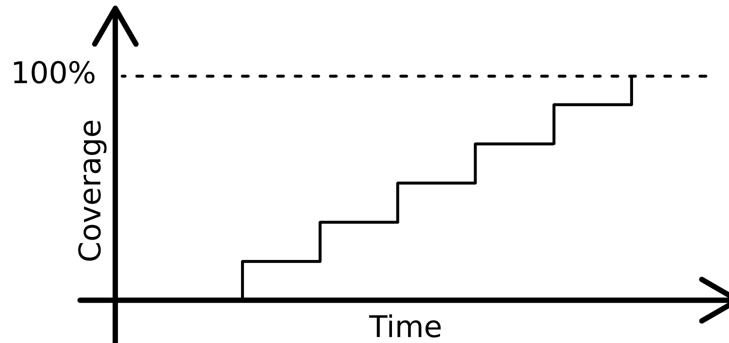

Figure 2.2: Direct testing progress [7, p.6]

But as the design grows bigger and more complex, this becomes a tedious and a time consuming task. Most likely, there will be not enough time to cover all the tests needed and there will be bugs that the verification engineer won't be able to predict. So, random stimulus help to cover the unlikely cases. However, in order to use random stimuli, there is the need of a block to automatically generate them and, besides of that, there is also the need of a scoreboard to predict and keep track of the results. Additionally, when using random stimulus, it will be needed to check what cases were covered by the generated stimuli, so the testbench will need functional coverage as well. Functional coverage is the process of measuring what space of inputs have been tested, what areas of the design have been reached and what states have been visited. [7, p.13]

This kind of testbench requires longer time to develop, causing a initial delay in the verification process, but on the other hand, random based testing can actually speed up the verification coverage of the design as seen in Figure 2.3.

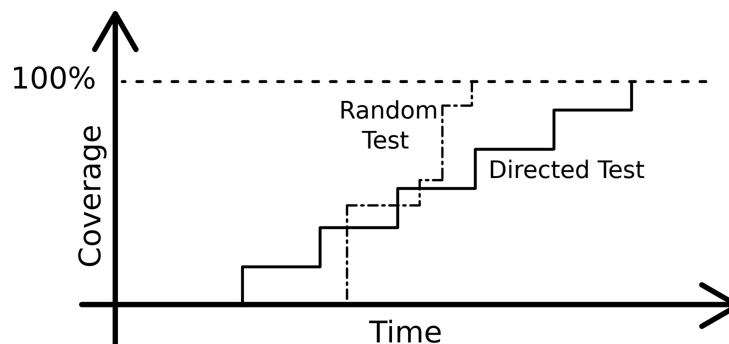

Figure 2.3: Random testing progress [7, p.8]

2.0.4 Hardware Verification Languages

The described testbench already has a stimulus generator, a functional coverage block, a scoreboard. However, it needs a block to drive the generated stimulus to the DUT and a block that listens to the communication bus, so that the responses of the DUT can be driven to the scoreboard block and to the functional coverage block. A representation of this layered testbench can be seen in Figure 2.4.

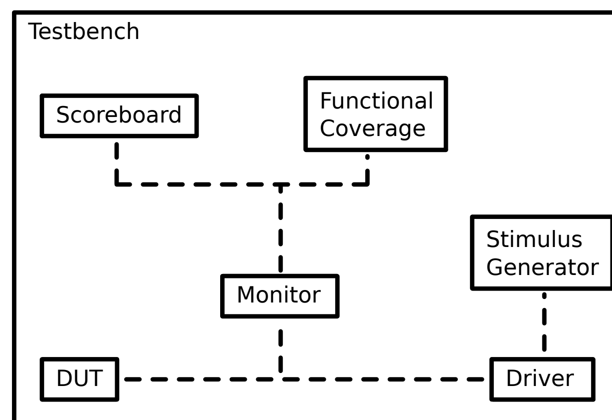

Figure 2.4: Layered testbench

The testbench starts to grow more and more complex and it starts to be possible to imagine that verification might be a more resource consuming task than the design itself. Today, in the semiconductor industry, verification takes about 70% of the design effort and the number of verification engineers is twice the number of RTL designers in the same project. After a project is completed, the code of the testbench takes up to 80% of the total volume code. [1, p. 2]

As the circuit complexity started to grow, the verification process started to increase in complexity as well and became a very important part of the project. However, the typical HDL aren't able to cope with the complexity of today's testbenches. Complex data structures, application of constraints to the random stimulus, presence of multiple functional blocks and functional coverage, are all examples of features that aren't available in HDL, mainly in Verilog and in VHDL. Hardware description languages are more focused in creating a model of a digital design than programming verification. [3]

In order to solve this problem, Hardware Verification Languages (HVL) were created. The first one to be created was Vera in 1995. It was an object-oriented language, originally proprietary, designed for creating testbenchs for Verilog. The language was bought in 1998 by Synopsys and released to the public in 2001 as OpenVera. OpenVera has support for high-level data structures, constraint random variables and functional coverage, which can monitor variables and state transitions, and Synopsys also added support for assertions capabilities.

In 1997, a company named Verisity created the proprietary language *e*. Like OpenVera, it's also object-oriented and feature-wise, it's very similar to Vera.

IBM also developed its own verification language, Property Specification Language. It's more narrower than OpenVera or *e* and it's designed to exclusively specify temporal properties of the hardware design. [3]

Eventually, the features of some languages like, OpenVera and PSL, were merged to Verilog and it was created a new language, SystemVerilog. SystemVerilog supports a variety of operators and data structures, as well constrained random variables, functional coverage checking and temporal assertions. This new language was adopted as a standard IEEE 1364 in 2005 and it's the most used verification language in the industry. [6, p. 15-17]

2.0.5 Verification Methodologies

The adoption of verification languages eased the process of verification but there was no consensus in the proper use of a verification language. So in the attempt of helping to deploy the right use of a verification methodology, Verisity Design (now Cadence Design Systems) published in 2000 a collection of the best practices for verification targeted to the users of *e* language. Named vAdvisor, it just consisted in a package of verification patterns in HTML format and it covered many aspects like coverage modeling, self-checking testbenches and stimuli creation. In 2002, the same company revised vAdvisor and created the first verification library, the *e* Reuse Methodology (*e*RM). It was a big step because it included fundamental examples like sequences, objection mechanisms and scoreboards.

In order to compete with Verisity Design, Synopsys presented in 2003 the Reuse Verification Methodology (RVM) for Vera verification language. The most notable contribution of RVM was the introduction of callbacks, that was inspired from software design patterns and adapted to verification languages. Eventually, RVM was port from Vera to SystemVerilog to create the Verification Methodology Manual (VMM).

Until this point in time, both verification methodologies had been proprietary and it was only in 2006 that it was introduced the first open source verification methodology, the Advanced Verification Methodology (AVM) from Mentor. This library incorporated the concept of Transaction-Level Modeling from SystemC.

In 2005, after the acquisition of Verisity, Cadence started, as well, to port *e*RM to the standard of hardware verification languages, SystemVerilog. The result was the open source methodology, Universal Reuse Metholody (URM) in 2007.

However, in the joint task of merging the best features of each methodology, in 2008, Cadence and Mentor integrated both URM and AVM into a single open source methodology, the Open Verification Methodology (OVM). This collaborative effort ended up to be a good solution, because not only unified the libraries and the documentation but also, due to the open source nature, users could make their own contributions to the methodology. [5, p. xvii]

Later on, Accellera group decided to adopt a standard for verification methodologies and OVM was chosen as basis for this new standard and together, with Synopsys' VMM contributions, a new methodology was created: the Universal Verification Methodology (UVM). A sum up of the evolution of verification methodologies is represented in Figure 1.5.

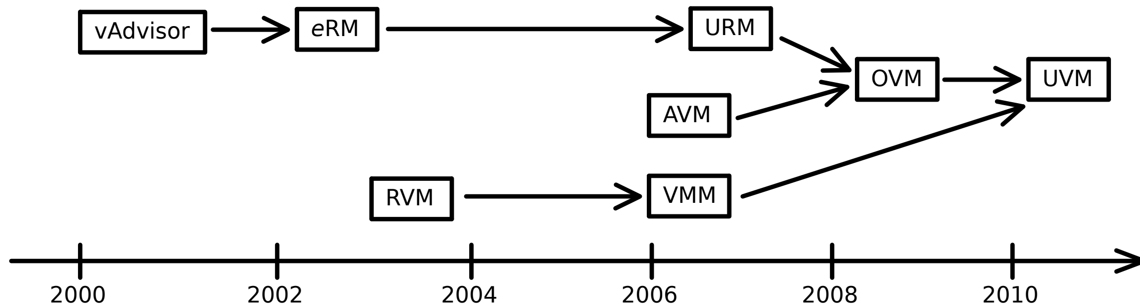

Figure 2.5: Evolution of verification methodologies

2.0.6 The Universal Verification Methodology

The UVM methodology is provided as an open-source library directly from the Accellera website and it should be compatible with any HDL simulator that supports SystemVerilog, which means it's highly portable. It's also based on OVM library, which means that it's proven code. These two points are two of the main reasons for industry approval of the methodology.

Another key feature of UVM includes reusability. In a traditional testbench, if the DUT changes, engineers would redo the testbench completely. This process takes some effort but most of the times, if the testbench is correctly programmed, some blocks of it could be reused for the new testbench.

On other occasions, under the same DUT, verification engineers might want to do apply a different test or change the kind of stimuli sent to the DUT. If the engineer doesn't take in mind that the test might change, he might end up revising the entire testbench.

Another situation: if another team of engineers inherits testbench programmed by someone else and the testbench wasn't programmed with a standard methodology in mind, the new team might end up wasting a lot of time just to understand the verification environment.

UVM also addresses these kind of situations and specifies an API and guidelines for a standard verification environment. This way, the environment is understood by any verification engineer that understands the methodology and it becomes easily modifiable.

In Figure 2.6, it's represented a typical UVM verification environment.

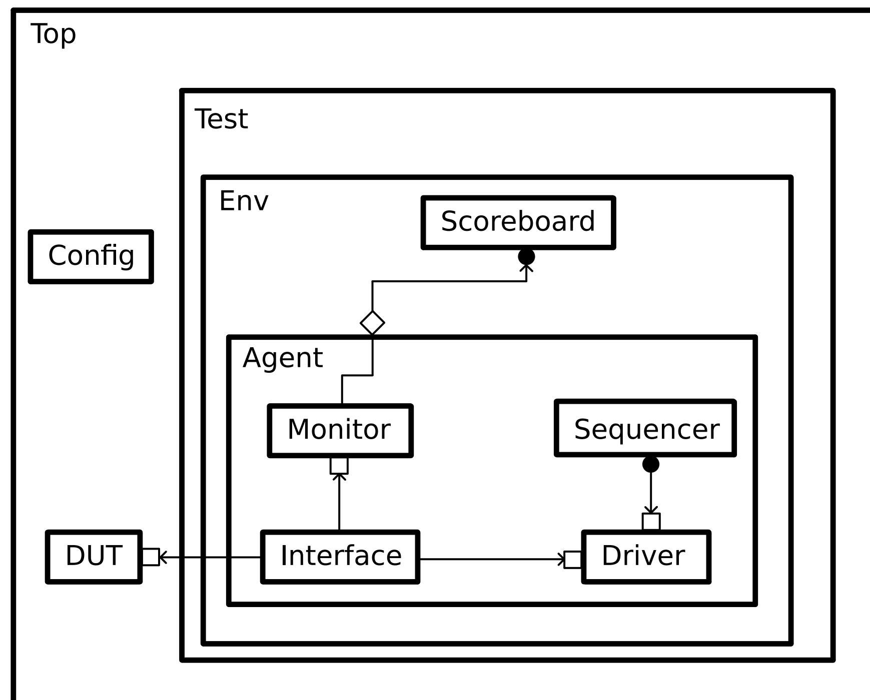

Figure 2.6: Typical UVM testbench

This environment is very similar to the one showed in Figure 2.4. The sequencer generates the random stimuli, the driver carries the stimuli to the interface and the interface send these stimuli to the DUT. The monitor registers the communication between the DUT and the drive, it analyses the responses of the DUT and sends the tests results to the scoreboard.

The way the environment is structured, it permits to easily add or change verification blocks. If the verification engineer wanted to change the constrains of the random sequences, he would know exactly which block he needed to modify. Or, as an example, if during the development phase, the specification of the communication bus changes, the only blocks that would need to be modified would be the monitor and the driver.

It's also possible to have multiple instances of the same type of block in the same testbench. For example, in Figure 2.7 it's represented an UVM environment with support for multiple different tests. This tests can be easily activated and deactivated during the simulation runtime, making it easier to execute a series of different tests.

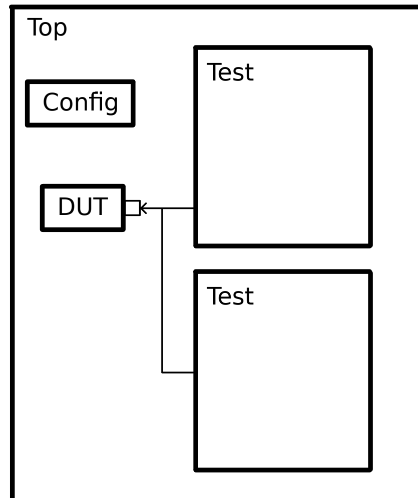

Figure 2.7: UVM testbench with multiple tests

Another example is represented in Figure 2.8. There are multiple agents communicating with the DUT, this is a useful case for single-master/multiple-slaves design configurations.

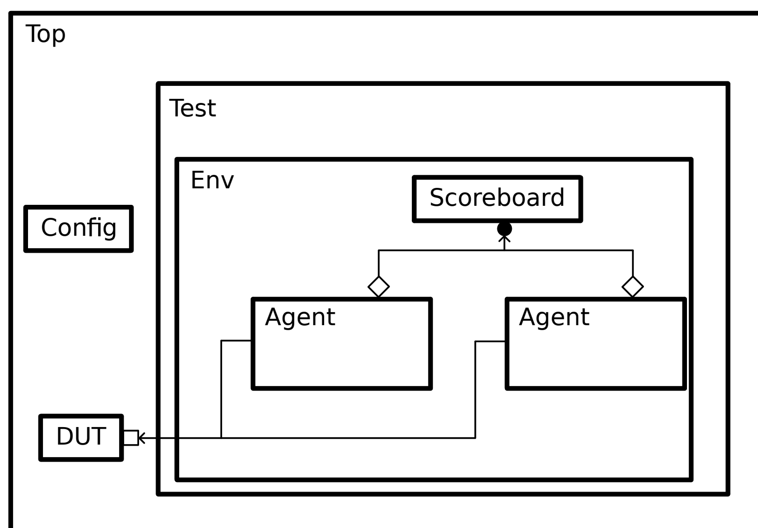

Figure 2.8: UVM testbench with multiple tests

2.0.7 Future work of the chapter

This methodology can be very flexible in terms of verification and includes a solid API oriented for verification. It wasn't possible to enumerate all the features of the technology at this time of writing, so this chapter will be revised later on, during the study and the development of the thesis' project.

Chapter 3

Work Plan

This chapter will present the work methodology that will be pursued throughout the semester.

3.0.8 Methodology

As it was mentioned in chapter 1, the objective of this project is to develop a reconfigurable verification environment that supports multiple protocols using the UVM methodology.

The first step will be to study and analyze the existing verification system, then with premise on that study, it will be proposed and discussed possible modifications on the system. After the discussion, it will be implemented the chosen approach on the environment.

The distribution of the work throughout the semester is described on Table 3.1.

Task	Description	Start	Finish
Task 1	Study of the existing UVM environment	3 Feb	21 Feb
Task 2	Proposal of possible implementations	24 Feb	28 Feb
Task 3	Implementation	3 Mar	2 May
Task 4	Aplication of the researched solution to an existing platform	5 May	16 May
Task 5	Writing of the dissertation	19 May	6 June

Table 3.1: Work plan

The Gantt chart of the work plan is represented on Figure 3.1.

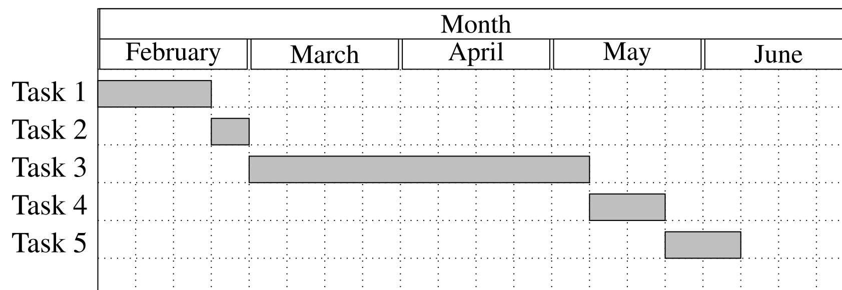

Figure 3.1: Gantt chart of the work plan

The project will be realized in the offices of Synopsys Porto using the company's resources. It's foreseen the usage of the tools:

- VCS: Verilog Compiled code Simulator
  Synopsys' HDL compiler and simulator. Verilog and SystemVerilog code will be compiled with this tool.
- DVE: Discovery Visualization Environment
  It's an advanced debugger and waveform visualization environment from Synopsys. This tool will be used to debug the developed environments.
- UVM Reference Implementation
  The UVM API that is freely available at Accellera's website:
  http://www.accellera.org/downloads/standards/uvm

Bibliography

[1] Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Kluwer Academic Publishers, 2003.

[2] Stephen A. Edwards. Design and Verification languages. *Columbia University Computer Science Technical Reports*, (CUCS-046-04), 2004.

[3] Nisvet Jusic and Jan Nillson. Design and Verification languages. 2007.

[4] GE Moore. Cramming more components onto integrated circuits. 86(1):82–85, 1965.

[5] S Rosenberg and KA Meade. *A practical guide to adopting the universal verification methodology (UVM)*. lulu.com, 2010.

[6] Louis Scheffer, Luciano Lavagno, and Grant Martin. *EDA for IC system design, verification, and testing*. CRC Press, 2006.

[7] Chris Spears. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2007.