# Building a better VHDL testing environment

Joren Guillaume

Supervisors: Prof. Luc Colman, dhr. Hendrik Eeckhaut (Sigasi)
Counsellor: dhr. Lieven Lemingre (Sigasi)

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de industriële wetenschappen: elektronica-ICT

Department of Industrial Technology and Construction
Chairman: Prof. Marc Vanhaelst
Faculty of Engineering and Architecture
Academic year 2014-2015

# Usage restrictions

# Preface

*Dankwoord en zo verder.*

**Abstract**

# Contents

**List of Acronyms** **viii**

**I Problem and background** **1**

**1 Problem** **1**

**2 Introduction** **2**
2.1 Digital Electronics . . . . . 2
2.2 Hardware Description Languages . . . . . 3

**3 Current industry practices** **5**
3.1 Coverage . . . . . 5
3.2 Verification . . . . . 5
3.3 Methodologies . . . . . 6
3.4 Simulation tools . . . . . 6

**4 Software development & practices** **7**
4.1 Test Driven Development . . . . . 7
4.2 Continuous Integration . . . . . 8
4.3 JUnit . . . . . 9
4.4 Python . . . . . 9

**II Developing a framework** **10**

**5 Outlining** **10**
5.1 First drafts . . . . . 10
5.2 Making improvements . . . . . 11
5.3 Hudson-CI . . . . . 11

**6 The future of testing** **12**

**7 Conclusion** **13**

# List of Figures

1 Typical CMOS design flow . . . . . 4
2 Three step TDD design flow . . . . . 8

# List of Acronyms


| | | |
|---|---|---|
| CC | Code Coverage | 5 |
| CI | Continuous Integration | 8, 9 |
| CMOS | Complementary Metal Oxide Semiconductor | 2 |
| CRV | Constrained Random Verification | 5, 6 |
| DFF | D Flip-Flop | 3 |
| DT | Directed Testing | 5 |
| DUT | Device Under Test | 4 |
| FC | Funcational Coverage | 5, 6 |
| FSM | Finite State Machine | 5 |
| HDL | Hardware Description Language | 3, 5, 6 |
| IC | Intelligent Coverage | 6 |
| IEEE | Institute of Eletrical and Eletronics Engineers | 3 |
| NAND | Not AND | 2 |
| NOR | Not OR | 2 |
| OSVVM | Open Source VHDL Verification Methodology | 6 |
| RC | Revision Control | 8, 9 |
| RTL | Register Transfer Level | 3 |
| TDD | Test Driven Development | 7 |
| TFD | Test First Development | 7 |
| UUT | Unit Under Test | 4 |
| VHDL | VHSIC Hardware Description Language | 1, 3, 6, 10 |
| XML | eXtensible Markup Language | 9 |

# Part I
# Problem and background

## 1 Problem

Developing VHSIC Hardware Description Language (VHDL), like any code, is prone to error creation, either by user or by wrong product specifications. To ensure errors are weeded out before the more expensive production begins, the code must be subjected to rigorous testing. Currently, large and impractical tests are needed to fully test a product.

Because testing is such a time consuming process, finding errors often results in severe delays due to the need to both correct the error and test for others. Therefore it is in the best interests of both testing- and software engineers to write testbenches with maximal coverage, optimal readability and minimum time spent. It is also important to find and correct errors with minimal delay and maximal efficiency. This entire process should affect the least amount of code possible in order to minimize time spent retesting and modifying the code.

In this thesis, the objective is to provide VHDL users with an operating system independent framework. This framework will allow users to quickly and consistently create, modify and execute testbenches. To accomplish all of this, a number of tried and proven external tools will be gathered around a central Python script. This combination will ensure timely and automated building, testing and test report generation.

# 2 Introduction

## 2.1 Digital Electronics

There are two kinds of electronic appliances and circuits; digital and analogue. Digital electronics differ from analogue electronics in that they use a discrete set of voltage levels to transmit signals. The most common number of items in the set is 2, a level for one (commonly named *high*) and a level for zero (commonly named *ground* or *low*). The advantage of using a discrete number of levels rather than a continuous signal as is used in analogue electronics, is that noise generated by the environment, thermal noise and other interfering factors will have but a minor influence on the signal.

To process these discrete signals, electronics are made up of transistors that nowadays are formed in with the Complementary Metal Oxide Semiconductor (CMOS) technology. This technology uses both an *NPN* and a *PNP* transistor that work in a push-pull configuration. The p's and n's in NPN and PNP simply stand for *Positive* and *Negative*. They are made of positively and negatively doped lumps of semiconductor, usually silicon-dioxide ($SiO_2$). A transistor is basically a blockage on a track and depending on the force applied to its *gate*, it opens or closes the track.

In reality, the force takes the form of a current and an NPN transistor opens its gate when a positive current is applied. A PNP transistor, however, always leaves its gate open until a current is applied. This means that if we send the same signal to an NPN and a PNP transistor, with one of the signals inverted, we can open and close two parts of the entire circuit at the same time. This is useful to both direct a certain signal to ground and at the same time close its connection with the *source* (the power source). Hence also the name *Complementary* MOS, the NPN and PNP complement each other.

A certain combination of transistors is used to make *logic gates*. These logic gates make sure that only a certain combination of ones and zeroes at the inputs result in certain ones or zeroes at the outputs. For instance, one of the most common logic gates is a Not AND (NAND) gate. This gate has a number of inputs ranging from 2 to theoretically infinity (but practically 3 or 4) and only outputs a *low* signal if all of the inputs are *high* (digital one), otherwise its output is *at ground* or *low* (digital zero). The other most common logic gate is the Not OR (NOR) gate. This gate outputs a low signal if any of its inputs are high, otherwise it outputs a low.

A common mistake is to think that low or ground mean *zero voltage.* This is only partially true, the high signals are measured with ground as their reference. So a high signal of 1.8 V would be 1.8 V higher than ground, and could be considered to be at 1.8 V if ground is the theoretical zero. These logic gates are themselves combined to build higher-level blocks such as flip-flops, which are used to make registers and so on up to the entire chip design.

## 2.2 Hardware Description Languages

A Hardware Description Language (HDL) can be used to describe any one of the levels mentioned in the previous section, right down to the logic gate level. However, this last one is wholly unnecessary considering synthesis tools can produce superior logic gate-level layouts [7]. The level that uses certain blocks of logic gates to describe more complex behaviour is called the Register Transfer Level (RTL). Some blocks are standard implementations that have been widely used and are nearly fully optimized, such as memories, flip-flops and clocks. An RTL flip-flop implementation written in VHDL is shown here:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF is
    Port(D   : in std_logic;
         CLK : in std_logic;
         Q   : out std_logic;
end DFF;
architecture Behavioural of DFF is
begin
    process (CLK)
    begin
        if rising_edge(CLK) then
            Q <= D;
        end if;
    end process;
end Behavioural;
```

The Institute of Eletrical and Eletronics Engineers (IEEE) library provides a number of extensions on the original VHDL specification that allow a more realistic simulation and description of hardware behaviour. An *entity* defines the inputs and outputs of a certain building block, in this case the D Flip-Flop (DFF). The D stands for Delay, and it simply puts on its output, Q, that which was on the input, D, one clock period earlier. The *architecture*, in this case Behavioural, takes the description of an entity and assigns a real implementation to it. All *processes* are executed in parallel, this does not mean that all are triggered at the same time, nor do they take equally as long to finish, but it means that any process can be executed alongside any other process. In this case there is only one (nameless) process that describes the entire behaviour of the flip-flop. The process waits for the rising edge of the clock, which is a transition from zero to one, after which it schedules the value of D to be put on Q when the next rising edge appears.

This is a basic example of an entity, an architecture and a process. This flip-flop could be used in certain numbers to build a *register*, a collection of ones and zeroes (henceforth referred to as *bits*) that is used to (temporarily) store these values. The register could then be used alongside combinational logic to build an even bigger entity. The idea here is that small building blocks can be combined to produce vast and complex circuits that are nearly impossible to describe in one go. Adding all these layers together also creates a lot of room for error, and having a multi-level design makes it challenging to pinpoint the exact level and location of any errors. Therefore, it is paramount that all code on all levels is tested thoroughly, which is done by use of *testbenches* [1].

Testbenches are made up of code that takes a certain building block, the Unit Under Test (UUT) or Device Under Test (DUT). The testbench then puts a certain sequence of values on the inputs and monitors the outputs. If the device performs normally, the received output sequence should match a certain *golden reference*, the expected output sequence. In these testbenches it is good practice to observe how well a device performs if its inputs behave outside of the normal mode of operation. When all of these tests have finished and the output performs as expected, the device is ready to be put into production or further down the developmental process.

If a device is not tested properly and faults propagate throughout development, they can be very expensive to correct, especially at the stage of production, where a single photomask, used to "print" part of the layout, can easily cost $100,000 (€81,000 at time of writing)[2]. Therefore a large portion of time is spent writing and executing tests.

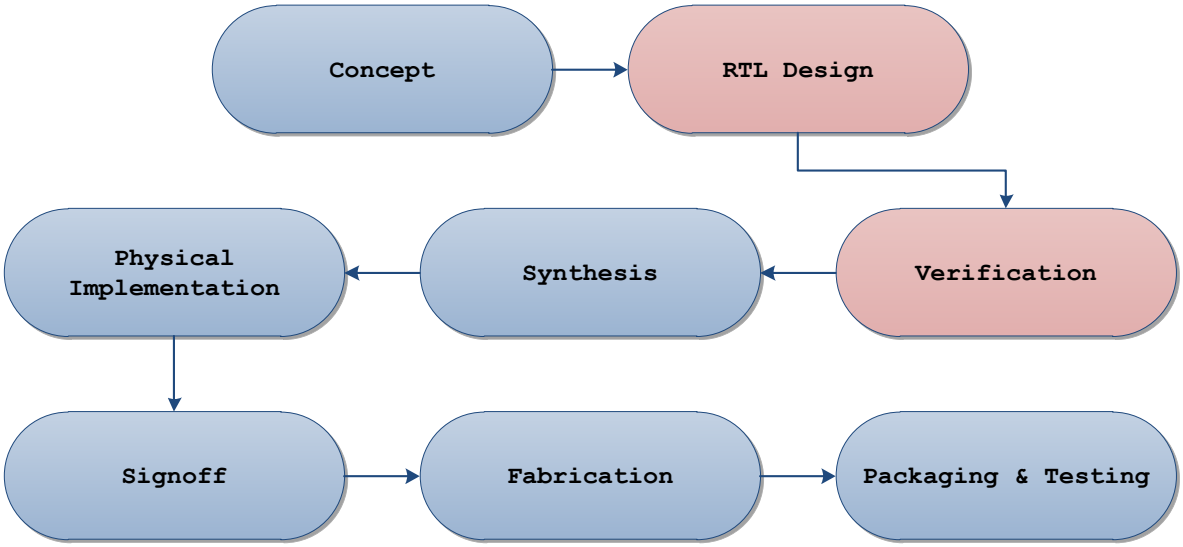


Figure 1: Typical CMOS design flow

# 3 Current industry practices

*Iets over asserts en dergelijke!*
As mentioned before, a number of practices exist to improve speed and quality of testing and coding. The bigger part of these practices are applied in the software development industry, where this is quite literally their entire business. Moving to HDLs, it stands to reason that code meant for synthesis may not be able to follow all of these best practices. However, the industry has formed several practices and *methodologies* to try and create a uniform verification process. The most well-known will be discussed in some detail below.

## 3.1 Coverage

Coverage is a generic term that is used to describe how fully a design has been tested on one aspect or another. There exist many tools for different coverage analyses, but in this section we will focus only on the types of coverage.

### 3.1.1 Code Coverage

In development, Code Coverage (CC) is a type of measurement to indicate how well the source code has been tested. With the use of a coverage report, unused blocks of code can be uncovered, these blocks might indicate unnecessary code or a bug. Imagine a Finite State Machine (FSM) with an unused reset state, this might indicate that the reset isn't functioning properly or there are no tests covering the reset. CC does not, however, provide any real functional analysis. It does not indicate any missing lines of code nor does it tell you whether the inputs and outputs behave properly.

### 3.1.2 Functional Coverage

Funcational Coverage (FC) is the practice of measuring whether design covers all possible use states it can possibly be in. Good practice is to include all possible cases, including corner cases, cases that cover multiple clock cycles, erroneous cases such as receiving an interrupt when processing or overflowing a buffer and so on.

## 3.2 Verification

### 3.2.1 Constrained Random Verification

Constrained Random Verification (CRV) is an industry practice where one or more inputs are generated randomly, within certain bounds or *constraints*. This practice was brought into use after designs grew too large for Directed Testing (DT) to support. DT has verification engineers write out very specific things they wanted to test, for instance, a reset pulse to verify the reset working correctly. CRV opposes this with the idea that for all behaviour to be tested properly in large designs, the amount of time spent writing and executing tests would simply become too great. It proposes a solution where inputs are generated randomly, within certain bounds, but in a sufficiently large quantity to have implicitly covered all scenarios. It is important to note that in DT, expected behaviour is directly tested, but in CRV it is likely to be the unexpected behaviour that gets tested too. This solved the long standing problem of testing any and all behaviour, including the unexpected.

### 3.2.2 Formal Verification

On top of the aforementioned, there are several more practices that have unique ways of verifying the properties of a design, but aren't used sufficiently to merit full detailing. Formal verification is such a practice, where the core idea is to mathematically prove the design from its specifications. The upside is that the design is completely verified, however, it it out of use because proving large designs is not only tedious but takes up ridiculous amounts of man-hours.

## 3.3 Methodologies

### 3.3.1 Open Source VHDL Verification Methodology

The Open Source VHDL Verification Methodology (OSVVM) is a set of packages that makes it easier for CRV and FC to be implemented in a project. It consists out of two packages, *Random.pkg* and *Coverage.pkg*. The new feature, which it dubs Intelligent Coverage (IC), redefines its FC model based on holes in the FC coverage and randomization. The advantage is obviously that 100% coverage is always within reach, even when the original draft did not achieve full coverage.[5]

### 3.3.2 Universal Verification Model

*To be written!*

## 3.4 Simulation tools

As mentioned in section 2 HDLs are used for developing hardware, and need to be tested and build as such. Like any other programming language, they need a dedicated compiler to fault-check the code and build the binaries. Unlike other programming languages, however, they need a simulator in order to verify the builds. There exist many compilers and simulators, almost none are exclusive for VHDL, almost all are dual-language and support some form of Verilog, a different HDL. However, many of them refuse to support the latest additions to the VHDL language specification, even the 2002 additions are hard to be found.

### 3.4.1 ModelSim

ModelSim is the simulator that was used for several reasons. First, a free student edition was available. Considering licenses outside of school can easily cost upward of $25,000 (€20,250 at time of writing) this made it ideal for any thesis or student related work. Second, ModelSim supports all versions of VHDL, which is the HDL we are processing. And last but not least, ModelSim is one of the industry's most used simulators, enabling a pool of experience to be consulted.

# 4 Software development & practices

During the making of this thesis, a number of approaches was investigated and some were put to practical use. In this chapter, the more useful and tried of the aforementioned are discussed in some detail.

## 4.1 Test Driven Development

Test Driven Development (TDD) is a proven development technique that has regained traction in the past decade, primarily through the efforts of Kent Beck[3]. This practice has proven to increase test coverage [6], decrease defect density [4] as well as improve code quality [4, 8]. The technique focuses on tests being created before the actual code. It is important to make certain distinctions before going more in depth on the used methods. The developing community has a great many practices, each with their own names and methods, and hardly none are mutually exclusive.

### 4.1.1 Unit Testing

To understand TDD, an basic knowledge of *Unit Testing* is required. In software testing, a unit test is a test designed to check a single unit of code. Ideally, this unit is the smallest piece in which the code can be divided. A unit test should always test only a single entity, and only one aspect of that entity's behaviour. This division in units has a number of benefits, one of the most important being that code is exceedingly easy to maintain. Furthermore, the division of the code makes changes, when needed, fast to be carried out and ensures that only the modified code needs to re-tested.

### 4.1.2 Test First Development

Another main component of TDD is Test First Development (TFD), a technique that has a developer writing tests first, before any code has been written. This method makes the developer think on what the code has to achieve, rather than what the specific implementation has to be. A key feature of a new test is that it has to fail during its first run. If not, the test is obsolete seeing as the functionality it tests has already been implemented. After being run for the first time (and failing), the developer implements just enough code to get the test to pass. Once the test succeeds, it is time for a new test.

### 4.1.3 Refactoring

The third pillar of TDD is refactoring. After the tests succeed, it is necessary to clean up the code. A well-done TFD implementation uses the bare amount of code needed to make the test pass, but this is of course usually not the best code possible. Refactoring means that you take existing code and modify only the code itself to perform better. This leaves both the input and output of the tests and code unchanged, only the way the code processes input is altered. It is important in this step to edit nothing in the test code or the outputs or inputs. Otherwise, either the test would behave differently or new tests would have to be written. The latter goes directly against the TFD principle.

### 4.1.4 Test Driven Development

Test Driven Development is a combination of the previously mentioned techniques. A Unit Test is written before any code, the test is then executed and should fail. After the failure, the most basic code to make the test pass is implemented, the test is then executed again and should pass. After

this first pass, the code written for the passing of the test, and only this code, is edited to perform and look better. This follows all steps mentioned above, a *Unit Test* is written according to *Test First Development* and is *Refactored* later.

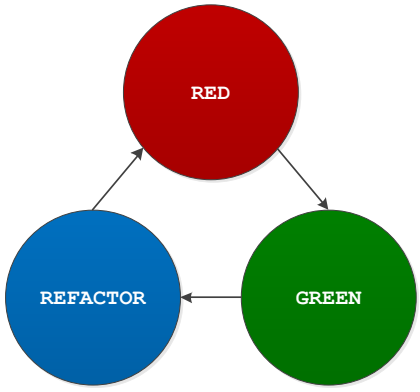


Figure 2: Three step TDD design flow

## 4.2 Continuous Integration

Continuous Integration (CI) is a software development technique in which developers upload the edits on the software, or their *build*, to a central server which then *continuously integrates* the code from multiple developers. This to prevent integration headaches when the code of multiple developers has diverged to such an extent that it would take much more time to make the edits work together than if they had been integrated early on.

### 4.2.1 Revision Control

There are many aspects to a properly maintained CI system, but one key aspect of overall programming has to be Revision Control (RC). Not to be confused with the "undo" button in your preferred editor, a good RC system does allow for any and all mistakes made over different edits to be undone with very little work. There exist many systems for revision control, but they all have in common that they track changes one way or the other, and most importantly that these changes can be undone.

### 4.2.2 Build Automation

A useful but not required aspect is build automation. Using a timer or trigger, the build automatically runs with the latest updates, preferably from an RC system. This way, the binaries are always up to date and the developer does not need to wait for compilation to run the latest tests. Although the latter is less imported in a proper CI system as will be discussed further on.

### 4.2.3 Test Automation

As the code is build at scheduled times, and testing is needed regardless it makes perfect sense to add a testing step to the automated build. Automating tests saves the developers yet another part of their time, thus freeing more for the actual development and debugging steps. A good CI system can read test reports, or at least some standard of reports.

### 4.2.4 Hudson-CI

Combining all of these practices saves developers a lot of time and, by extension, the company they might work for, a lot of money. Considering today's competitive market for software development, any edge that can be obtained is a plus. Even more so when plenty of free Continuous Integration servers exist that employ open or widely used standards. The CI solution that was investigated is Hudson-CI. Hudson-CI provides an extensive range of features, including everything listed above. The used features are:

- Timed and triggered building from an RC repository.
- Automated testing of said build.
- Humanly readable reports in the *JUnit* format.
- Graphical and statistical overview of test progress throughout builds.

## 4.3 JUnit

JUnit is an implementation for the Java programming language of the xUnit specification, which defines several key components for a testing framework. Considering the relevance here is not the framework itself but the format used for its reports, details about its implementation and use will not be discussed.
The JUnit reports are written in eXtensible Markup Language (XML), which is a type of language used for formatting data. The JUnit format is supported by numerous (open source) tools, such as Eclipse and the aforementioned Hudson-CI, which makes it an ideal candidate for further investigation.

## 4.4 Python

Python is a high-level computer programming language that first appeared in 1991, with the third major revision being released in 2008. It supports object-oriented and structured programming, and is easy to use as a scripting language. A great feat of Pythons community is that there are many, many (open source) libraries available for just about any function that comes to mind. This is on top of the already impressive amount of libraries Python itself supports. In addition to this, Python has all of its standard features explained in great detail with good examples in its online documenting system. The combination of these features allows any programmer, even with very little know-how, to quickly put together anything that comes to mind.

# Part II
# Developing a framework

In part I different aspects from both the software and VHDL development worlds were discussed, each in varying levels of detail. In this part bits and pieces of each subject are combined to form a new whole, a VHDL development framework.

## 5 Outlining

Before work begins on developing the framework, there need to be some bounds set and goals to work towards. As VHDL development is done in both windows and linux environments, it only makes sense to keep the whole framework as platform independent as possible. To add to this, the whole needs to be developed in the short span of under a year by one developer who has had little experience doing so. The language chosen thus must be easy to master and work on all platforms with as few as possible modifications.

Furthermore, software practices will have to be applied to maintain clean code and a steady development process. To this end, work started with unit testing in mind. Parts of the VHDL testbench are seperated to be executed independently in an automated fashion.

### 5.1 First drafts

In section 3 some current industry practices were discussed and one of the very basic VHDL testing methods was found to be assertions. In order to remain as broad as possible, work started with processing these assertions first. The assertions thus need to be found inside the VHDL testbench, seperated and saved in individual files to be executed.

One of the main problems that quickly arose was: *Where are the assertions located?*. To prevent overcomplicating things, the assumption was made that all tests are fitted inside their own procedure or function and that these would be called from inside the architecture body. All that would remain was to find the architecture body, find the keyword *begin* and extract every line from the body.

When the script was run on a file, a few problems quickly came in view:

- Tests can be a lot more complicated than first assumed
- Procedures and functions aren't that easily made to contain fully working code
- Writing tests this way is actually *more* time consuming

To try and counter the problems, making use of a standardized function library was proposed. This way, verified code could be integrated which reduced the workload when writing the testbench. Making use of this library would also bring a degree of uniformity to the testbenches, which would help with readability.

## 5.2 Making improvements

As mentioned above, one of the first improvements was a standardized function library. To be of any use, this library should contain functions and procedures that are either used extensively throughout the developmental process, have a great potential of being used or that, when created on their own, would impose a significant workload on the developer. Considering the large percentage of the industry still working in the VHDL-93 specification, the library would need to be compatible with this version. However, to ensure future useability and steer development towards the newer versions, i.e. 2002 and 2008, these will need to be supported as well.

A second improvement, yet undiscussed, would be maintaining a clean setup. Throughout the first testing, it became apparant that when multiplying the amount of testbenches that are executed, the files around them are multiplied in the same way. Massive clutters of 'temporary' files and folders would quickly pollute the workspace, and thus keeping track and removing unneeded files would be of the essence, especially considering later modifcations (see section 5.3).

## 5.3 Hudson-CI

# 6 The future of testing

*Iets over VHDL features, verbeteringen etc*

# 7 Conclusion

# References


[1] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer US, 2000.

[2] Neil C. Berglund Charles M. Weber and Patricia Gabella. "Mask Cost and Profitability in Photomask Manufacturing: An Empirical Analysis". In: *IEEE Transactions on Semiconductor Manufacturing* 19.4 (2006), pp. 465–474.

[3] Martin Fowler. *UnitTest*. 2014. URL: `http://martinfowler.com/bliki/UnitTest.html` (visited on 30/06/2014).

[4] J.C. Sanchez L. Williams and E.M. Maximilien. "On the Sustained Use of a Test-Driven Development Practice at IBM". In: *AGILE 2007 Conference (AGILE 2007), 13-17 August 2007, Washington, DC, USA*. AGILE '07. IEEE Computer Society, 2007, pp. 6–14. ISBN: 0-7695-2872-4. DOI: `10.1109/AGILE.2007.43`.

[5] Jim Lewis. *OSVVM's Intelligent Coverage is 5X or More Faster than SystemVerilog's Constrained Random*. 2013. URL: `http://www.synthworks.com/blog/2013/05/24/osvvms-intelligent-coverage-is-5x-or-more-faster-than-systemverilogs-constrained-random/` (visited on 25/07/2014).

[6] M. Siniaalto and P. Abrahamsson. "A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage". In: *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–284. ISBN: 0-7695-2886-4. DOI: `10.1109/ESEM.2007.2`. URL: `http://dx.doi.org/10.1109/ESEM.2007.2`.

[7] Deepak Kumar Tala. *VHDL Introduction*. 1998. URL: `http://www.asic-world.com/vhdl/intro1.html` (visited on 30/01/2014).

[8] B. Thirumalesh and N. Nachiappan. "Evaluating the efficacy of test-driven development: industrial case studies". In: *ISESE*. Ed. by Guilherme Horta Travassos, José Carlos Maldonado and Claes Wohlin. ACM, 2006, pp. 356–363. ISBN: 1-59593-218-6. URL: `http://dblp.uni-trier.de/db/conf/isese/isese2006.html#BhatN06` (visited on 23/01/2007).