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

Abstract

Using the Document Class phdsymp.cls

Joris Thybaut

Supervisor(s): Eric Laermans, Luc Dupré

Abstract—**This article explains how to use the LaTeX style recommended for the Proceedings of the FTW PhD Symposium. The article is itself an example of the phdsymp.cls style in action.**

Keywords—**Style file, LaTeX, FTW PhD Symposium**

I. Introduction

THE Symposium covers a wide range of topics and reflects the diversity of research activities at our Faculty, such as:

- Applied Physics
- Architecture
- Automation
- ...

Authors who have prepared their articles using LaTeX can get them formatted in the style we would like to recommend for the Proceedings of the Symposium. The style file `phdsymp.cls` can be used together with the bibliography style file `phdsymp.bst`.

We recommend a *double column* style as this makes the article easier to read. The column width is 21 pica (approximately 90 mm). In this sample file you will find examples for the layout of displayed equations, theorems, tables, figures, etc.

II. How to Use the File phdsymp.cls

A. General Information

This style file has been written so to allow, with very few changes, the formatting of input that is suitable for the LaTeX `article` style. First, the `phdsymp.cls` style file has to be selected with a command of the form

```
\documentclass[twocolumn]{phdsymp}
```

The default font size is 10 points.

The Symposium Proceedings will not include author affiliations below or beside the name(s) of the author(s); instead, use the command `\thanks{...}` to list addresses. Note that the `\thanks{..}` command in the title no longer produce marks: the thanks-footnote should therefore be self-contained, with address and name of the author(s).

The command "`\PARstart{X}{YYY} ZZZ`" produces a large letter `X` at the beginning of the paragraph. The string `YYY` will be automatically changed to capital letters.

The bibliography style file `phdsymp.bst` allows BibTeX to include the references from the chosen bibliography file(s) according to the format recommended for the Symposium Proceedings.

Footnotes produce a footnote mark as usual.[1]

J. Thybaut is with the Chemical Engineering Department, Ghent University (UGent), Gent, Belgium. E-mail: Joris.Thybaut@UGent.be .

[1]The footnote is indicated by a footnote mark

In figure **??** we can see an example for the definition of the title page and of the main commands needed to compile a LaTeXfile with phdsymp.cls.

```
\documentclass[twocolumn]{phdsymp}

\usepackage{times}

\begin{document}

\title{Using the Document Class phdsymp.cls}

\author{Joris Thybaut
   \thanks{J.~Thybaut is...}}

\supervisor{Eric Laermans, Luc Dupr\'e}

\maketitle

\begin{abstract}
This article ...
\end{abstract}

\begin{keywords}
Style file...
\end{keywords}

\section{Introduction}
\PARstart{T}{he} Symposium ...

\bibliographystyle{phdsymp}
\bibliography{bib-file}

\end{document}
```

Fig. 1. Input used to produce this paper.

B. Additional Changes

Most changes resulting from the use of phdsymp.cls should be transparent to the user. For instance, captions for figures and tables have been modified. Caption of tables, however, should be defined before the table item.

B.1 Environments

The environments for theorem, propositions, lemmas, etc. can be defined with the usual LaTeX [**?**], [**?**] command `\newtheorem{..}{..}`. The proof environment is already defined.

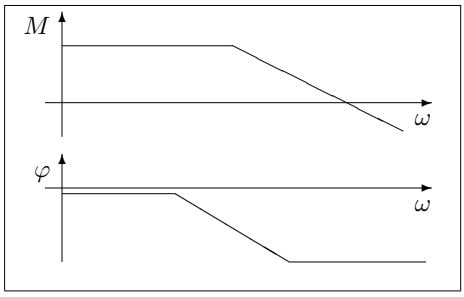

Fig. 2. This is a sample figure. The caption comes after the figure.

TABLE I
THE CAPTION COMES BEFORE THE TABLE.

	title page	odd page	even page
onesided	leftTEXT	leftTEXT	leftTEXT
twosided	leftTEXT	rightTEXT	leftTEXT

Theorem 1 (Theorem name) Consider the system

$$\begin{aligned} \dot{x} &= A.x + B.u \\ y &= C.x + D.u \end{aligned} \tag{1}$$

If A is stable, then the pair $\{A, B\}$ is stabilizable. Moreover, this holds for any B.

Proof: The proof is trivial. ■

III. OPTIONAL FORMATTING

When you are happy with the *ultimate unequivocal final version*, you may perform following additional changes, although this is certainly not necessary.

A. "Hard-Coding" Symbolics

Change the symbolics so that the file actually contains the reference numbers *i.e.* "`... \cite{fred:88} ...`" should be changed to "`... [3] ...`". One author (who used the style file) did a smart thing *after* he had decided upon a final version. He put his `\cite{..}` command and other symbolics on a line on their own and commented them out (from the formatting) by putting a `%` sign before each symbolic. Then, on the next line he just inserted the copy-matching numerical, like this:

```
Well, according the Fred Bloggs
%\cite{fred:88}
[24]
the value of $\alpha$ should be even
greater than what we think it should be.
```

Thus, *he* knows he put in the correct (copy-matching) numerical and the publishing staff can send him back an author-proof that correctly matches his submission. This is not so useful here as we do not expect you to send the LaTeXfile, but a PDF-version of your article.

The above also applies to the referencing of table and figures (and any "auto-numbering" feature, standard or synonymous with your system).

Figure captions can be part of the text (in between paragraphs) like this:

```
And in Fig. 3 we see that the
value of $\alpha$ increases exponentially.

   Fig. 3\quad This is the caption for
   figure 3 showing some $\alpha$.

And after the caption we continue on
with the next paragraph, like this.
```

In essence, by you actually putting in the *correct copy matching* numericals so that no problems arise with incomplete files being sent to the transactions (the wrong `*.bib`, `*.bbl` files, the wrong versions of figures etc). Also, and more importantly, the numbers that are on your hard-copy (and in the reviewer's hands) will be the same ones that you receive in your author proof. Again, this is not so useful here as we do not expect you to send the LaTeXfile, but a PDF-version of your article.

B. Including the Bibliography into the LaTeX Source File

You can reduce the number of files you have to send to the publishers in the following way. Run BibTeX on the `*.aux` file. This creates a `*.bbl` file: include this into your LaTeX source file at the place where you defined the `\bibliography{..}` command and comment this command out. Remove the `*.bbl` file. Then, your LaTeX file will include all the necessary information about your bibliography and no `*.bbl` or `*.bib` file will be needed.

This may seem like an awful lot of work... but not really.. This will allow to process your paper quickly and efficiently, and assure you that what you send in *will* actually be sent back to you without mistakes (cites, refs etc.).

However, this would only have been useful if you had been requested to send the LaTeXfile itself instead of a PDF-version of the article.

IV. CONCLUSIONS

This sample article has presented the style file phdsymp.cls This file can be especially useful in preparing articles for submission to the FTW PhD Symposium.

ACKNOWLEDGMENTS

The authors would like to acknowledge the suggestions of many people.

REFERENCES

[1] Leslie Lamport, *A Document Preparation System: LaTeX, User's Guide and Reference Manual*, Addison Wesley Publishing Company, 1986.
[2] Helmut Kopka, *LaTeX, eine Einführung*, Addison-Wesley, 1989.
[3] D.K. Knuth, *The* TeX*book*, Addison-Wesley, 1989.
[4] D.E. Knuth, *The* METAFONT*book*, Addison Wesley Publishing Company, 1986.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

Part I
Problem and background

1 Problem

Developing 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. For full product testing by conventional means, large and impractical tests are needed.

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 engineers and software engineers to find and correct errors with minimal delay and maximal efficiency. This process should affect the least amount of code possible as to minimize time spent retesting and modifying the code.

In this thesis, a number of mechanics are used to optimize both testing and coding. Based loosely off of Test Driven Development (TDD), tests are written to both function and be tested independently to maximize test coverage with minimal effort. To this end, a library with often used functions and other useful code is made available. Alongside of it is a tool, written in Python, to process tests independently and represent the results in a quick and easy to read format.

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 or SO_2). A transistor is basically a blockade on a track and depending on the force applied to its Gate, it opens or closes the track.

In reality, the force takes 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 it so 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 *nand* gate (a *not and* 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 *nor* gate (a *not or* 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 Volts would be 1.8 Volts higher than ground, and could be considered to be at 1.8 Volts 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 these levels, right down to the logic gate level. However, this last one might not be a good idea considering most synthesis tools can produce superior logic gate-level layouts [1]. The level that uses certain blocks of logic gates to describe more complex behaviour is called the *Register Transfer Level* or RTL. Some blocks are standard implementations that have been widely used and nearly fully optimized, such as memories, flip-flops and clocks. An RTL flip-flop implementation 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 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 or DFF. The D stands for Delay, and it simply puts on its output, Q, that which was on the input, D, one clock cycle 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. It waits for the rising edge of the clock, which is a transition from zero to one, and then 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 somewhat difficult to pinpoint the exact level and location of any errors. Therefore it is paramount that all code on all levels is tested thoroughly, this is done by use of *testbenches* [2].

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 also interesting 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.

It is easy to see that if a device is not tested properly and faults propagate it 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 [3]. Therefore a large portion of time is spent writing and executing tests[4]. Practices to improve both the speed and quality of testing and coding exist in a large number, but the focus chosen in this thesis is *Test Driven Development* (TDD). This practice has proven to increase test coverage [5], decrease defect density [7] as well as improve code quality [7, 8].

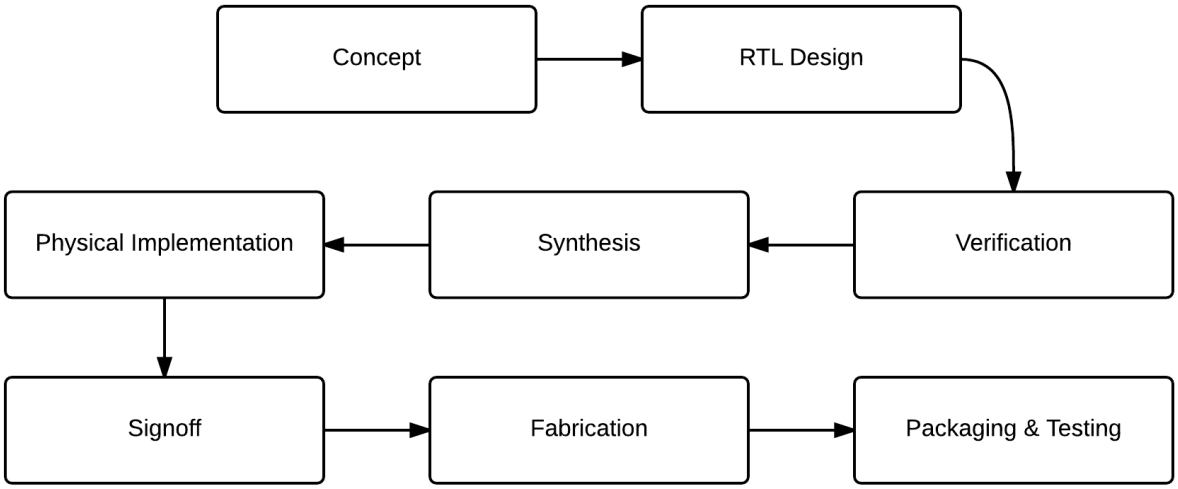

Figure 1: Typical CMOS design flow

2.3 Test Driven Development

Test Driven Development is an older development technique that has regained traction in the past decade, primarily through the efforts of Kent Beck [**?**]. The technique focuses on tests being written 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.

2.3.1 Unit Testing

To understand TDD, an understanding 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 only test 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.

2.3.2 Test First Development

Another main component of TDD is *Test First Development*, 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 when run for the first time, 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.

2.3.3 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.

2.3.4 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.

3 Implementation

During the making of this thesis, a number of approaches was investigated and some were put to practical use. Ultimately, the most practical and straightforward approach was to implement a form of Unit Testing.

4 Mission and objectives

The goal of this thesis is to ease development and testing of VHDL code, with a focus on testing and test reporting. Practically this means the development of several tools which can be independently used and each have their own merit.

- A testbench parser that ensures independent testing and proper test report generation.
- A library with many widely-used functions as to speed up programming, as well as functions to make reporting possible.

- A test report that can easily be read but still holds a significant amount of detail on errors and successes.

References

[1] http://www.asic-world.com/vhdl/intro1.html

[2] Writing Testbenches: Functional Verification of HDL Models

[3] Mask Cost and Profitability in Photomask Manufacturing: An Empirical Analysis

[4] Citation needed !!

[5] A comparative case study on the impact of test-driven development on program design and test coverage

[6] http://martinfowler.com/bliki/UnitTest.html

[7] A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry

[8] Evaluating the Efficacy of Test-Driven Development

Part II
Appendix