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.

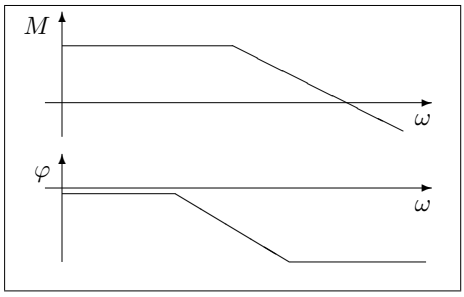

Fig. 2. This is a sample figure. The caption comes after the figure.

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. 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 find and correct errors with minimal delay and maximal efficiency. This process should affect the least amount of code possible in order 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 function and be tested independently. In order to maximize coverage and keep the time spent writing tests to a minimum, 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 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 *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 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 [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 are 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 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* [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 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 [3]. Therefore a large portion of time is spent writing and executing tests[4]. A large number of practices to improve both the speed and quality of testing and coding exist, 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].

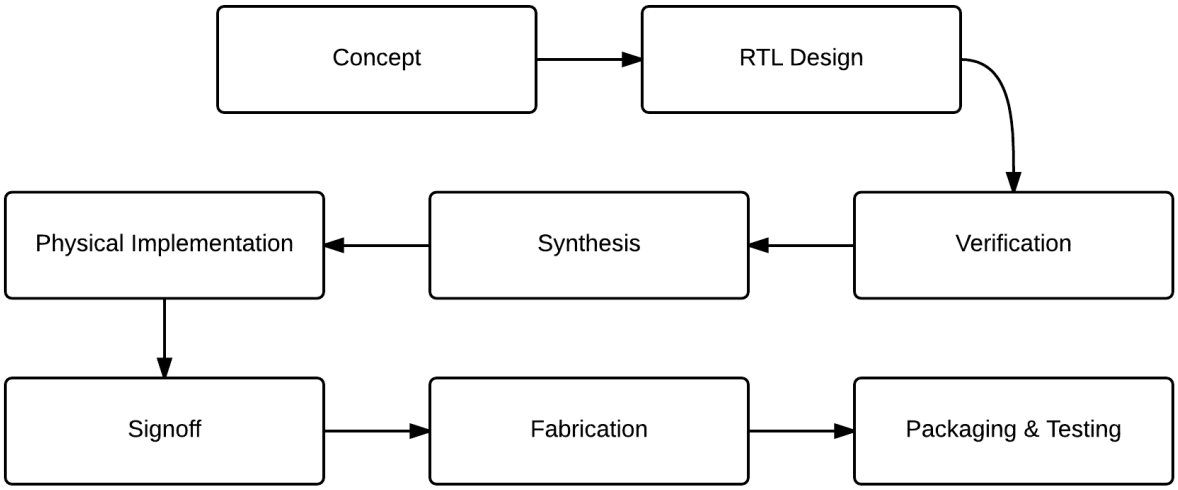

Figure 1: Typical CMOS design flow

2.3 Test Driven Development

Test Driven Development is a proven development technique that has regained traction in the past decade, primarily through the efforts of Kent Beck [**?**]. 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.

2.3.1 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.

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 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.

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 Testing VHDL

4 Exploring solutions

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. Alongside, an extensive library, written in VHDL, was used to ease the building of testbenches. Finally, a central server was installed that repeatedly and automatically ran the script and read the resulting test outputs.

4.1 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.

4.1.1 Revision Control

There are many aspects to a properly maintained CI system, but one key aspect of overall programming has to be *revision control*. 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.

4.1.2 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.

4.1.3 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.

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 used throughout this thesis is *Hudson-CI*. Hudson 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.

4.2 Modelsim

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.

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 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, making the research done of practical use.

5 The future of testing

6 Conclusion

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

Code

```
#########################################################################################################################################
###     Author:     Joren Guillaume
###
###     Changelog
###     2014.08.11 - Formatting, filled in code for functions
###     2014.08.17 - More filling in code
###     2014.08.21 - Added logwrite()
###     2014.08.25 - Editing functions, use of global variables
###     2014.09.16 - Bugfixing, overhaul, extended log function
###     2014.09.17 - Overhaul of setup() and logwrite(), filling of parse_source() to actually parse a source
###                  Program now saves every sourcefile seperately, every test in it's unique architecture
###                  Files are executed seperately now, output is captured
###     2014.09.23 - Started expanding format()
###     2014.09.25 - Expanded format() further, preliminary version finished. Started on xmlwrite()
###     2014.09.26 - Prelimenary working version. Produces actual xml report with (all passed) test reports
###     2014.10.01 - Introduced format2 and xmlwrite2 from previous version "test_parser".
###                  Testbenches are now required to use the vhdlUnit.vhd library for reporting (reportback() function)
###                  vhdlUnit.vhd is expected to be in the same directory as testbench_parser.py
###                  Small error fixing in handling multiple testbenches named the same (or actually being the same)
###     2014.10.07 - Small all-around changes, including the commandline options, file handling, file locating et
###
#########################################################################################################################################
###
###     ToDo:
###         - Use of Try/Except for error catching
###         - Expand commentary to full descriptions
###         - (Actually detect whether tests pass!!)
###
############  FUNCTIONS  ################################################################################################################
#   setup()                 ## Sets up (temporary) files, sets global vars, processes commandline arguments with argparse
#   logwrite()              ## Writes things to the logbook, errors, completed jobs etc.
#   get_path(path)          ## Returns absolute path if not already absolute path, otherwise return argument unchanged
#   setup_parser()          ## Prepares the parser to accept correct commandline arguments
#   make_tempdir()          ## Creates the temporary working directory
#   parse_source()          ## grabs sourcefile, extracts needed code, arranges functions & procedures
#   test_format()           ## arranges found functions & procedures in their own executable files
#   parse_tests             ## grabs processed source/files, executes & captures output
#   format()   (Deprecated)## grabs output, formats output
#   xmlwrite()(Deprecated)## grabs processed output, converts to JUnit compatible XML file
#   format2()               ## grabs output, formats output
#   xmlwrite2()             ## grabs processed output, converts to JUnit compatible XML file
#   get_time()              ## extracts the passed time from a modelsim time notice
#   cleanup()               ## removes temporary files & directories
##############     END     ################################################################################################################

import sys
import os
import shutil
import tempfile
import string
import random
import argparse
import time
from junit_xml import TestSuite, TestCase
```

```
## Sets up (temporary) files, sets global vars, processes cmdline arguments with argparse
##
# args is the arguments part of an argparse.ArgumentParser
#     The possible expected arguments are c,l,f,d,s; for further explanation see setup_parser()
##
# Tempname_t is the random name used throughout the script, it has no constraints for being different
##
# Foldername_t is the path to the folder that will contain all permanent logs and results
##
def setup(args, tempname_t=None, foldername_t=None):
    if (tempname_t == None):                                        # If tempname was not given with the function, use global name
        tempname_t = tempname
    if (foldername_t == None):                                      # If foldername was not given with the function, use global name
        foldername_t = foldername

    if not args.cmdline:
        folderpath_t = os.path.join(os.getcwd(), 'VHDL_TDD_Parser')
        if (not os.path.isdir(folderpath_t)):                       # Make the folder that contains all non-deleted files from all time, if non-
            os.makedirs(folderpath_t)
            logbuffer('n','Created working directory \'VHDL_TDD_Parser\' for the first time.')
        outputdir = time.strftime('%Y.%m.%d - %H.%M - ') + tempname_t
        foldername_t = os.path.join(folderpath_t, outputdir)
        if (os.path.isdir(outputdir)):                              # Make the folder for the current run
            logbuffer('n','Output directory \'' + tempname_t + '\' already existed.')
        else:
            os.makedirs(foldername_t)
    else:
        folderpath_t = os.path.join(os.environ['USERPROFILE'], 'VHDL_TDD_Parser')
        if (not os.path.isdir(folderpath_t)):                       # Make the folder that contains all non-deleted files from all time, if non-
            os.makedirs(folderpath_t)
            logbuffer('n','Created working directory \'VHDL_TDD_Parser\' for the first time.')
        outputdir = time.strftime('%Y.%m.%d - %H.%M - ') + tempname_t
        foldername_t = os.path.join(folderpath_t, outputdir)
        if (os.path.isdir(outputdir)):                              # Make the folder for the current run
            logbuffer('n','Output directory \'' + tempname_t + '\' already existed.')
        else:
            os.makedirs(foldername_t)
    return foldername_t

## Stores log reports in a buffer until the logfile is made
##
# level is the criticality of the logwrite. It can be notice, warning, severe, critical or unknown
#     Some levels are not (yet) used and may be obsolete
##
# message is the actual logmessage, with no timestamps or level
##
# tempname_t is the unique identifier for the testbench run
##
# args_t is the argument parser object generated in setup_parser()
##
def logbuffer(level='n',message ='No message given.', tempname_t=None, args_t=None):
    global logs_buffer                                                                  # Uses the global variables logs_buffer and logstarted
    if logstarted:                                                                      # This function cannot work with no such variables
        logwrite(level, message, tempname_t, args_t)
    else:
        logs_buffer.append([level, message, time.strftime("%Y.%m.%d - %H:%M:%S"), ])

## Writes things to the logbook: errors, completed jobs etc.
## Logwrite is fully explained above near logbuffer
## Logwrite cannot function without global variables logstarted and logs_buffer
def logwrite(level='n',message ='No message given.', foldername_t=None, tempname_t=None, args_t=None):
    if (tempname_t == None):
        tempname_t = tempname
    if (args_t == None):
        args_t = args
    if (foldername_t == None):
```

```
        foldername_t = foldername
    dest = ''
    if (args_t.log):
        dest = get_path(args_t.log)
    else:
        dest = foldername_t

    #FORMAT: Timestamp \tab Level \tab Message
    levels = {'n': 'notice', 'w' : 'warning', 's': 'severe', 'c': 'critical', 'u' : 'unknown'};
    logfile = open(dest + os.sep + 'logfile.txt', 'a+')

    logfile.seek(0)
    first_char = logfile.read(1)
    if first_char != '#':                                                    # Check for any file contents and create header if none
        logfile.write('################################################################################\n')
        logfile.write('#######################␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#######################\n')
        logfile.write('#######################␣LOGFILE␣FOR␣' + tempname_t + '␣#######################\n')
        logfile.write('#######################␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#######################\n')
        logfile.write('################################################################################\n\n')
        logfile.write('YYYY.MM.DD␣-␣HH:MM:SS␣␣-␣␣␣LEVEL␣␣␣␣-␣␣MESSAGE\n')
    else:
        logfile.seek(2)

    global logstarted
    if (not logstarted):
        logstarted = True
        for command in logs_buffer:
            logfile.write(command[2] + '␣␣-␣␣' + (levels[command[0]] + '\t').expandtabs(5) + '-␣␣' + command[1] + '\n')
                                                                             # Print the message with information on time and level
    logfile.write(time.strftime('%Y.%m.%d␣-␣%H:%M:%S') + '␣␣-␣␣' + (levels[level] + '\t').expandtabs(5) + '-␣␣' + message + '\n')
    logfile.close()

## Returns absolute path if not already absolute path, otherwise return argument unchanged
def get_path(path):
    if (not os.path.isabs(path)):
        path = os.path.abspath(path)
    return path

## Prepares the parser to accept correct commandline arguments
def setup_parser():
    parser = argparse.ArgumentParser(  description='VHDL␣testbench␣to␣TDD␣parser'
                                     , formatter_class=argparse.ArgumentDefaultsHelpFormatter)
        # The -c/--cmd argument is to specify the script being called from the commandline.
        # The flag is stored in 'cmd', default value is 'False'
    parser.add_argument('-c', '--cmd'
                        , help='specifies␣script␣being␣called␣from␣commandline,␣not␣automated'
                        , action = 'store_true', dest='cmdline', default=False)
        # The -d/--dest argument is to specify a custom folderpath for the log.
        # The flag is stored in 'log', default value is 'None'
    parser.add_argument('-d', '--dest'
                        , help='specifies␣a␣custom␣folder␣for␣the␣log'
                        , action = 'store', dest='log', default=None)
        # The -f/--file argument is to specify that the -l/--list is a file/files containing a list of .vhd files.
        # The flag is stored in 'file', default value is 'False'
    parser.add_argument('-f', '--file'
                        , help='specifies␣-l/--list␣is␣a␣file␣with␣a␣list'
                        , action = 'store_true', default=False)
        # The -l/--list argument is to specify the list of .vhd files to be processed.
        # The flag is stored in 'list', no default value considering there NEEDS to be at least 1 .vhd file
    parser.add_argument('-l', '--list'
                        , help='specifies␣the␣list␣of␣.vhd␣files␣to␣be␣processed␣--␣note:␣ONLY␣.vhd␣files'
                        , action = 'store'
                        , nargs = '+', required=True, metavar='path')
        # The -m/--method argument is to specify which method of writing was used in testbench creation.
        # The flag is stored in 'list', no default value considering there NEEDS to be at least 1 .vhd file
    parser.add_argument('-m', '--method'
```

```
                        , help='specifies what method was used to write the testbench. '
                        , action = 'store'
                        , default='line', choices=['line', 'startstop', 'partitioned'])
        # The -p/--precompiled argument is to locate a precompiled library that will be needed during simulation.
        # The flag is stored in 'list', no default value considering there NEEDS to be at least 1 .vhd file
    parser.add_argument('-p', '--precompiled'
                        , help='specifies the location of the precompiled dependencies, requires a full path.'
                        , action = 'store'
                        , dest='prepath', metavar='path', default=None)
        # The -v/--version argument is to compile in a different VHDL version other than 2008.
        # The flag is stored in 'version', default value is 2008
    parser.add_argument('-v', '--version'
                        , help='specifies a different VHDL version, default is 2008'
                        , action = 'store'
                        , default='2008' , metavar='version')
#         # The -s/--script argument is to specify the testbench(es) use the scriptstart/scriptend comments
#         # The flag is stored in 'script', default value is 'False'
#     parser.add_argument('-s', '--script'
#                         , help='specifies the testbenches use the ScriptStart and ScriptEnd comments'
#                         , action = 'store_true', default=False)

    return parser

## Creates the temporary working directory
def make_tempdir():
    char_set = string.ascii_uppercase + string.digits           # Generates the character set used: letters & numbers, lower & uppercase
    tempname_t = ''.join(random.sample(char_set*8,8))           # Save temporary name for deletion later
    tempdir_t = tempfile.gettempdir() + os.sep + tempname_t

    while(os.path.isdir(tempdir_t)):                            # Make sure directory doesn't exist yet
        tempname_t = ''.join(random.sample(char_set*8,8))
        tempdir_t = tempfile.gettempdir() + os.sep + tempname_t

    os.makedirs(tempdir_t)                                      # Create temporary working directory
    return tempdir_t, tempname_t


## grabs sourcefile, extracts needed code, arranges functions & procedures
def parse_source(path, method=None):

    if (method==None):
        method=args.method
    archstart = False

    try:
        source = open(path)                                     # Open the file
    except:
        logbuffer('c','Could not open file ' + str(path) + ', ignoring.')
        return ['ERROR', path]
    line = source.readline()
    line_lower = line.lower()                                   # VHDL is case sensitive, but using lowercase is easier to target identifiers such as 'architecture'
    words = line_lower.split(' ')
    header, archheader, body, footer = '','','',''
    entname, archname = '',''
    templist = []
    tests = 0                                                   # Count the tests in the body

    while not archstart:                                        # Find the entity and architecture name, used in executing the testbenches
        if (line == ''):
                logwrite('c','Reached end of source file without an architecture in file ' + str(path) + '.')
                return ['ERROR', path]
        if words:
            if (words[0] == 'entity'):
                entname = words[1]
            if (words[0] == 'architecture'):
                archstart = True
```

```
            archname = words[1]
            archheader += line
        else:
            header += line
    line = source.readline()
    line_lower = line.lower().strip()
    words = line_lower.split(' ')

if (method=='startstop'):                                   # Check whether scriptstart/end is being used with 'script'
    scriptstart, scriptend = False, False
    while (not scriptstart):
        if (line == ''):
            logwrite('c','Reached end of source file without encountering \'--scriptstart\' despite -s flag specified in file ' + str(path) + '.')
            return ['ERROR', path]
        archheader += line
        if (line_lower.strip() == '--scriptstart'):         # Find the start of the script, add all lines before the start to the header
            scriptstart = True
        line = source.readline()
        line_lower = line.lower()

    while (not scriptend):                                  # All lines between scriptstart and scriptend are placed in the body
        if (line == ''):
            logwrite('c','Reached end of source file without encountering \'--scriptstop\' despite -s flag specified in file ' + str(path) + '.')
            return ['ERROR', path]
        elif (line_lower.strip() == '--scriptend'):
            scriptend = True
            footer += line
        else:
            body += line
        line = source.readline()
        line_lower = line.lower()

    footer += line                                          # All remaining lines are placed in the footer
    for line in source:
        footer += line
    source.close()
elif (method =='line'):
    archbody, archend = False, False
    beginwords = ['function','procedure','for','while','if','process','component',]
    depth = 0

    while not archbody:
        if (line == ''):
            logwrite('c','Reached end of source file with incorrect architecture body (found no begin) in file ' + str(path) + '.')
            return ['ERROR', path]
        if words:
            if depth == 0 and words[0] == 'begin':          # If the 'begin' is the begin of the architecture body
                archbody = True
            elif words[0] in beginwords:
                depth += 1
            elif words[0] == 'end':
                depth -= 1
        archheader += line
        line = source.readline()
        line_lower = line.lower().strip()
        words = line_lower.split(' ')

    depth = 0
    process_start, process_begin = False, False

    while not archend:
        if (line == ''):
            logwrite('c','Reached end of source file with incorrect architecture body (found no end) in file ' + str(path) + '.')
            return ['ERROR', path]
        if not process_begin:
            if words:
```

```
                if not process_start:
                    if 'process' in words:                          # Finding the process that envelopes the functions
                        process_start = True
                else:
                    if depth == 0 and words[0] == 'begin':          # If the 'begin' is the begin of the process
                        process_begin = True
                    elif words[0] in beginwords:                    # Else, check for 'depth' changing words
                        depth += 1
                    elif words[0] == 'end':
                        depth -= 1
            archheader += line
        else:
            if words:
                if (line_lower in ['end architecture;', 'end ' + archname + ';','end architecture' + archname + ';']):
                    archend = True
                    footer += line
                elif (words[0] in ['end','wait;']):                 # 'wait;' and 'end process;' don't count as tests
                    footer += line
                else:
                    body += line
        line = source.readline()
        line_lower = line.lower().strip()
        words = line_lower.split(' ')

    for line in source:                                             # All remaining lines, if any, are placed in the footer
        footer += line
    source.close()

elif(method=='partitioned'):
    in_test = False
    temp_lines = ''

    while (line != ''):
        if (in_test):
            if (len(line) > 4):
                if (line_lower[0:5] == '--end'):                    # Find the start of the script, add all lines before the start to the header
                    in_test = False
                    templist.append(temp_lines)
                    temp_lines = ''
                else:
                    temp_lines += line
            else:
                temp_lines += line
        elif (len(line) > 5):
            if (line_lower[0:6] == '--test'):
                in_test = True
                tests += 1
            else:
                if (tests == 0):
                    archheader += line
                else:
                    temp_lines += line

        line = source.readline()
        line_lower = line.lower().strip()
        words = line_lower.split(' ')

    if (tests == 0):
        logwrite('c','Reached end of source file prematurely in file ' + str(path) + '.')
    footer = temp_lines

    source.close()

if (method != 'partitioned'):
    bodylines = body.split('\n')
    for line in bodylines:                                          # Lines in the body are all single tests/functions (or should be)
```

```
            if line.strip():
                tests += 1
                templist.append(line)                                   # [META] Improving detection here can be a major asset
        else:
            tests = len(templist)
        if (tests == 1):
            logwrite('n','Successfully parsed ' + entname + '.' + archname + ' with ' + str(tests) + ' testsuite found.')
        else:
            logwrite('n','Successfully parsed ' + entname + '.' + archname + ' with ' + str(tests) + ' testsuites found.')
        return [archname, entname, header, archheader, footer, templist]

    ## arranges found functions & procedures in their own executable files
    def test_format(parsedlist_t = None):
        if parsedlist_t == None:
            parsedlist_t = parsedlist
        ## Format: parsedlist is a tuple of tuples: [archname, entname, header, archheader, footer, list_of_tests]

        testcount = {}

        da = 0                                                          # Duplicate Avoider: Multiple instances of the same tb otherwise go wrong here
        for parsedfile in parsedlist_t:                                 # Assign correct variables for making the files
            if parsedfile[0] != 'ERROR':
                archname = parsedfile[0]
                entname = parsedfile[1]
                header = parsedfile[2]
                archheader = parsedfile[3]
                footer = parsedfile[4]
                tests = parsedfile[5]

                localcount = 0
                localname = entname + '.' + archname + '.' + str(da) + '.vhd'       # Name for use in dictionary and testing
                testfile_path = tempdir + os.sep + localname
                testfile = open(testfile_path,'w+')
                testfile.write('library TDD;\nuse TDD.vhdlUnit.all;\n')
                testfile.write(header)

                for test in tests:                                      # Place every test in the same file, under a different architecture
                    testfile.write(archheader.replace(archname, archname + str(localcount)))
                    #testfile.write('assert false report \"test started\" severity note;\n')
                    testfile.write(test + '\n')
                    #testfile.write('assert false report \"test ended\" severity note;\n')
                    testfile.write(footer.replace(archname, archname + str(localcount)) + '\n\n')
                    localcount += 1
                testfile.close()
                testcount[localname] = localcount                       # Keep a dictionary of the tests in each file
            else:
                logwrite('n','Ignoring test ' + str(len(testcount)) + ', file: ' + str(parsedfile[1]))
            da += 1
        return testcount

    ## grabs processed source/files, executes & captures output
    ## testcount_t is a dictionary of key: entityname.architecturename.vhd, with value the number of tests inside
    def parse_tests(testcount_t = None, tempdir_t = None, foldername_t = None, args_t = None, currentdir_t = None):
        if (testcount_t == None):                                       # Use global values if arguments not filled
            testcount_t = testcount
        if (tempdir_t == None):
            tempdir_t = tempdir
        if (foldername_t == None):
            foldername_t = foldername
        if (args_t == None):
            args_t = args
        if (currentdir_t == None):
            currentdir_t = currentdir

        args_t.prepath = get_path(args_t.prepath)
```

```
    os.chdir(tempdir_t)                                                 # Change working directory to the temporary folder
    testcount = 0


    if (args_t.prepath != None):
        source = open(args_t.prepath,'r+')
        for line in source:
            os.system(line.strip())
        #shutil.copytree(get_path(prepath), tempdir_t + os.sep + prepath.split(os.sep)[-1])
    os.system('vlib TDD')
    os.system('vcom -' + args_t.version + ' -quiet -work TDD ' + currentdir_t + os.sep + 'vhdlUnit.vhd')
    for key,value in testcount_t.iteritems():                           # Run across the dictionary, executing every test in each file
                                                                        # Make use of the vhdlUnit 'reportback' function
        commands = 'vlib work' + str(testcount) +'\n' + 'vcom -' + args_t.version + ' -quiet -work work' + str(testcount) + ' ' + key

        for line in commands.split('\n'):                               # Form a work directory per file
            os.system(line)

        words = str(key).split('.')
        entname = words[0]                                              # Get entity and architecture name from the key
        archname = words[1]

        outputname = words[0] + '.' + words[1] + '.' + words[2]
        outputfile = open(os.getcwd() + os.sep + outputname + '_cmd_output.txt','w+')
        logwrite('n','Starting execution of tests in ' + str(key))
        for test in range(0, value):
            readcmd = os.popen('vsim -c "work' + str(testcount) + '.' + entname + '(' + archname + str(test) + ')" -do "run -all;exit"').read()
            outputfile.write(readcmd)
        outputfile.close()
        shutil.copy(os.getcwd() + os.sep + outputname + '_cmd_output.txt', foldername_t)
        testcount += 1
    os.chdir(currentdir_t)

## grabs output, formats output
def format(tempdir_t = None):
    if (tempdir_t == None):
        tempdir_t = tempdir
    testname = tempdir_t.split(os.sep)[-1]                              # Folder is named for the testresult
    testresults = [testname, 0]                                         # Format of testresults: [Testresultname , Testcount, Testresult1, Testresult2, ... ]
    currentdir = os.getcwd()                                            # Testresults are formatted: [Name, Bool Passed, NumberOfTests, Result(s)]
    os.chdir(tempdir_t)                                                 # Move working directory to local dir

    for file in os.listdir(tempdir_t):                                  # Get every file that is a commandline output file
        if file.endswith('_cmd_output.txt'):
            source = open(file,'r+')
            testresult = [file.split('_cmd_output.txt')[0],0]           # Testresult is every test in the file, start with name and 0 tests
            in_test = False                                             # Opposed to testresult!S! which is a collection of testresults of files in the folder
            prev = ''                                                   # For storing the previous line
            line = source.readline()

            while(line != ''):                                          # As long as EOF is not reached
                if (line == '# run -all\n'):
                    test_name = prev.split(' ')[-1]
                    in_test = True
                while in_test:                                          # In_test is a boolean that checks whether or not we are "inside" a test
                    line = source.readline()
                    words = line.split(' ')
                    if (words[1] == 'Errors:' and words[3] == 'Warnings:' ):    # Standard end for modelsim, use as 'endpoint' of tests
                        in_test = False
                    else:
                        if (line == '# ** Note: test started\n'):               # Currently no good detection on whether test passed or not!
                            test = [test_name, True, 0, '']                     # Test format is [Name, Test passed, NumberOfTests, testresult1, testresult2, ...]
                            line = source.readline()
                            line = source.readline()                            # Skip 1 line with the time status
                            no_test = True
```

```
                            while (line != '# ** Note: test ended\n'):
                                no_test = False
                                if (line == ''):
                                    logwrite('c', 'Encountered faulty results in ' + 'Reached EOF mid-test.')
                                    in_test = False
                                    break
                                if (len(line) > 12):                                # Otherwise "line[5:10]" might cause array size errors
                                    if (line[5:10] != 'Time:'):
                                        test[2] += 1                                # Increase number of tests found
                                        test[3] += '\n' + line                      # Append line for info
                                else:
                                    test[2] += 1
                                    test[3] += '\n' + line
                                line = source.readline()
                            if(not no_test):
                                test[3] = test[3][1:]                               # Remove unnecessary newline at start of log
                            else:
                                test[2:] = [1, 'One test ran, no output detected.']
                            testresult.append(test)
                            testresult[1] += 1
                        # This is the end of "while in_test:"
                    prev = line
                    line = source.readline()
                    # This is the end of "while(line != ''):
                testresults.append(testresult)
        os.chdir(currentdir)
        return testresults

    ## grabs processed output, converts to JUnit compatible XML file
    # Available format is Package (collection of testsuites with an own name)
    # containing testsuites with each their own name and testcount
    # Who each have testcases with their own name and error/success report
    # Format of testresults: [Name , Count, Testresult1, Testresult2, ... ]
    # Testresult format is: [Name, Count, Test1, Test2, ...]
    # Test format is [Name, Test passed, NumberOfSubTests, testresult1, testresult2, ...]
    def xmlwrite(testresults, foldername_t=None):
        if (foldername_t == None):
            foldername_t = foldername

        hostname = str(testresults[0])                                              # Hostname is the RANDOMNAME
        xmltargetpath = foldername + os.sep + hostname + '_testresults.xml'         # Save in file named RANDOMNAME_testresults.xml
        xmltargetfile = open(xmltargetpath, 'w+')

        ts = []                                                                     # Every file is a testsuite

        # JUnit-xml format is: TestCase(Name, ClassName, Elapsed_Time_In_Sec, Stdout, Stderr)
        #                      TestSuite(Name, Test_cases[], Hostname, ID, Package, Timestamp, Properties)
        for testresult in testresults[2:]:
            test_cases = []                                                         # Every testsuite can contain multiple testcases
            for test in testresult[2:]:
                testcase = TestCase(test[0], testresult[0], None)
                if(not test[1]):
                    testcase.add_failure_info(None, test[3:])
                test_cases.append(testcase)
            ts.append(TestSuite(testresult[0], test_cases, testresults[0]))
        xmltargetfile.write(TestSuite.to_xml_string(ts))
        xmltargetfile.close()

    def format2(tempdir_t=None, foldername_t=None):
        if (tempdir_t == None):
            tempdir_t = tempdir
        if (foldername_t == None):
            foldername_t = foldername
        testname = tempdir_t.split(os.sep)[-1]
        currentdir = os.getcwd()
        os.chdir(tempdir_t)
```

```
failedtests, passedtests, othernotes, totaltests = 0, 0, 0, 0
failedlines, passedlines, otherlines, everyline = [], [], [], []

for file in os.listdir(tempdir_t):                              # Get every file that is a commandline output file
    if file.endswith('_cmd_output.txt'):
        failedtests_l, passedtests_l, othernotes_l, totaltests_l = 0, 0, 0, 0
        failedlines_l, passedlines_l, otherlines_l, everyline_l = '', '', '', ''
        source = open(file,'r+')

        ReadNote = False
        lastline = ''
        for line in source:
            words = line.split('␣')
            if ReadNote == True:
                if (len(words) > 5):
                    to_add = '␣-␣time:␣' + words[5] + '␣' + words[6] + '\n'
                else:
                    to_add = line
                if lastline == 'success':
                    passedlines_l += to_add
                elif lastline == 'failed':
                    failedlines_l += to_add
                elif lastline == 'other':
                    otherlines_l += to_add
                everyline_l += to_add
                ReadNote = False

            elif (len(words) > 2):
                if(words[2] == 'Note:'):
                    ReadNote = True
                    if(len(words) > 4):
                        if(words[4] == 'failed\tname:'):
                            failedtests_l += 1
                            totaltests_l += 1
                            failedlines_l += str(failedtests_l).zfill(4) + '␣-␣' + line[11:-1]
                            lastline = 'failed'
                        elif(words[4] == 'success\tname:'):
                            passedtests_l += 1
                            totaltests_l += 1
                            passedlines_l += str(passedtests_l).zfill(4) + '␣-␣' + line[11:-1].split('-')[0]
                            lastline = 'success'
                        else:
                            othernotes_l += 1
                            totaltests_l += 1
                            otherlines_l += str(othernotes_l).zfill(4) + '␣-␣' + line[11:-1]
                            lastline = 'other'
                    else:
                        othernotes_l += 1
                        totaltests_l += 1
                        otherlines_l += str(othernotes_l).zfill(4) + '␣-␣' + line[11:-1]
                        lastline = 'other'
                    everyline_l += str(totaltests_l).zfill(4) + '␣-␣' + line[11:-1]

        if ReadNote == True:
            to_add = '␣-␣time:␣' + words[5] + '␣' + words[6]
            if lastline == 'succes':
                passedlines_l += to_add[1:-1]
            elif lastline == 'failed':
                failedlines_l += to_add
            elif lastline == 'other':
                otherlines_l += to_add
            everyline_l += to_add

        failedtests += failedtests_l
        passedtests += passedtests_l
```

```
            othernotes += othernotes_l
            totaltests += totaltests_l
            failedlines.append(failedlines_l)
            passedlines.append(passedlines_l)
            otherlines.append(otherlines_l)
            everyline.append(everyline_l)

            source.close()

                #Write plain .txt file with testresults
        targetpath = foldername_t + os.sep + testname + '_testresults.txt'
        targetfile = open(targetpath,'w+')
        targetfile.write('total tests: '      + str(totaltests)
                         + '\ntests passed: ' + str(passedtests)
                         +'\ntests failed: '  + str(failedtests)
                         + '\nother notes: '  + str(othernotes))
        output_failed, output_passed, output_other, output_every = '', '', '', ''
        for x in range (0, len(failedlines)):
            output_failed += failedlines[x]
            output_passed += passedlines[x]
            output_other += otherlines[x]
            output_every += everyline[x]
        targetfile.write('\n\n\nPassed tests reports:\n' + output_passed
                         + '\n\nFailed tests reports:\n' + output_failed
                         + '\n\nOther notes:\n'          + output_other
                         + '\n\n\nAll test results:\n'   + output_every)

        print( 'total tests: '      + str(totaltests)
               + '\ntests passed: ' + str(passedtests)
               +'\ntests failed: '  + str(failedtests)
               + '\nother notes: '  + str(othernotes))
        #Print left out - optional command line output

        targetfile.close()
        os.chdir(currentdir)
        return everyline


    #Write .xml file in JUnit format (For use in Eclipe)
    #Uses JUnit-xml package by Brian Beyer
    #source at https://github.com/kyrus/python-junit-xml
    #Requires setuptools to install at https://pypi.python.org/pypi/setuptools
    def xmlwrite2(everyline, foldername_t=None):
        if (foldername_t == None):
            foldername_t = foldername

        testname = foldername_t.split(os.sep)[-1]

        xmltargetpath = foldername_t + os.sep + foldername_t.split(os.sep)[-1] + '_testresults.xml'
        xmltargetfile = open(xmltargetpath, 'w+')

        test_suites = []
        suite_count = 1
        for lines in everyline:
            test_cases = []
            for line in lines.split('\n'):
                words = line.split(' ')
                if line.find('success') != -1:
                    time_taken = get_time(line.split('-')[-1][7:])
                    name = words[0] + ' - ' + line.split('-')[1].split('name:')[1].strip()
                    test_cases.append(TestCase(name, testname, time_taken, None))
                elif line.find('failed') != -1:
                    time_taken = get_time(line.split('-')[-1][7:])
                    name = words[0] + ' - ' + line.split('-')[1].split('name:')[1].strip()
                    message = ("-".join(line.split(' - ')[2:-1])).strip()
                    tc = TestCase(name, testname, time_taken, None)
```

```
                tc.add_failure_info(None, message)
                test_cases.append(tc)
        test_suites.append(TestSuite('TestSuite number: ' + str(suite_count), test_cases))
        suite_count += 1

    xmltargetfile.write(TestSuite.to_xml_string(test_suites))
    xmltargetfile.close()

# extracts the passed time from a modelsim time notice
def get_time(timestring):
    words = timestring.split(' ')
    new_time = float(words[0])
    multiplier = 1
    if words[1] == 'ps':
        multiplier = 10**(-12)
    elif words[1] == 'ns':
        multiplier = 10**(-9)
    elif words[1] == 'us':
        multiplier = 10**(-6)
    elif words[1] == 'ms':
        multiplier = 10**(-3)
    return new_time * float(multiplier)

## removes temporary files & directories
def cleanup():
    logwrite('n','Cleanup started at ' + str(time.time()))
    shutil.rmtree(tempdir)


##########################################
######## MAIN PROGRAM STARTS HERE ########
##########################################

#Allows the code to be used as a module
if __name__ == "__main__":

    parsedlist, logs_buffer = [], []                                    # A buffer is needed for loglines from before the logdirectory was created
    logstarted = False
    tempdir, tempname, foldername = '', '', ''                          # Future variables that will be used througout the script
    parser = None                                                       # Assigning name for the parser

    systemtime = time.time()                                            # Marks the starting time of the script
    currentdir = os.getcwd()                                            # CWD is changed during simulation

    parser = setup_parser()                                             # Creates an argument parser for the commandline arguments
    args, unknown = parser.parse_known_args()                           # Parses commandline arguments, stores unknown arguments in 'unknown'

    tempdir, tempname = make_tempdir()                                  # Creates a unique (name for the) temporary directory in the systems temp dir
    foldername = setup(args, tempname)                                  # Grab all functions and procedures to be processed, returns output folder

    logwrite('n','Started script at ' + str(systemtime))
    if (unknown):                                                       # If there are unknown arguments, write error to log
        logwrite('w', 'Found unusuable arguments: ' + ',' .join(unknown))

    if (args.file):                                                     # If the files to be processed are in a file
        for file in args.list:                                          # Get all filenames
            listfile = open(get_path(file))                             # Open the files one by one
            sourcelist = [line.strip() for line in open(listfile)]      # Get all .vhd files listed within
            for line in sourcelist:                                     # Parse each .vhd file
                parsedlist.append(parse_source(get_path(line), args.method))
    else:                                                               # If arguments are not lists, they are the files themselves
        for file in args.list:
            parsedlist.append(parse_source(get_path(file), args.method))

    testcount = test_format(parsedlist)                                 # Get list of number of tests
    if args.cmdline:                                                    # Location of vhdlUnit.vhd may vary on how the script is called
```

```
    parse_tests(testcount, tempdir, foldername, args)
else:
    parse_tests(testcount, tempdir, foldername, args, os.path.dirname(os.path.realpath(__file__)))

testresults = format2(tempdir)                                  # Format the testresults to humanly readable words
xmlwrite2(testresults)                                          # Format the above format into a JUnit-XML format

cleanup()                                                       # Remove the temporary working directory
logwrite('n','Stopped script at ' + str(time.time()))           # Note the time of ending
```