

# Automated testbench generation from digital timing diagrams

Winand Seldeslachts

Supervisors: Prof. Luc Colman, Ir. Hendrik Eeckhaut (Sigasi nv)

Counsellor: dhr. Lieven Lemiengre (Sigasi nv)

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

Department of Information Technology  
Chair: Prof. dr. ir. Daniël De Zutter  
Faculty of Engineering and Architecture  
Academic year 2015-2016





# Automated testbench generation from digital timing diagrams

University of Ghent



Seldeslachts Winand

May 20, 2016

# Abstract

# Acknowledgements

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Problem analysis</b>	<b>9</b>
<b>3</b>	<b>Functional analysis</b>	<b>12</b>
<b>4</b>	<b>Technical analysis</b>	<b>17</b>
4.1	Testbench creation . . . . .	18
4.1.1	Testbench specifications . . . . .	18
4.1.2	Conversion script . . . . .	20
4.1.3	Source file . . . . .	20
4.2	Wave trace analysis . . . . .	21
<b>5</b>	<b>Technical design</b>	<b>22</b>
5.1	Testbench creation . . . . .	22
5.1.1	Simple example . . . . .	22
<b>6</b>	<b>Conclusion</b>	<b>27</b>
<b>A</b>	<b>example</b>	<b>28</b>

# Chapter 1

## Introduction

In software and hardware development, verification and validation is the process of checking that a system meets specifications and that it fulfills its intended purpose. It may also be referred to as quality control. It is normally the responsibility of testers as part of the development lifecycle [1].

In other words, the questions

- Are we building the product right? (Verification)
- Are we building the right product? (Validation)

are becoming more and more important to designers as consumers become increasingly demanding.

The software development industry has invested enormously in new verification methods and is still building toward better verification. Software developers have an abundance of support tools available for their respective languages (e.g. C, Java, Python, etc.). These tools help the programmer by automating a lot of the actions required to write, validate, and document the code.

Hardware development has always lagged behind however. Support tools for hardware development languages (HDLs) like VHDL and Verilog are limited and often outdated. Modernising verification methods for HDLs is progressing slow than for programming languages. There is still a lot of room for improvement in this area.

In light of this, this report introduces a way to improve VHDL design verification. More specifically, provide an easy way to easily design tests and simulate them, while at the same time generating documentation. The simulation result is then automatically compared to the documentation. The operation of the resulting software is illustrated below.

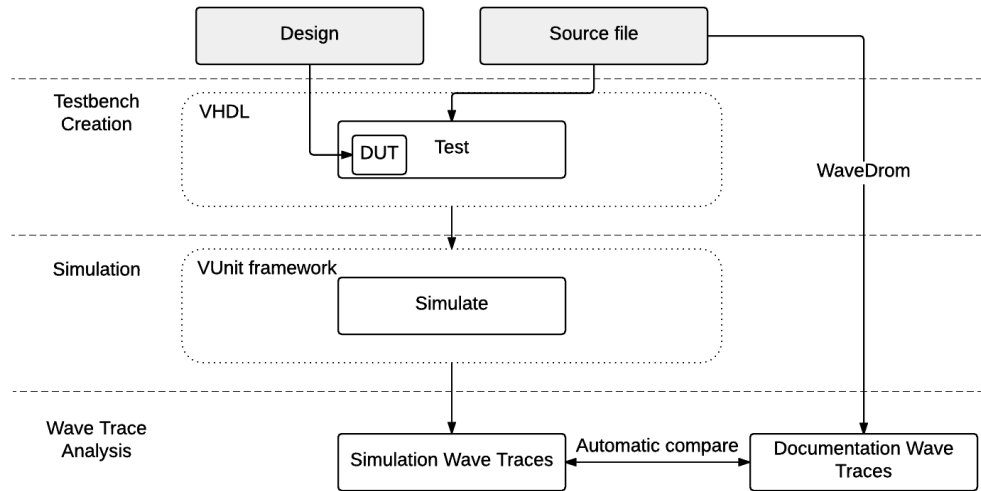


Figure 1.1: Operation overview of the new system

To understand the purpose of the proposed tool, a short introduction in the verification process is necessary.

Whenever a hardware component is implemented in a hardware description language, its behavior is defined the design file. The behavior of the component is often documented visually by showing input and expected output wave traces. Wave traces are the visualisation of a signals values over time. An example for an AND-gate is shown below. Input signals are connected to a designs input and the output signals represent the output of the device.

The behavior is tested by attaching it to a so called testbench. Historically, it is also written in a HDL and describes the environment in which the design is to be tested. In this testbench one or more tests can be performed



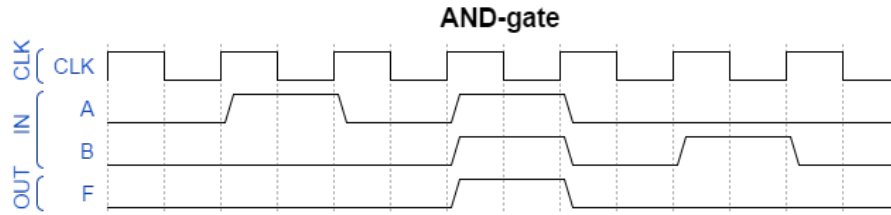


Figure 1.2: AND-gate wave trace example

by attaching certain input signals to the designs input ports. Creating this testbench is the first step in the verification process. The design being tested is often referred to as the Device Under Test (DUT) or the Unit Under Test (UUT). Figure 1.3 illustrates the hierarchy of a testbench.

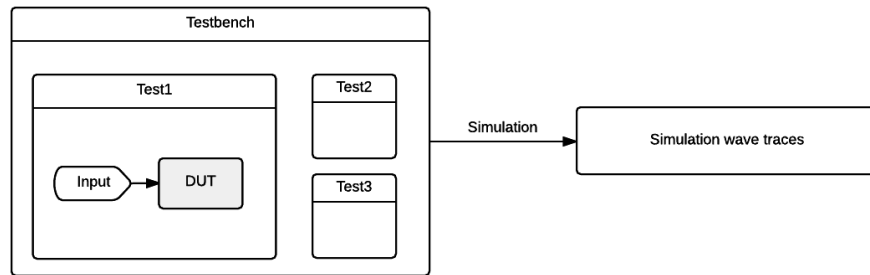


Figure 1.3: Hierarchy of a testbench

The second step of the verification process is to simulate the tests described in a testbench. A simulation yields output signals depending on the input signals and the behavior of the DUT. Figure 1.4 shows an example of a simulation output (simulation wave traces).

## Maak Mij

Figure 1.4: Modelsim simulated wave traces

To verify a design, ideally, every possible input has to be simulated in the testbench and have its simulated output compared to the expected output specified in the documentation. This would mean that the tests cover 100% of the design code. In practice trying to reach this percentage is not economically desirable, but designers still strive to reach an acceptably high coverage percentage. Checking whether test results are in accordance with the documentation is the last step in verification cycle.

The complete verification cycle is shown in figure 1.5.

Each step can be optimized in their own way. The proposed tool will focus on streamlining the first and the last step, relying on existing simulation tools to handle the second step. It will use a new source file to help generate documentation wave traces, while at the same time using that file to create testbenches and analyse simulation wave traces. The improved verification cycle is shown in figure 1.6.

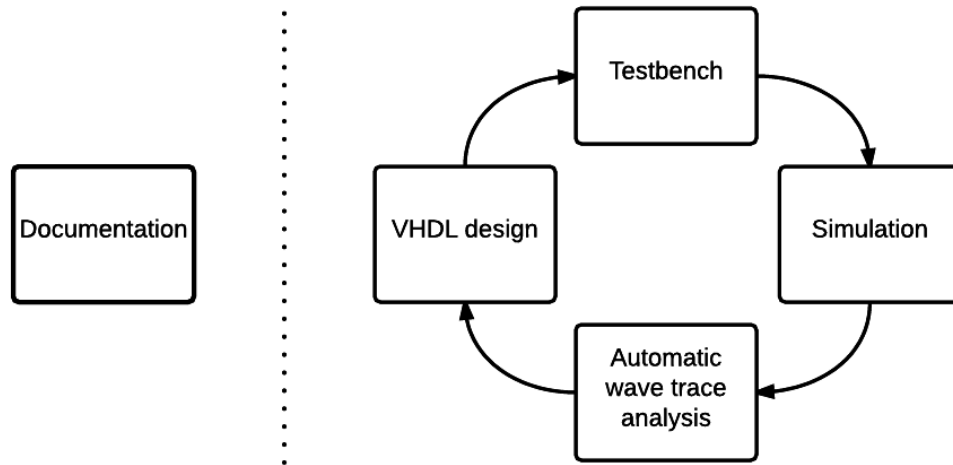


Figure 1.5: Traditional verification cycle

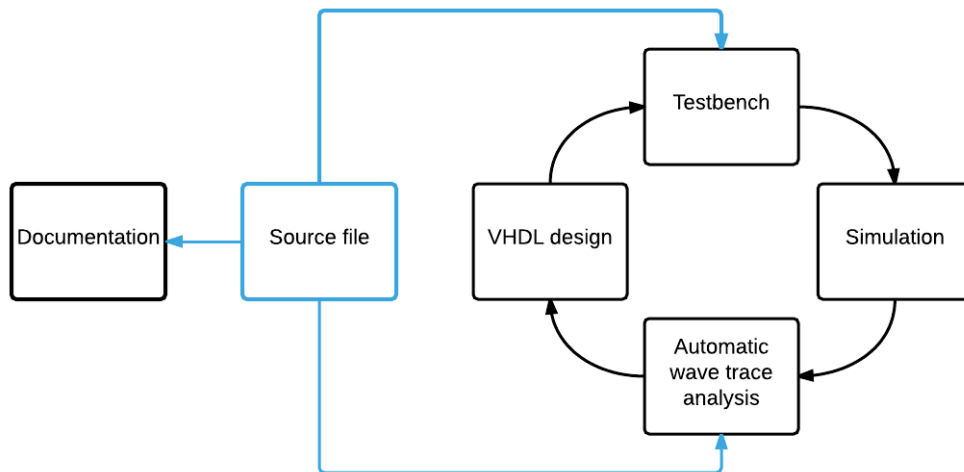


Figure 1.6: Improved verification cycle

# Chapter 2

## Problem analysis

The classic (V)HDL verification process is shown in the figure 1.5. Starting from a VHDL design a testbench is created. The testbench is then simulated, which yields the simulation wave traces. These wave traces are then visually checked for compliance with the documentation wave traces. If the design behaves as the documentation specifies, the verification process is complete. If the design does not behave accordingly, the design is reviewed and the process is run again. The behavior of the design is described in the documentation.

The least efficient steps in this process are the creation of the test benches (top) and the visual comparison of wave traces (bottom). They will be the primary elements of focus in this report.

The first element of focus is the creation of testbenches. Designers either have to rely on older, outdated, testbenches who might not be self checking or design a new testbench manually. Self checking testbenches are testbenches that report any irregular behavior automatically, whereas conventional testbenches simply simulate output signals and leave the analysis to the designer. Testbenches are often at least partly written manually and creating them is a repetitive task. This means writing a test for every aspect is time consuming and prone to human error. If this process could be automated, a lot of time and work could be saved. At the same time writing the documentation is a tedious task, where every function has to be explained. Documentation often includes example wave traces. As can be seen from image 1.2 and 1.4, which are documentation wave traces and simulated wave traces a certain feature respectively, these wave traces are exactly the same (considering the

design functions as documented). Although the way they were created differs greatly, the documentation and test result essentially hold the same information. In theory it would be possible to devise a system that could build both the documentation wave traces and the test for a specific feature from the same information, eliminating the need to create testbenches manually, while still offering the same functionality.

In short, the system should be able to build a self checking testbench automatically based on the information in the documentation.

The second element of focus is comparing the expected output in the documentation to the simulated output of the testbenches, or simply wave trace analysis. Self checking testbenches perform an analysis during simulation and show errors to the user by logging them to a file or terminal, but testbenches that are not self checking leave all wave trace analysis to the user. Comparing wave traces visually is not ideal, because the complexity of some tests and their corresponding wave traces is considerable, which often results in a staring contest between designer and simulation result. Because of this, the proposed solution will not only build self checking testbenches, but will process the analysis result and show it in a user friendly and easy to understand format. Building on the presumption that there is a way to build both documentation wave traces and testbenches from the same information, we can say that the documentation wave traces should be exactly the same as the simulated wave traces. Because of this wave trace analysis boils down to checking these two wave trace files for discrepancies and showing the difference in a clear way to the user.

In short, the system should be able to take the error reports logged by the self checking testbench and show them in an easily understandable way to a user.

The resulting software package should be easy to use, that is why a simple gui will be added where the user can input the source file and the VHDL design to be tested. From this information the testbench can be created and simulated by pressing a button. Finally the simulated wave traces are automatically compared to the documentations and the result is shown.

Lastly a short analysis will be made regarding the use of this software and

the benefits it provides over existing frameworks.

To summarise this report will try to answer the following questions:

- Can we design a system that will streamline the process of design validation?

Can we design a system that can create self checking testbenches and documentation wave traces from the same source?

Can we optimise wave trace analysis based on this system?

- Does this system provide any benefits over existing verification frameworks?

# Chapter 3

## Functional analysis

The goal of this project is to create a tool that can streamline the testbench creation and the wave trace analysis processes to help developers validate their design, while at the same time generating wave trace images to aid in documenting a design. The original validation process flow can be seen in §[REF PA]. To improve on this process the information provided in a source file is used. Figure 1.6 shows the improved information flow of the validation process. Where the original system required (more) user input at both the documentation side and the validation side, this system bundles most user input into one file, reducing the total amount of user input and saving a considerable amount of time. An overview of how it works is shown in the figure below.

The tool will complete a full cycle of the validation process. It starts from the user input files: the source file and the design file. And then continues in three steps:

First, the tool can create testbenches automatically and generate documentation information using a source file. This source file is a waveJSON file based on standard WaveDrom [2] files. It can create a testbench for every source file provided in mere seconds, faster than a developer could create just one testbench manually. These testbenches are designed according to the specifications of a VUnit [3] testbench.

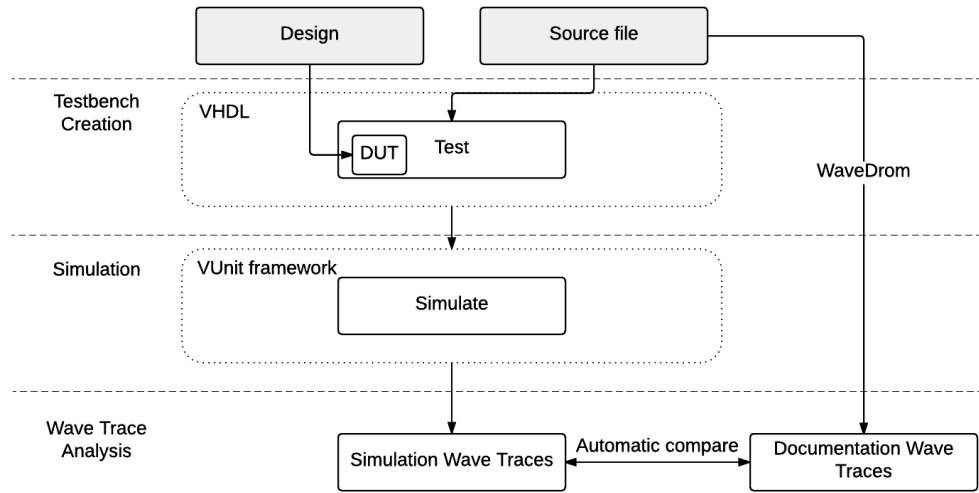


Figure 3.1: Operation overview of the new system

Then, the framework provided by VUnit is used to run the created testbenches through an external simulator. This can be any simulator supported by the VUnit framework.

Finally, instead of the manual comparison of two wave traces, it offers a way to quickly find non corresponding wave traces in a visual way. It returns a waveJSON file similar to the source file, which can be visualised using WaveDrom. The WaveDrom generated figure 3.2 illustrates this concept. This visual representation will be accompanied by an error message specifying the error as seen in log message 1.



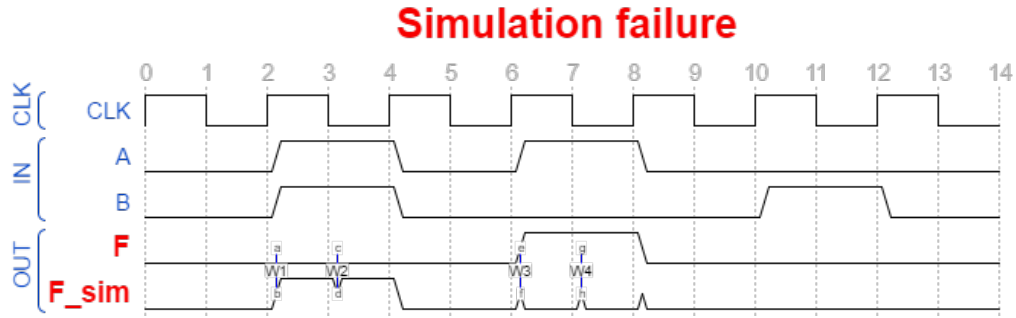


Figure 3.2: Simulation failure example

W1: Expected sig\_F = '0', got sig\_F = '1' at n = 2.

W2: Expected sig\_F = '0', got sig\_F = '1' at n = 3.

W3: Expected sig\_F = '1', got sig\_F = '0' at n = 6.

W4: Expected sig\_F = '1', got sig\_F = '0' at n = 7.

Log message 1: Example logged error messages

The two features are bundled together in one tool controlled by a simple GUI.

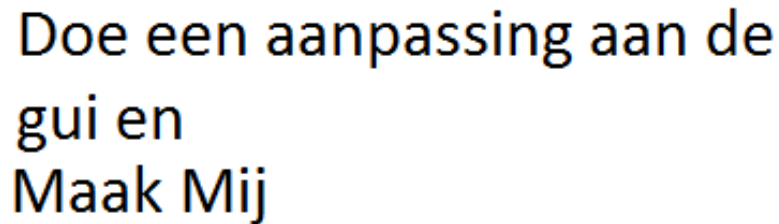
The image shows a window with a title bar. The text inside the window is arranged in four lines: 'Doe een aanpassing aan de', 'gui en', 'Maak Mij'. The text is in a dark blue, sans-serif font. The window appears to be a standard graphical user interface element.

Figure 3.3: Look of the user interface

The gui allows the user to open the input folder where WaveDrom input files should be added, the src folder where the design to be tested should be added and the output folder, where the resulting files will be stored. The run verification button creates a testbench for every json file in the source folder, simulates the result, verifies it and stores the result in the output folder.

To summarise the tool will convert

- Source files for every feature to test
- The design under test VHDL description

into

- A VHDL test for every source file
- Wave trace analysis files for every test

There are many benefits to this system. The user can be sure the test that has been generated does not hold any mistakes (at least if we assume no mistake has been made in creating the source file) as it is directly derived from the specifications of the device. Also, validating a device becomes easier way of compared to what it used to be - (create testbench,) simulate, compare

simulation wave traces manually. One click offers an immediate validation. The test will either pass or fail. In the latter case extra information is immediately available.

It is however not a full validation system on itself. It requires the use of existing frameworks to function. It offers an easy to use and approachable way to use these frameworks while at the same time adding functionality to them. In this case the VUnit framework was chosen, but it could also be adapted to run on other frameworks such as CoCoTB [4]. It is a useful extension of these frameworks for all those who want to validate a design and document its functionality.

# Chapter 4

## Technical analysis

This chapter will analyse each part of the system separately based on the system overview in figure 4.1.

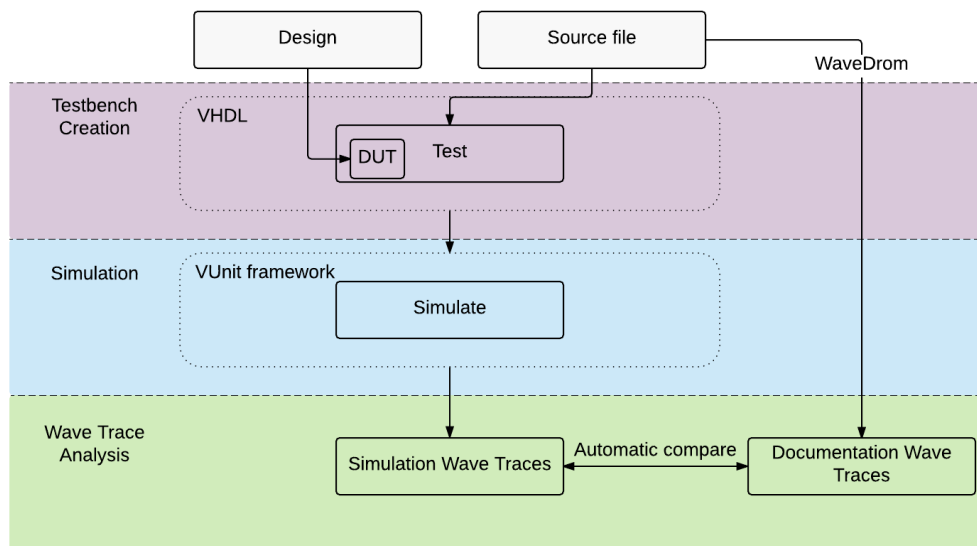


Figure 4.1: Color coded overview of the program operation

## 4.1 Testbench creation

Creating testbenches requires several steps. First the specifications for the testbench have to be determined, then the way they will be created has to be determined and finally the source material (i.e. what the testbench will be created from) has to be chosen. This part corresponds to the purple part.

### 4.1.1 Testbench specifications

Before the way testbenches should be constructed can be defined, the specifications for the testbenches have to be determined. This depends on which framework is used in the simulation step (blue part). Several testbench frameworks are available that improve on the standard VHDL testbenches. They offer features such as unit testing, (conditional) logging packages and many more. Some even offer the possibility to write testbenches in different programming languages.

Some of the most known frameworks, their supported languages and their most important features are listed below. These frameworks are all open source.

**VUnit** : VHDL, (System)Verilog

- Unit testing
- Python test runner
- Advanced logging libraries
- Multi simulator support

**CoCoTB** : VHDL, (System)Verilog

- Python testbenches
- Python test runner
- Multi simulator support

**SVUnit** [5]: (System)Verilog

- Unit testing

**OSVVM** [6]: VHDL

- Maximum test coverage support
- Randomised testing

**UVVM** [7]: VHDL

- AANVULLEN

The frameworks with the most advanced features and support for both (System)Verilog and VHDL are VUnit and CoCoTB. The framework suggested by the promoters of this project is VUnit. It will be the framework supporting this system.

The VUnit framework aims to bring the best software testing practises to HDL languages. A big part of that is supporting unit testing, hence the name V(HDL)Unit. The VUnit documentation can be found in the vunit documentation [8]. VUnit testbenches do not differ a lot from regular testbenches. The only difference is that all test have to be written inside the same process as seen in the example below.

```
1 main : process
  begin
3 test_runner_setup(runner, runner_cfg);
  while test_suite loop
5   if run("test_pass") then
      report "This will pass";
7   elsif run("test_fail") then
      assert false report "It fails";
9   end if;
  end loop;
```

Code 4.1: Main test loop for a VUnit testbench

Multiple tests can be implemented using the if-elsif mechanism. All tests are run by the python test runner, which can be configured according to the user's wishes. This means that a test can be started by a python command. These test can be run separately thanks to the VUnit framework. The framework offers the possibility to set the simulator backend that will simulate all the tests. How to set up the VUnit framework and run test can be found in the VUnit documentation.

### 4.1.2 Conversion script

With the form of the testbenches defined, a general construction method can be implemented. This is done using the popular programming language Python, because of its clarity, flexibility and ease of use. On top of this it is the language used to build the VUnit framework, which makes it easier to integrate VUnit into the software.

The conversion script will read the wave trace file, convert it to a VUnit compatible testbench and run it using the VUnit framework.

A lot of relevant information could be extracted from the device description, such as the name, type and direction of all ports. For now however the conversion script expects all information to be available in the wave trace files. An optimisation can be made at a later time.

### 4.1.3 Source file

The source file is the source for creating the testbench, it contains all the information necessary to create a testbench, while still being understandable. Information should be easily extractable and processable. These constraints are all met by WaveDrom Files.

WaveDrom is an open source timing diagram rendering engine that converts waveJSON input files into the corresponding timing diagram. These waveJSON files are considered the source file, but will also be referred to as WaveDrom files in this report.

WaveJSON [?] is an application of the JSON format. The purpose of it

is to provide a compact exchange format for digital timing diagrams.

Standard WaveDrom files do not hold enough information to create a test-bench. However, it is possible to add new JSON fields to these WaveDrom files without compromising the generated wave traces. This way any information needed can be integrated in these files.

## 4.2 Wave trace analysis

This part corresponds to the green part of image 4.1. After simulation a test will either pass or fail. This functionality is integrated in VUnit. During simulation the output is compared to the expected output and whenever they are not equal, the test will automatically fail and an error message will be generated. This error message is logged to a CSV file using the conditional logging functionality of the VUnit check library.

The logged errors can then be processed by a script to create files in which the errors are clearly marked. This script will be written in Python and will create WaveDrom files based on the original input WaveDrom file.



# Chapter 5

## Technical design

### 5.1 Testbench creation

This chapter holds a step by description of the development of the software. Every new element is introduced at the hand of an increasingly complicated example input file.

#### 5.1.1 Simple example

##### The WaveDrom file

Although the primary goal of this product is to create testbenches for timed designs, a basic first draft of the program conversion script can be created based on a simple combinatorial design.

The example this draft was based on was an AND gate. The AND-gate design can be found in ??.

A WaveDrom file for describing the wave traces of a purely combinatorial AND gate is shown in code 5.1.

---

```
1 { "signal" : [  
2   { "name" : "A", "wave" : "0" },  
3   { "name" : "B", "wave" : "0" },  
4   { "name" : "F", "wave" : "0" }]  
5 }
```

---

Code 5.1: Standard WaveDrom description for a combinatorial AND-gate

Where A and B are considered input signals and F is the output signal. This yields the following wave trace image which shows that F should be low when both inputs are also low.

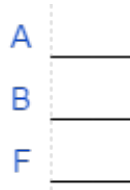


Figure 5.1: Wave traces corresponding to description in code 5.1

This is neither very clear, nor does it hold enough information on its own to create a testbench. To be able to create a testbench several more fields containing extra information should be added.

The necessities for creating a testbench are:

- The exact name of the unit under test
- The vhdl signal type for every signal
- The direction of the signal (input or output)

These fields have been added to the WaveDrom file together with a name and description for this test. The new WaveDrom file is shown in code 5.2:

```
1 { "name": "andGate", "test" : "andgate00",
2   "description": "Input A = '0' & B = '0' should produce a
   low output",
3   "signal": [
4     ["IN",
5       { "name": "A", "wave": "0", "type": "std_logic"},
6       { "name": "B", "wave": "0", "type": "std_logic"}],
7     ["OUT",
8       { "name": "F", "wave": "0", "type": "std_logic"}]
9   ] }
```

---

Code 5.2: Extended WaveDrom description for a combinatorial AND-gate

Where the name should be the exact name of the unit under test and the port names have to be exactly the same as in the uut design. Each port should be defined under their corresponding label of direction. All input files must be defined under the “IN” label and all output signals under the “OUT” label. Now all information needed to create a testbench is available.

### Creating VHDL code from the WaveDrom file

The information in this file has to be translated to VHDL code. More specifically the information in the WaveDrom file should be converted into:

- Signal declarations for every signal
- DUT declaration
- DUT port map
- Input signal stimulus
- Output signal checking

These are the elementary parts of a VHDL testbench. The python JSON package enables an easy conversion of information by reading all of the the waveJSON information and placing it in a python JSON container. This way data in the python code has the same structure as the data in the WaveDrom file. Because all the information is now available in the python program all

that has to be done is manipulate standard strings to fit the current design.

For example the string manipulation code for signal declarations is given below.

```
1 def generate_signal_declaration(json_signal):
    signal_name = json_signal["name"]
3    signal_type = json_signal["type"]

5    #example: "signal sig_example : std_logic := '0';"
    return "signal sig_" + signal_name + " : " + \
7        signal_type + " := 0;"
```

Code 5.3: Generating signal declarations in Python

This method reads the information regarding a JSON signal and produces a VHDL signal declaration string which initialises the signal to '0'.

In the same way signal stimuli and the UUT port map can be generated.

```
1 def create_stimulus(input_signals):
    stimulus = ""
3    for signal in input_signals:
        if len(signal) > 0:
5        value = signal["wave"]

7        #example: "sig_example <= '1';"
        stimulus += "sig_" + signal["name"] + " <= '" + value
        + "';\n"
9    return stimulus
```

Code 5.4: Generating signal stimuli in Python

This method needs a list of all input signals in JSON format. It creates a VHDL assignment assigning them the value contained in the “wave” field and returns that string. The result can be directly added to the testbench.

```
1 def generate_port_map(json_signal_set):
    port_map = "PORT MAP(\n"
3    first = True
    for signal_type in json_signal_set:          # IN and OUT.
5        if len(signal_type) > 0:                # JSON allows empty
            elements
```

```
    for signal in signal_type:
7        if len(signal) > 0:           # JSON allows empty
            elements
            if first:
10                #example: "example => sig_example"
                new_map = signal["name"] + " => sig_" + signal[
                    "name"]
11                first = False
            else:
13                new_map = ",\n" + signal["name"] + " => sig_" +
                    signal["name"]
                    port_map += new_map
15 port_map += " );"
return port_map
```

Code 5.5: Generating a port map in Python

## Chapter 6

## Conclusion

# Appendix A

## example

# Bibliography

- [1] “Software verification and validation.” [Online]. Available: [en.wikipedia.org/wiki/Software\\_verification\\_and\\_validation](http://en.wikipedia.org/wiki/Software_verification_and_validation)
- [2] “Wavedrom homepage.” [Online]. Available: [wavedrom.com](http://wavedrom.com)
- [3] “Vunit home page.” [Online]. Available: [vunit.github.io](http://vunit.github.io)
- [4] “Cocotb documentation.” [Online]. Available: [cocotb.readthedocs.io/](http://cocotb.readthedocs.io/)
- [5] “Svunit home page.” [Online]. Available: <http://www.agilesoc.com/open-source-projects/svunit/>
- [6] “Osvvm home page.” [Online]. Available: <http://osvvm.org/>
- [7] “Uvvm home page.” [Online]. Available: <http://bitvis.no/products/uvvm-utility-library/>
- [8] “Vunit home page.” [Online]. Available: [vunit.github.io/documentation](http://vunit.github.io/documentation)