Faculty of Engineering and Architecture

Building a better VHDL testing environment

Joren Guillaume

Supervisors: Prof. Ir. L. Colman, Dr. Ir. H. Eeckhaut
Counsellor: Ir. Ing. L. Lemiengre

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Electronics and ICT Engineering Technology

Academic year 2014–2015

Usage restrictions

Preface

In the past year and a half I had the pleasure to work on the concept of integrating the software and hardware development worlds. In this thesis is contained the culminated knowledge and experience that was obtained during these many months, lest it be forgotten.

In the first part the tackled problem is explained along with a summarised background on many of the practices and concepts one needs know to fully understand the next part. In the second part the practical side of the work is demonstrated and thoughts are shared on the usability and future of this line of work.

Acknowledgements

In the course of learning, producing and revising both the code and the written text many helping hands guided my way. With this I would like to extend a thanks to them.

To my fellow students for the pleasant atmosphere that dominated most of my studies. For providing help when asked and receiving advice without reluctance. And for those many hours spent in the telecommunications lab working side by side on many things.

To my supervisors prof. ir. Luc Colman and dr. ir. Hendrik Eeckhaut. The former for providing his many years of experience in supporting students, keeping a calm but firm grip on matters and always pushing to the next step. The latter for granting me the subject, providing his technical expertise and willingness to work on my problems whenever I asked.

Most certainly to my counsellor ir. ing. Lieven Lemiengre for providing an endless amount of feedback and brainstorming whenever. For providing technical expertise and insight when I had none to be found, for providing ideas and hooks to keep me going. For remaining realistic and setting achievable goals when the light at the end of the tunnel was darkest. And of course, for reviewing and critiquing this thesis.

To Jeroen Baeken and ing. Sion Verschraege for being the friends that they are. For providing feedback on anything and support whenever it was needed.

Finally, to my darling fiancée Dorine Ndagsi for supporting me the many years we have been together. For providing a critical eye when needed and bringing me down a notch when explanations were overly complicated. And most certainly for being the driving force behind most of the work that got done at home, at work and in life.

To all those that are not named, you are not forgotten, the list would simply be endless.

Abstract

Contents

List of Figures

List of Acronyms

CC	Code Coverage	5
CI	Continuous Integration	8, 15, 17
CRV	Constrained Random Verification	5
DFF	D Flip-Flop	2, 12, 15
DT	Directed Testing	5
DUT	Device Under Test	3, 4
FC	Functional Coverage	5
FSM	Finite State Machine	5
HDL	Hardware Description Language	2, 4–6
IEEE	Institute of Electrical and Electronics Engineers	2
RC	Revision Control	8, 15
RTL	Register Transfer Level	2
TDD	Test Driven Development	7
TFD	Test First Development	7
UUT	Unit Under Test	3, 5
VHDL	VHSIC Hardware Description Language	1, 2, 5, 6, 10, 16, 18, 19
XML	eXtensible Markup Language	9, 14, 15

Part I
Problem and background

1 Problem

Revisie nodig: belangrijkste pagina
Developing digital hardware with the VHSIC Hardware Description Language (VHDL) is, like any code, prone to errors, either by the developer or by wrong product specifications. To ensure errors are weeded out before the more expensive roll-out or production begins, the code must be subjected to rigorous testing. *Currently, large and impractical tests are needed to fully test a product.*
productie != einddoel alle VHDL -> FPGAs en FPGA verdelingen

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

In this thesis, the objective is to explore the possibility of creating an operating system independent framework. This framework should allow users to quickly and consistently create, modify, execute and evaluate testbenches. To accomplish all of this, a number of industry standard tools will be incorporated around a central Python script. This combination should ensure timely and automated building, testing and test report generation.

2 Introduction

2.1 Hardware Description Languages

A Hardware Description Language (HDL) can be used to describe digital electronics, i.e. hardware, in different levels. The level that uses certain blocks of logic gates to describe more complex behaviour is called the Register Transfer Level (RTL). Some blocks are standard implementations that have been widely used and are nearly fully optimized, such as memories, flip-flops and clocks. VHDL is such a language, having been developed in the eighties of the twentieth century, originally to have a uniform description of hardware brought in by external vendors to the U.S. department of defence. It was quickly realized that simulation was possible with a correct description and the language evolved to be used as such.[1, 2] The final step was to create tools that could not only simulate, but also synthesize (i.e. create actual hardware layouts) from these descriptions.[3–6] An RTL flip-flop implementation written in VHDL is shown here:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY dff IS
    PORT(d    : IN  std_logic;
         clk  : IN  std_logic;
         q    : OUT std_logic;
END dff;

ARCHITECTURE Behavioural OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            q <= d;
        END IF;
    END PROCESS;
END Behavioural;
```

The Institute of Electrical and Electronics Engineers (IEEE) 1164 library provides a number of extensions on the original VHDL IEEE 1076 specification that allow a more realistic simulation and description of hardware behaviour.[7] An *entity* defines the inputs and outputs of a certain building block, in this case the D Flip-Flop (DFF). The *architecture*, in this case named *Behavioural*, takes the description of an entity and assigns a real implementation to it. In this architecture, we have one process and it is *sequential*, meaning that every update in the process follows an update of *clk*, a clock signal. The d stands for delay, and it simply puts on its output *q*, that which was on the input *d*, one clock period earlier. 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 until the next rising edge appears.

This is a basic example of an entity, an architecture and a process. Before the code can be put to use in a working environment, it needs to be tested first. This is done through the use of *testbenches*.[8] Testbenches are made up of code that takes a certain building block, the Unit

Under Test (UUT) or Device Under Test (DUT). The specialized tools mentioned above take the testbenches and read them. Inside are usually instructions to put a certain sequence of values on the inputs and monitor the outputs.[9] Applying this to the example listed earlier, the testbench would contain a process with a clock, signals coupled with the ones in the entity, some stimuli and *wait* statements. The signals are linked to the DFF, which is now the UUT. Then the clock starts ticking and the input d is made '0' or '1' every now and then. All that is left is to assure the output q is always '0' and '1' exactly one clock cycle later. The full code is listed in appendix A.1.

If the device performs normally, the received output sequence should match an 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 moved further down the developmental process.[10]

The higher level components, such as a registry, employ a number of the lower level ones to create more complex logic.[6] The 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.[11] 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. If a device is not tested properly and faults propagate throughout development, they can be very expensive to correct.[12] Therefore a large portion of time is spent writing and executing tests.

3 Current industry practices

A number of practices exist to improve speed and quality of testing and coding. Many of these practices are already applied in the software development industry, where this is quite literally their entire business. Moving to HDLs, it stands to reason that code meant for synthesis may not be able to follow all of these practices. However, the industry has formed several practices of its own to try and create a uniform verification process. The most well-known will be discussed in some detail below.

3.1 Assertions

Assertions are the standard practice of verification. An assert in VHDL is very simple: check whether some boolean condition is true.[13] If true do nothing, if false return the error message and throw an error of a certain level, as in the following example:

```
assert (not Q) report "Unexpected output value" severity failure;
```

In this example, an error is thrown of the severity *failure*, which ends the simulation, if the value on the output Q (see section 2.1)) isn't *logically* true. In this case it means the output has to have a value of *high* or *1*. The severity of the assertion can be in the range of notice to failure, and the simulator might be set to respond or stop only to a certain level of severity. Doing this for the right values at intervals gives the developer a quick overview of whether everything went according to plan. After all, if things went wrong, the assertion should have thrown an error here or there.

3.2 The generic testbench

The generic testbench operates as mentioned in section 2.1. It assigns every input and output on the DUT their counterpart in the testbench and contains a number of processes with stimuli.[14] Standard procedure is:

1. Wait for some clock periods to 'ready' the design
2. Apply stimuli to the inputs
3. Wait for the appropriate number of clock cycles
4. Use asserts to check whether the outputs have the right values
5. Repeat steps 2 through 4 until satisfied
6. End with an infinite 'wait' to suspend the process

Creating this kind of testbench for a large, hierarchical project would certainly become lengthy and unclear. To counter these disadvantages, a number of practices were created to keep control over what and how designs are tested. The more used of these practices are explained below.

3.3 Coverage

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

3.3.1 Code Coverage

In development, Code Coverage (CC) is a type of measurement to indicate how well the source code has been tested. With the use of a coverage report, unused blocks of code can be uncovered, these blocks might indicate unnecessary code or a bug. Imagine a Finite State Machine (FSM) with an unused reset state, this might indicate that the reset isn't functioning properly or there are no tests covering the reset. However, CC does not provide any real functional analysis. It does not indicate any missing lines of code nor does it tell you whether the inputs and outputs behave properly.[15, 16]

3.3.2 Functional Coverage

Functional Coverage (FC) is the practice of measuring whether the UUT meets with certain specifications at specific times during the testing process. These specifications are created by the developer and are used to check whether the design performs as expected. Good practice is to include corner cases, cases that cover very rare occurrences and so on. That way, the device is sure to be in working condition even under unexpected circumstances. As such, FC provides the type of coverage CC lacks.[17, 18]

3.4 Verification

3.4.1 Constrained Random Verification

Constrained Random Verification (CRV) is an industry practice where one or more inputs are generated randomly, within certain bounds or *constraints*.[19, 20] The output is then accepted as correct if it too is within certain (other) constraints. This practice was brought into use after designs grew too large for Directed Testing (DT) to support. DT has verification engineers write out very specific things they want to test, for instance, a reset pulse to verify the reset working correctly. CRV opposes this with the idea that for all behaviour to be tested properly in large designs, the amount of time spent writing and executing tests would simply become too great. It proposes a solution where inputs are generated randomly, within certain bounds, but in a sufficiently large quantity to have implicitly covered all scenarios. It is important to note that in DT, expected behaviour is directly tested, but in CRV it is likely to be the unexpected behaviour that gets tested too. This solved the long standing problem of testing any behaviour, including the unexpected.

3.4.2 Formal Verification

On top of the aforementioned, there are several more practices that have unique ways of verifying the properties of a design, but aren't used sufficiently to merit full detailing. Formal verification is such a practice, where the core idea is to mathematically prove the design from its specifications.[13, 21] The upside is that the design is completely verified, however, it is out of use because proving large designs is not only tedious but takes up large amounts of man-hours.

3.5 Simulation tools

As mentioned in section 2, HDLs are used for developing hardware, and need to be tested and build as such. Like many other programming language, they need a dedicated compiler to fault-check the code and build the binaries. Unlike those other programming languages they need a simulator in order to verify the builds. There exist many compilers and simulators, almost none are exclusive for VHDL, almost all are dual-language and support some form of Verilog, a different HDL. However, many of them refuse to support the latest additions to the VHDL language specification, even the 2002 additions are hard to be found.[22–25]

3.5.1 ModelSim

ModelSim is the simulator that was investigated for several reasons. Firstly, a free student edition was available. Considering licenses outside of the educational system can easily cost upward of $25,000 (€20,250 at time of writing) this made it ideal for any thesis or student related work. Proof of this is the extensive use of ModelSim in the courses related to HDL development. Secondly, ModelSim supports all versions of VHDL, which is the HDL we are processing. Even in 2014, only one other tool was found to support all of VHDL-2008, which is Aldec's *Active-HDL*. And last but not least, ModelSim is one of the industry's most used simulators, enabling a pool of experience to be consulted.[26]

4 Software development practices

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

4.1 Test Driven Development

Test Driven Development (TDD) is a proven development technique that has regained traction in the past decade, primarily through the efforts of Kent Beck.[27, 28] This practice has proven to increase test coverage , decrease defect density [29] as well as improve code quality.[29–31] The technique focuses on tests being created before the actual code. It is important to make certain distinctions before going more in depth on the used methods. The developing community has a great many practices, each with their own names and methods, and hardly none are mutually exclusive.

4.1.1 Unit testing

To understand TDD, a basic knowledge of *unit testing* is required. In software testing, a unit test is a test designed to check a unit of code.[**VHDLunit**, 32] Ideally, this unit is the smallest piece in which the code can be divided. A unit test should always test only a single entity, and only one aspect of that entity's behaviour. This division in units has a number of benefits, one of the most important being that code is exceedingly easy to maintain. Furthermore, the division of the code makes changes, when needed, fast to be carried out and ensures that only the modified code needs to re-tested.

4.1.2 Test First Development

Another main component of TDD is Test First Development (TFD), a technique that has a developer writing tests first, before any code has been written.[32] This method puts behaviour as the primary drive for development. After all, if the test is conceived before the code it tests, the goal is to make the tests pass by implementing the behaviour it tests. This has developers think about what the code should do rather than what the exact implementation has to be. A key feature of a new test is that it has to fail during its first run. If not, the test is obsolete seeing as the functionality it tests has already been implemented. After being run for the first time (and failing), the developer implements just enough code to get the test to pass. Once the test succeeds, it is time for a new test.

4.1.3 Refactoring

The third pillar of TDD is refactoring. After the tests succeed, it is necessary to clean up the code.[33] A well-done TFD implementation uses the bare amount of code needed to make the test pass, but this might not be the best code possible. Refactoring means that you take existing code and modify it to perform better. This should leave 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.

4.1.4 Test Driven Development

Test Driven Development is a combination of the previously mentioned techniques. A Unit Test is written before any code is, 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 better and be less complex. This follows all steps mentioned above, a *Unit Test* is written according to *Test First Development* and is *Refactored* later.

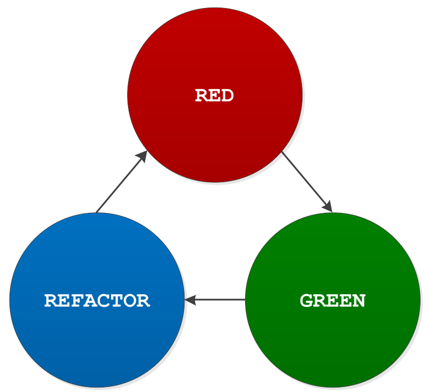

Figure 1: Three step TDD design flow

4.2 Continuous Integration

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

4.2.1 Revision Control

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

4.2.2 Build Automation

A different 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 important in a proper CI system as will be discussed further on.

4.2.3 Test Automation

As the code is built 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 results and reports and preferably visualise them in a way that makes them easy to read and locate faults.

Combining all of these practices saves developers, and by extension the company they might work for, a lot of time and 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 solutions exist that employ open or widely used standards.[34–36]

4.3 xUnit

xUnit is a collection of frameworks that all follow the same basic principle. Its specification defines several key components for a testing framework. They encompass an implementation of a *testcase* which contains a single unit test. These testcases are then enveloped in *testsuites*, which are groups of tests that need the same conditions before they can be executed. There are more implementation details but these will not be discussed.[37]

JUnit is an implementation for the Java programming language of the xUnit specification. A useful part of the specification is a standard format for the test reports. The reports are written in eXtensible Markup Language (XML), which is a type of language used for formatting data. The JUnit format is supported by numerous (open source) tools, which makes it an ideal candidate for further investigation.[38–40]

4.4 Python

Python is a high-level computer programming language that first appeared in 1991, with the third major revision being released in 2008. It supports object-oriented and structured programming and is easy to use as a scripting language. A great feat of Pythons community is that there are many (open source) libraries available for just about any function that comes to mind. This on top of the already impressive amount of libraries that Python itself supports. In addition to this, Python has all of its standard features explained in great detail with good examples in its online documenting system. The combination of these features allows any programmer, even with very little know-how, to quickly put together anything that comes to mind.[41]

Part II
Developing a framework

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

5 Outlining

Before work began on developing the framework, there needed to be some boundaries set and goals to work towards. As VHDL development is done on both Windows and Linux, it made sense to keep the whole framework as platform independent as possible. To add to this, the framework needsed to be developed in the short span of under a year by one developer who had little experience doing so. The language chosen thus had to be easy to master and work on all platforms with as few as possible modifications.

5.1 First draft

In section 3 some current industry practices were discussed and one of the very basic VHDL testing methods was found to be assertions. In order to remain as broad as possible, work started with processing these assertions first. As mentioned in section 4.1, unit testing is a method that tests an as small as can be part of a code's behaviour. Applied to VHDL, a unit test would then be the part of the code that the assertion checks. This under the assumption that the assertion checks for simple boolean truths such as *output q equals '1'* and not a complex mix of logic.

To the likeness of unit testing, these parts of code were separated into their own testbenches. Methods of separation were not available, and thus one was developed. For complete accuracy, a lexical analyzer for VHDL would have been needed, but the development of such was worth its own thesis and as such a more simple method was used. Instead, estimation of the level of depth in the testbench structure was made using VHDL keywords such as *process* and *begin*.

Furthermore, because lexical analysis was out of the question, the constructs that each assert relied on were unable to be identified. A solution was found in the form of procedures and functions. These were placed in the architecture header, and with these single-line unit test could be called from the architecture body. All that was left for the parser to do was to find the location of these lines and place each of them in a newly created testbench.

5.2 Draft review

Even though it sounds simple, this method was still needlessly complicated and a better method had to be devised. Even then, the parser created a heap of little, seperate testbenches which added more work for the developer as he had to execute all of these separately. A possible solution was the automatic execution and result capture of the created testbenches, reducing the amount of work to a lightly modified testbench and a single execution of the parser.

5.3 Fresh start

As everything up until now was based on code created on the fly, it became clear that the original files weren't ready to be refactored or otherwise modified. The experience gained from writing everything mentioned in section 5 was put to good use, and a clean build was started. With this build a number of assumptions were made:

- Optional arguments to modify the behaviour should be specified
- Full on automation should be included
- A log detailing events should be held
- Proper documentation should be provided

All of these modifications made the framework more suitable for developers. The optional arguments would allow for a greater control over the program. The automation countered the extra work created from the generation of all the separate testbenches. The log made it easier to figure out where things went wrong should they go wrong. And lastly, the documentation was a given for any decent project.

6 Using the framework

For the framework to be properly usable, a number of things had to be considered. Firstly, not all testbenches were suited to be processed by the parser; it relied heavily on the independence of tests from one another. Therefore, tests that relied on outputs from previous tests needed to be kept together, which lead to a large portion of such testbenches needing to be completely rewritten.

Secondly, even with properly suited testbenches, the output needed to remain uniform and recognizable. As mentioned in section 5.1, to have this done automatically would have required a full blown lexical analysis which was simply too complex. As a solution, the use of a testing library was proposed which would provide uniform tests and output.

6.1 Preparing the testbench

For the testbench to be properly parsed, it needed some preparation first. The blocks of tests, henceforth referred to as testsuites, were to be independent and separated by keywords that signify the beginning and end of a testsuite. Not only this, but the standard asserts were to be replaced with functions from the library, and the library of course needed to be included. An example using the DFF from section 2.1 (full code in appendix A.1):

```
...
    assert q = '0'
        report "Wrong output value at startup" severity FAILURE;
    d <= '1';
    WAIT FOR clk_period;
    assert q = '1'
        report "Wrong output value at first test" severity FAILURE;
...
```

Would turn into (See appendix A.2):

```
...
    --Test 1
    check_value(q = '0', FAILURE, "Wrong output value at startup");
    write(d, '1', "DFF");
    check_value(q = '1', FAILURE, "Wrong output value at first test");
    ...
    --End 1
...
```

In the full code it is shown that the functions are overloads of existing functions in the *BitVis* library, which will be discussed in section 6.5. Also visible here are the keywords --*Test* and --*End* with which the testsuites are separated. The number next to it is a testsuite identifier, but was not necessary as the parser kept track of the count.

Overloading functions, using libraries and seemingly completely overhauling the testbench may appear to be like a lot of work for a small testbench such as this. While true, this method was easily scalable for testbenches of much larger sizes and with designs of greater complexity. If the developer created a testbench starting with this approach, no time would have been wasted redoing any code and the readability would have been greatly improved. Lastly, automated logging was built in these procedures to keep track of progress during testruns. This allowed for easier and uniform report generation (see section 6.3).

6.2 Calling the parser

The parser was created as a script, written in Python, thus it was necessary to have access to a command-line interface as python requires one to run its scripts. Originally, using the parser was as easy as typing *python testbench_parser.py tb_dff.vhd* in the command-line. There was one testbench to be parsed and everything happened automatically according to default methods, hardcoded in the script. During further development it became clear that there were many ways a developer might want to test their project. As such there was a need for a way to modify the parser's behaviour when it ran. Possible solutions included different scripts, a settings file but ultimately the decided upon method was the use of arguments.

6.2.1 Arguments to the parser

Python had a built-in argument parser, in the library *argparse*. With this it became possible to have a complex yet simple to use command-line interface with both optional and required arguments. Included in the library was the option for a clear and thorough help, providing users with all the information they needed to run the script. A simple example is listed below:

```
$ python src\testbench_parser.py -m partitioned -l  Project\sim\tb_dff_r.vhd -c
```

By using this command, the testbench from appendix A.2 was processed using the *partitioned* method, which is the method that uses the keywords *Test* and *End* (see section 6.2.2). The *-c* flag ensured that output of the simulator (in this case ModelSim) was displayed and not just captured and stored as was default. Below are all the flags and their help text:

- '-c', '--cmd' specifies output to be displayed on the command line
- '-d', '--dest' specifies a custom path for the log
- '-f', '--file' specifies -l/--list is a file with a list of .vhd files
- '-l', '--list' specifies the list of .vhd files to be processed (at least one)
- '-m', '--method' specifies what method was used to write the testbench
- '-p', '--precompiled' specifies location of the precompiled dependencies, requires full path
- '-r', '--runops' specifies custom arguments for simulation start
- '-v', '--version' specifies a different VHDL version, default is 2008

With this parser, it also became possible to run multiple testbenches with one call to the script. This allowed to test multiple levels of a hierarchic design, or even completely different designs in one run. Most of the flags are self-evident, but those that are not are explained below.

The *-p* or *--precompiled* flag points to a text file that contains instructions very similar to those found in a unix *Makefile*. They are commands that need to be executed before the testbench can be compiled. This would be all of the source files used, in their specific order and any special libraries that need to be created beforehand. A full editor might have recognized the dependencies and libraries on its own and created a Makefile or similar accordingly but here the lack of a decent lexical analysis prevented this.

The-*r* or *--runops* flag is to specify a custom command for ModelSim to run during the execution of the newly made testbenches. The default value is *-do "run -all;exit"*, which simply starts the simulation until its end and then exits. However, developers might want to load different memory files in a memory or specifiy a custom file with more elaborate commands.

6.2.2 Processing methods

In section 6.2.1 the processing method *partitioned* was mentioned as well as the *-m* or *--method* flag. The parser thus supported multiple methods of dividing the testbench, they are listed below:

- Start Stop with identifiers
- Line per line
- Partitioned with identifiers

In the Start Stop method, every single line between certain keywords was assumed to contain a test. It was assumed that everything above the keywords was architecture header and everything below was a general 'end of the architecture body'. This method was the easiest to parse, it looked for *--Start* and created a new testbench with every line until *--Stop*.

In the Line method, the parser looked for a process. This process should contain one test per line. It tried to identify this process on its own and as such, could easier make a mistake than the Start Stop method.

Finally, in the Partitioned method, the developer indicated which blocks are testsuites by using the *--Test* and *--End* keywords. Everything in between these keywords was assumed to be a single block of testcases, to be put in the same new testbench. As such, and unlike the previous two methods, it was possible to define multiple testsuites.

6.3 Report and log generation

After the parser from section 6.2 had run its course, results were captured automatically. However, making sense of a whole lot of command line output from a text file was challenging. ModelSim made this more difficult by adding a header and a lot of information that isn't really interesting at that time of development. Things such as which files were loaded prior to the execution and which preference file was being read were not needed for a quick overview. However, they might have still be needed in the event of severe failure.

6.3.1 Basic report

A great deal of filtering of the output file was required, only the passed and failed testresults were needed for a proper report. Whether tests passed or failed was outputted with easy to recognize keywords. A simple filtering on these words per line of the output file returned every testresult, neatly ordered in passed and failed groups. A total testcount was tallied from the combination, and as such a crude report was written to a text file.

6.3.2 XML report

Previously discussed in section 4.3, the JUnit implementation of xUnit uses XML for report viewing. The editor that was used throughout the development of the thesis is *Eclipse*, which was written in Java, the same language JUnit is an implementation for. They could be read and displayed in Eclipse by manually navigating to them or editing the project properties to automatically read them.

To create the XML reports, a pre-made, open-source Python implementation of a JUnit report generator, aptly named python-junit-xml was used. This package required its own external component to be installed, called setuptools.[39, 42] In this thesis, versions 1.0 and 3.5.1 respectively were used, but versions 1.3 and 8.0.4 are available at time of writing.

6.3.3 Logkeeping

Finally, a log was kept detailing events throughout the parser's execution. It contained critical events along with timestamps for precise debugging. Everything within this log file was related to only the parser and its workings. The file that contained details about ModelSims execution was not saved.

6.4 Hudson-CI

For full automation, a CI system was integrated into the framework. The CI solution that was incorporated was Hudson-CI which provided an extensive range of features including:

- Timed and triggered building from an RC repository
- Automated testing of builds
- Reading reports in the JUnit format
- Graphical and statistical overview of test results throughout builds

Hudson was available as a running .war file (a Java web application that runs on a local machine) or as an installed service.[43] The version used throughout this thesis is 3.2.0 but at time of writing 3.2.1 is available.

6.4.1 Using Hudson-CI

For ease of use, Hudson was installed as a local service. By default it is accessed at port 8080 of the local machine, usually by typing *localhost:8080* in a browser. The following steps were followed to successfully complete several testruns on a Windows 8 machine:

1. Create a new job with:
   (a) Git repository as SCM
   (b) The batch command listed in section 6.2.1.
   (c) JUnit test results published under *VHDL_TDD_Parser/*/*.xml*
   (d) Manual build activation
2. Press the *Build now* button

This automatically downloaded the source, compiled the external testbench for the DFF, executed the different architectures, captured and formatted the output and published the XML report. For successive runs only step 2 had to be repeated, the code is automatically updated from git should it change. An example of captured statistics is shown in appendix B.1.

6.5 Bitvis utility library

Throughout the development of the framework multiple approaches were used to enhance developer experience including a library with widely used and usable functions. At first this was a self developed library, however, after a short while the existing *Bitvis utility library* was used. It was developed by the Norse company Bitvis for verification and compatible with all versions of VHDL. [44] The library promoted overloading its functions with implementations for the project that was being developed. It also provided an easy to use logging function with great customizability, which made it ideal for use in the framework.

To make use of the library, all that needed to be done was to include it in the project, use its functions and overload where needed. For the framework, the use of the *check* or *check_value* function was critical seeing as the test parser relied on the output created by this function to assess test success. An example is listed below (full code in appendix A.2):

```
...
LIBRARY bitvis_util;
USE bitvis_util.types_pkg.ALL;
USE bitvis_util.string_methods_pkg.ALL;
USE bitvis_util.adaptations_pkg.ALL;
USE bitvis_util.methods_pkg.ALL;
    ...
    PROCEDURE log(
        msg     : STRING) IS
    BEGIN
      log(ID_SEQUENCER, msg, C_SCOPE);
    END;

    PROCEDURE write(
        SIGNAL    data_target   : IN STD_LOGIC_VECTOR;
        CONSTANT data_value     : IN STD_LOGIC_VECTOR;
        CONSTANT msg            : IN STRING) IS
    BEGIN
        data_target <= data_value;
        WAIT FOR clk_period;
        log(msg & " write(Target:" & to_string(data_value, HEX, AS_IS, INCL_RADIX)
            & ", " & to_string(data_target, HEX, AS_IS, INCL_RADIX) & ")";);
    END;
    ...
        write(d, '1', "DFF");
        check_value(q = '1', FAILURE, "Wrong output value at first test");
    ...
```

7 Future work

While a working framework was achieved, many parts are missing for a usable whole. Below are listed some of the more critical absent pieces as well as suggestions that are not necessarily needed for a working framework but might make development more pleasant.

7.1 Wider tool support

In section 3.5 it was decided to support ModelSim as the compiler for the framework. Despite its extensive use in the classroom, it is not the only tool at the top. Different tools might use other methods, accept different input and produce different output. While there are too many tools to support them all, either a general approach can be achieved or the most widely used can be integrated.

On the topic of tools, the current support of ModelSim itself is limited at best. Only the very basic steps have been taken to assure compiling. Complex projects might need an extensive list of options to be run before compiling, during simulation or otherwise.

7.2 Lexical analysis

In section 5 it was decided that lexical analysis was out of the question. However, future modification should include this as the advantages greatly outweigh the work that it requires. A full lexical analysis should allow for more advanced methods of testbench creation to be supported, as well as a precise integration for current methods. Furthermore, a good analysis could separate a testbench on its own, seeing dependencies and eliminating code duplication. And lastly, a great analysis might even be able to convert testbenches to the used methods by recognizing groups of test, make its own functions out of them and saving developers a lot of time in the process.

7.3 Adapted CI tool

In section 4.2 continuous integration was introduced and in section 6.4 Hudson-CI was proposed as the CI tool. However, Hudson-CI is a software development tool meant for software development practices. Its inner workings have shortcomings that, while not life-threatening, are needed for hardware developers. The JUnit reports, for instance, do not recognize sub-millisecond times, and debugging output of the compiler could be integrated. A dedicated hardware CI tool might be developed to provide full support for all the features that are lacking.

8 Commentaries on developing VHDL

During the making of this thesis a great deal of time was spent working with VHDL and testbenches in particular. Evaluating, re-evaluating and dissecting these testbenches to find ways to improve their workings and their developing environment brought good familiarity with the inner workings. As such, certain aspects of the language and its surroundings or lack thereof were the proverbial thorn in the eye and could stand to be improved.

8.1 An industry stuck in 1993

in section 3 industry standard practices were discussed along with some considerations on the tools used in section 3.5. While there definitely is work being done to improve the coding practices, the hardware development world is slow to adapt to any change. The biggest proof of this is that just now (Q4 2014) slow adaptations are being made for the latest VHDL standard, which was defined in 2008. The lack of willingness to improve upon working methods -"*Don't fix what isn't broken.*"- stifles innovation. Luckily for the language the industry is a very niche world of behemoths that doesn't see many competitors gain much traction fast.

8.2 Lack of reflection or introspection

Reflection and introspection are two programming concepts, the former already existing since assembly languages were used. Reflection means that a programming language can modify parts of its innards at runtime. While this certainly sounds not synthesizable, it could bring about a revolution in the creation of functions for testbenches. Instead of having to overload for an unreasonable amount of different configurations, code would simply adapt to the situation by adjusting its arguments to the right type. This, of course, providing the function's structure continues to work after the modifications.

Introspection can be seen as a part of reflection, it is the ability to examine an object at runtime. While VHDL is not an object-oriented programming language, signals, variables and constant can be seen as a type of object. They have many possible configurations and can have properties such as vector length. Being able to examine these properties at runtime and matching object types against each other would provide for some of the same functionality mentioned with reflection.

8.3 Hardware developers are not software developers

While considering the previous commentaries it became clear that a major shortcoming of hardware developers is that they are not trained to develop software. Software developers learn certain practices, methods, group works and the likes in the course of their education. Hardware developers usually come from an engineering background where sciences and knowledge of *what* is being developed is taught. This creates a work environment where seldom there is a developer taught in software development methods. Combined with the slow and stagnant change mentioned in section 8.1 this leads to outdated development practices.

9 Conclusion

At the start of this thesis, it was stipulated that developing VHDL is prone to errors and that the testbench and its environment play an integral part in locating and solving these. As a solution, a testing framework was proposed that would work around a central script and integrate ideas from the software development world.

A first draft revealed many shortcomings of the methods investigated and a subsequent revision was scheduled. More detailed outlines provided better planning and easier incorporation of the envisioned methods as well as a better coding practice.

The framework was made to integrate one of the most commonly used simulation tools, ModelSim. Around it, the software development practice of continuous integration was built using the Hudson-CI tool. Hudson-CI was used to retrieve the code from an RC repository, call the script and display its output.

The central script was made to accept different forms of modified testbenches and it was made possible to modify the behaviour of the script with command-line arguments. A full text output report, a text summary and a processed report in the JUnit XML format were made available after compilation and simulation.

Compilation and simulation test runs were performed with a simple example and found to be working as expected. Runs with more complicated testbenches were found to be lacking in support and not fully completing.

The objective of developing a framework was generally achieved, but it severely lacked in implementation details. Better support for the current tools and support for more a larger variety of tools is absolutely necessary for a working framework. Finally, it was most definitely shown that applying software development practices is possible and useful in a hardware development environment.

References

[1] Roger Lipsett, Erich Marschner and Moe Shahdad. "VHDL - The Language". In: *IEEE Design & Test of Computers* 3.2 (Apr. 1986), pp. 28–41. DOI: `10.1109/mdt.1986.294900`.

[2] J.H. Aylor, R. Waxman and C. Scarratt. "VHDL - Feature Description And Analysis". In: *IEEE Design & Test of Computers* 3.2 (Apr. 1986), pp. 17–27. DOI: `10.1109/mdt.1986.294899`.

[3] R.D. Acosta, S.P. Smith and J. Larson. "Mixed-mode simulation of compiled VHDL programs". In: *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. IEEE Comput. Soc. Press, 1989. DOI: `10.1109/iccad.1989.76930`.

[4] T.E. Dillinger et al. "A logic synthesis system for VHDL design descriptions". In: *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. IEEE Comput. Soc. Press, 1989. DOI: `10.1109/iccad.1989.76906`.

[5] R. Camposano, L.F. Saunders and R.M. Tabet. "VHDL as input for high-level synthesis". In: *IEEE Design & Test of Computers* 8.1 (Mar. 1991), pp. 43–49. DOI: `10.1109/54.75662`.

[6] R. Waxman, L. Saunders and H. Carter. "VHDL links design, test, and maintenance". In: *IEEE Spectr.* 26.5 (May 1989), pp. 40–44. DOI: `10.1109/6.30762`.

[7] *1164-1993 - IEEE Standard Multivalue Logic System for VHDL Model Interoperability*. 1993. DOI: `10.1109/ieeestd.1993.115571`.

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

[9] D. Biederman. "An overview on writing a VHDL testbench". In: *Proceedings The Twenty-Ninth Southeastern Symposium on System Theory*. IEEE Comput. Soc. Press, 1997. DOI: `10.1109/ssst.1997.581677`.

[10] F. Wotawa. "Debugging {VHDL} designs using model-based reasoning". In: *Artificial Intelligence in Engineering* 14.4 (2000), pp. 331–351. ISSN: 0954-1810. DOI: `10.1016/S0954-1810(00)00021-2`.

[11] P. Kission, Hong Ding and A.A. Jerraya. "VHDL based design methodology for hierarchy and component re-use". In: *Proceedings of EURO-DAC. European Design Automation Conference*. IEEE Comput. Soc. Press, 1995. DOI: `10.1109/eurdac.1995.527446`.

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

[13] R. Reetz, K. Schneider and T. Kropf. "Formal specification in VHDL for hardware verification". In: *Proceedings Design, Automation and Test in Europe*. IEEE Comput. Soc, 1998. DOI: `10.1109/date.1998.655865`.

[14] W.G. Swavely et al. "A generic VHDL testbench to aid in development of board-level test programs". In: *Proceedings of AUTOTESTCON '94*. IEEE, 1994. DOI: `10.1109/autest.1994.381506`.

[15] T.W. Williams et al. "Code coverage, what does it mean in terms of quality?" In: *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179)*. IEEE, 2001. DOI: `10.1109/rams.2001.902502`.

[16] Albert Tran, Michael Smith and James Miller. "A Hardware-Assisted Tool for Fast, Full Code Coverage Analysis". In: *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2008. DOI: 10.1109/issre.2008.22.

[17] Alfonso Martinez Cruz, Ricardo Barron Fernandez and Heron Molina Lozano. "Automated Functional Coverage for a Digital System Based on a Binary Differential Evolution Algorithm". In: *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*. IEEE, Sept. 2013. DOI: 10.1109/brics-cci-cbic.2013.26.

[18] S. Regimbal et al. "Automating functional coverage analysis based on an executable specification". In: *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.* IEEE Comput. Soc, 2003. DOI: 10.1109/iwsoc.2003.1213040.

[19] Ankit Gopani. *Importance of Constrained Random Verification Approach*. 2012. URL: http://asicwithankit.blogspot.be/2012/04/importance-of-constrain-random.html (visited on 20/12/2014).

[20] Nathan Kitchen and Andreas Kuehlmann. "Stimulus generation for constrained random simulation". In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, Nov. 2007. DOI: 10.1109/iccad.2007.4397275.

[21] D. Deharbe, S. Shankar and E.M. Clarke. "Formal verification of VHDL: the model checker CV". In: *Proceedings. XI Brazilian Symposium on Integrated Circuit Design (Cat. No.98EX216)*. IEEE Comput. Soc, 1998. DOI: 10.1109/sbcci.1998.715418.

[22] Aldec Inc. *Active-HDL Configurations*. 2014. URL: https://www.aldec.com/en/products/fpga_simulation/active-hdl (visited on 19/12/2014).

[23] Cadence. *Incisive Enterprise Simulator*. 2014. URL: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx (visited on 19/12/2014).

[24] Xilinx. *Vivado Synthesis*. 2014. URL: http://www.xilinx.com/support/answers/62005.html (visited on 19/12/2014).

[25] Altera. *Quartus II support for VHDL 2008*. 2014. URL: http://quartushelp.altera.com/14.1/master.htm#mergedProjects/hdl/vhdl/vhdl_list_2008_vhdl_support.htm?GSA_pos=1&WT.oss_r=1&WT.oss=vhdl%202008 (visited on 19/12/2014).

[26] Altera. *ModelSim 10.0 release notes*. 2014. URL: http://www.altera.com/download/os-support/release-notes_10.0c.txt (visited on 19/12/2014).

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

[28] Scott Wambler. *Introduction to Test Driven Development (TDD)*. 2002. URL: http://agiledata.org/essays/tdd.html (visited on 14/03/2014).

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

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

[31] B. Thirumalesh and N. Nachiappan. "Evaluating the efficacy of test-driven development: industrial case studies". In: *ISESE*. ACM, 2006, pp. 356–363. ISBN: 1-59593-218-6. DOI: 10.1145/1159733.1159787. (Visited on 23/01/2007).

[32] Don Wells. *Code the Unit Test First*. 2000. URL: http://www.extremeprogramming.org/rules/testfirst.html (visited on 20/03/2014).

[33] Hui Liu, Yuan Gao and Zhendong Niu. "An Initial Study on Refactoring Tactics". In: *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, July 2012. DOI: 10.1109/compsac.2012.31.

[34] Fazreil Amreen Abdul and Mensely Cheah Siow Fhang. "Implementing Continuous Integration towards rapid application development". In: *2012 International Conference on Innovation Management and Technology Research*. IEEE, May 2012. DOI: 10.1109/icimtr.2012.6236372.

[35] James H. Hill et al. "CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments". In: *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*. IEEE, Mar. 2008. DOI: 10.1109/ecbs.2008.20.

[36] Sean Stolberg. "Enabling Agile Testing through Continuous Integration". In: *2009 Agile Conference*. IEEE, Aug. 2009. DOI: 10.1109/agile.2009.16.

[37] Martin Fowler. *Xunit*. URL: http://www.martinfowler.com/bliki/Xunit.html (visited on 02/06/2014).

[38] J. Kivi et al. "Extreme programming: a university team design experience". In: *Canadian Conference on Electrical and Computer Engineering*. CCECE '00. IEEE, 2000, pp. 816–820. ISBN: 0-7803-5957-7. DOI: 10.1109/CCECE.2000.849579.

[39] Brian Beyer. *Python JUnit XML*. 2014. URL: https://pypi.python.org/pypi/junit-xml/1.3 (visited on 07/05/2014).

[40] *JUnit*. URL: http://JUnit.org (visited on 25/07/2014).

[41] *Python*. URL: https://www.python.org/ (visited on 28/12/2014).

[42] Jason R. Coombs and Phillip J. Eby. *setuptools*. 2014. URL: https://pypi.python.org/pypi/setuptools (visited on 07/05/2014).

[43] *Hudson-CI*. 2014. URL: http://hudson-ci.org/ (visited on 25/05/2014).

[44] *Bitvis Utility Library*. 2014. URL: http://bitvis.no/products/bitvis-utility-library/ (visited on 25/08/2014).

Appendices

A Code examples

A.1 DFF testbench

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY tb_dff_r IS
END tb_dff_r;

ARCHITECTURE Behavioural OF tb_dff_r IS
    COMPONENT dff
    PORT(
        d   : IN  STD_LOGIC;
        clk : IN  STD_LOGIC;
        q   : OUT STD_LOGIC;
    END COMPONENT;

    SIGNAL d   : STD_LOGIC := '0';
    SIGNAL clk : STD_LOGIC := '0';
    SIGNAL q   : STD_LOGIC := '0';

    CONSTANT clk_period : TIME := 10 ns;

BEGIN
    uut: dff PORT MAP (
        d   => d,
        clk => clk,
        q   => q
        );
    clk_process : PROCESS
        BEGIN
            clk <= '0';
            WAIT FOR clk_period/2;
            clk <= '1';
            WAIT FOR clk_period - clk_period/2;
    END PROCESS;

    stim_proc: PROCESS
    BEGIN
        WAIT FOR clk_period;
        ASSERT q = '0'
            REPORT "Wrong output value at startup" SEVERITY FAILURE;
        d <= '1';
        WAIT FOR clk_period;
        ASSERT q = '1'
            REPORT "Wrong output value at first test" SEVERITY FAILURE;
        d <= '0';
        WAIT FOR clk_period;
        ASSERT q = '0'
            REPORT "Wrong output value at final test" SEVERITY FAILURE;
        WAIT;
   END PROCESS;
END Behavioural;
```

A.2 Revised DFF testbench

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY bitvis_util;
USE bitvis_util.types_pkg.ALL;
USE bitvis_util.string_methods_pkg.ALL;
USE bitvis_util.adaptations_pkg.ALL;
USE bitvis_util.methods_pkg.ALL;

ENTITY tb_dff IS
END tb_dff;

ARCHITECTURE Behavioural OF tb_dff IS
    COMPONENT dff
    PORT(
        d   : IN  STD_LOGIC;
        clk : IN  STD_LOGIC;
        q   : OUT STD_LOGIC;
    END COMPONENT;

    SIGNAL d   : STD_LOGIC := '0';
    SIGNAL clk : STD_LOGIC := '0';
    SIGNAL q   : STD_LOGIC := '0';
    CONSTANT clk_period : TIME := 10 ns;

    PROCEDURE log(
        msg   : STRING) IS
    BEGIN
      log(ID_SEQUENCER, msg, C_SCOPE);
    END;

    PROCEDURE write(
        SIGNAL   data_target  : IN STD_LOGIC_VECTOR;
        CONSTANT data_value   : IN STD_LOGIC_VECTOR;
        CONSTANT msg          : IN STRING) IS
    BEGIN
        data_target <= data_value;
        WAIT FOR clk_period;
        log(msg & " write(Target:" & to_string(data_value, HEX, AS_IS, INCL_RADIX)
            & ", " & to_string(data_target, HEX, AS_IS, INCL_RADIX) & ")";);
    END;
BEGIN
     uut: dff PORT MAP (
        d   => d,
        clk => clk,
        q   => q
        );
     clk_process : PROCESS
        BEGIN
            clk <= '0';
            WAIT FOR clk_period/2;
            clk <= '1';
            WAIT FOR clk_period - clk_period/2;
    END PROCESS;

     stim_proc: PROCESS
    BEGIN
        check_value(q = '0', FAILURE, "Wrong output value at startup");
        write(d, '1', "DFF");
        check_value(q = '1', FAILURE, "Wrong output value at first test");
        write(d, '0', "DFF");
        check_value(q = '0', FAILURE, "Wrong output value at final test");
   END PROCESS;

END Behavioural;
```

B Figures

B.1 Hudson-CI example statistics

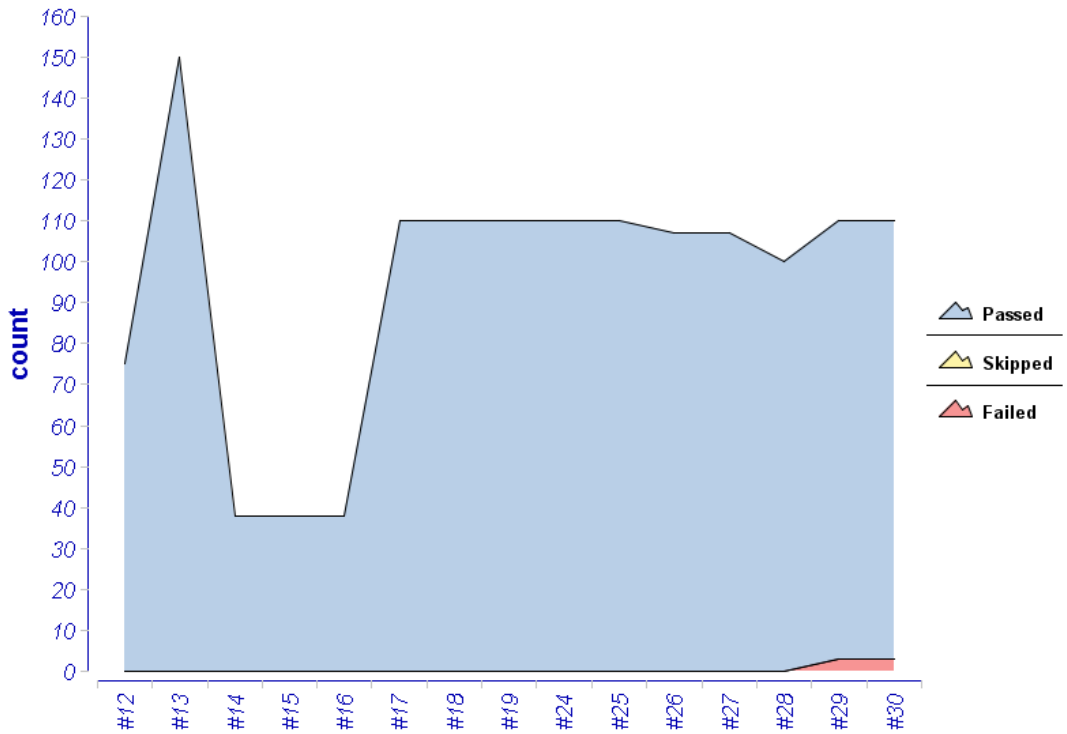

Figure 2: Hudson-CI example statistics.