FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

BACHELOR'S THESIS

Automated UVM testbench generation

Author Vili-Valtteri Ojala

Supervisor Juha Häkkinen

May 2022

Ojala V. (2022) Automated UVM testbench generation. University of Oulu, Degree Programme in Electronics and Communications Engineering. Bachelor's Thesis, 23 p.

ABSTRACT

This thesis studies the possibilities to automate UVM testbench creation in the telecommunications industry. First, the ideas behind UVM are looked at and automatable parts of the testbench coding process are studied. Facilitating the reuse of code is also examined.

Development of an automation script with python for Nokia is covered in the work, and the possibilities for future improvements are discussed.

Key words: Universal Verification Methodology, SystemVerilog, verification, code generation, UVM reuse, telecommunications, System-on-Chip.

Ojala V. (2022) Automatisoitu UVM testipenkin generointi. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 23 s.

TIIVISTELMÄ

Tämä kandidaatintyö tutkii mahdollisuuksia automatisoida UVM testipenkin kehitystä tietoliikennetekniikan saralla. Aluksi käydään läpi UVM:n taustaideat ja pohditaan automatisoitavia osia koodausprosessissa. Koodin uudelleenkäytettävyyttä tutkitaan myös tarkasti.

Työssä käydään läpi automaatioskriptin kehitys Nokialle pythonilla ja mietitään mahdollisia suuntia jatkokehitykselle.

Avainsanat: Universaali varmennusmenetelmä, SystemVerilog, koodin generointi, UVM uudelleenkäyttö, tietoliikennetekniikka, System-on-Chip.

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1 INTRODUCTION 7
2 UNIVERSAL VERIFICATION METHODOLOGY 8
2.1 SystemVerilog OOP 8
2.2 UVM Theory 9
2.3 Testbench architecture 10
2.4 Enabling code reuse 12
3 AUTOMATED TESTBENCH GENERATION 14
3.1 Previous research 14
3.2 Implementation 14
3.2.1 Filesystem creation 14
3.2.2 Parsing signal data from VHDL code 15
3.2.3 Creating UVM components 15
3.2.4 Debugging 16
3.3 Test and sequence generation 16
3.4 Development summary 17
4 DISCUSSION 18
5 SUMMARY 19
6 REFERENCES 20
7 APPENDICES 21

FOREWORD

This Bachelor’s thesis was produced at Nokia. UVM as a topic is quite challenging, but I had done some UVM work before which helped a lot in the process.

Thanks to Juha Häkkinen for supervising this thesis and to Nokia for financing the work. I also want to thank Matti Niemistö for helping in the writing process and Ashish Balanna Prabhu for giving me the topic and supporting in the work. I had a good time with writing and developing python code and studying UVM.

Oulu, May 13, 2022

Vili-Valtteri Ojala

LIST OF ABBREVIATIONS AND SYMBOLS

ASIC	Application Specific Integrated Circuit
DFE	Digital front-end
DL	Downlink
DSP	Digital signal processing
DUT	Design under test
EDA	Electronic design automation
FSM	Finite State Machine
HW	Hardware
IDE	Integrated development environment
IP	Intellectual property
JSON	JavaScript Object Notation
OOP	Object-Oriented Programming
RAL	Register abstraction layer
RTL	Register-transfer level
RX	Receiver
SoC	System on chip
SV	SystemVerilog
SW	Software
TX	Transmitter
UL	Uplink
UVM	Universal verification methodology
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VIP	Verification intellectual property

1 INTRODUCTION

In system-on-a-chip (SoC) development, the verification process to prove designs correctness takes up the most resources and time in the project. Errors in verification can lead to catastrophic increase in development time and cost. In designs, where reusable intellectual property (IP) blocks are implemented, verification takes around 70 - 80% of the total development time. Increasing complexity of SoCs makes verification even more challenging and error prone [1].

As an answer to the problem, universal verification methodology (UVM) has been developed. UVM introduces object-oriented programming (OOP) paradigms to verification and enables verification intellectual property (VIP) reuse across abstraction levels and different projects. UVM also raises the abstraction level from individual pins to transactions. UVM relies heavily on constrained random stimulus generation that has proven to be an effective method in verification [2]. However due to bad coding practices and varying architectural choices, the desired reuse of VIP is hindered, especially from IP-block to system level. In telecommunications, the SoCs are very complex, containing numerous IPs and often a processor integrated in the same chip, which emphasizes the problem.

This thesis studies the possibilities to automate and unify the UVM based functional verification process to reduce development time and increase reusability. Focus is on digital circuits in telecommunications industry. Aim is to create a tool for generating an easily reusable and understandable UVM verification environment trunk and filesystem from scratch. Study focuses on a scenario, where most of the lower-level VIP is reusable from old projects and the testbench architecture is the main problem. Emphasis is on creating an easily reusable environment template based on given input parameters. Design Under Test (DUT) top-level VHDL code file parsing will also be experimented on. Tool is implemented with python due to its advanced string manipulation capabilities and execution time not being an issue. Input parameters are given to the tool in form of a text file. A graphical user interface may be developed in the future, to make the usage easier.

Ideas behind UVM are also discussed and the testbench architectures permitting reutilization reflected on. Some parts of the developed python code will be shown in the appendices.

2 UNIVERSAL VERIFICATION METHODOLOGY

This chapter looks into UVM generally. First section goes briefly through Object-Oriented Programming (OOP) in SystemVerilog (SV), and the second section handles the theory behind UVM. Third section gives an overview of developing a testbench with UVM and in last section the architectural choices and coding styles enabling VIP reuse are looked in, with emphasis on telecommunications.

2.1 SystemVerilog OOP

OOP means, that code composes of classes that model some behavior. Instantiations of the classes are called objects. These classes can have variables and functions in them called methods. The classes can inherit properties and functionality from other classes, a feature called inheritance. Other fundamental properties of OOP are polymorphism and encapsulation [3]. Encapsulation means that some properties and implementation details are hidden from the user of the class. Polymorphism is a possibility for different classes to respond differently to the same command. These features help in raising abstraction level and increase productivity, especially in larger projects.

SV, which supports OOP, has two different function types, a task, and a function. Difference between them is that tasks can consume time, but functions cannot. Additionally, it is possible in SV to declare variables random with certain constraints [4], which is very important property for constrained random stimulus generation. For example, in the figure 1. two SV classes are created. *Long_frame* inherits variables *addr* and *payload* from *frame*. Payload is overridden, as is its constraint. *Rand* keyword means that a variable is randomizable.

```
class frame;
  rand bit payload [15:0];
  rand bit addr [3:0];

  constraint c_payload {
    payload > 2;
    payload < 65532;
  };

  constraint c_addr { addr == 0'b110; }

endclass: frame
```

```
class long_frame extends frame;

  rand bit payload [20:0];

  constraint c_payload {
    payload > 5;
    payload < 65532;
  };

  function void change_addr(int address);
    addr = address;
  endfunction : change_addr

endclass: long_frame
```

Figure 1. SV-classes.

In the figure 2. a handle is declared for each class, and classes are constructed using the default *new* constructor, that creates an instance of an object. Inside the for loop, classes are randomized, with the built in *randomize* method that takes into account the written constraints.

```
module testbench;
  m_frame frame;
  m_long_frame long_frame;

  initial begin
    m_frame = new();
    m_long_frame = new();

    for (i = 0 ; i<10 ; i++) begin
      m_frame.randomize();
      m_long_frame.randomize();
      $display("Frame address: 0x%0h Long_frame address: 0x%0h", m_frame.addr, m_long_frame.addr);
    end
  end

endmodule
```

Figure 2. SV-classes instantiated and randomized.

2.2 UVM Theory

UVM is and open-source class library provided by Accellera [5]. UVM is built on OOP principles with SV, and it models the circuit functionality in transaction level, with component reuse kept in mind. Functional coverage, which is used for monitoring the maturity of the verification work, is typically tracked in UVM by writing SV coverage groups. UVM eases test writing by enabling the reuse of the environment in different tests. Environment is just configured differently when needed. Configuration is done via configuration objects that are passed to the actual components with UVM database. There exists *uvm_config_db* and *uvm_resource_db*, but they are actually based on the same database and there is very little difference between them. In this thesis *uvm_config_db* is used. *Set* method is used for setting configurations in the database and *get* for getting. Also, a register abstraction layer (RAL) exists to ease register reads and writes by enabling access via names instead of addresses. RAL can be used with the bus protocol, which is used in the actual chip and through backdoor access methods. Backdoor access is very handy in monitoring finite state machine (FSM) status and interrupt registers. [6]

UVM handles transactions with the DUT via interfaces and agents. There can be multiple agents in one environment. An agent can contain a driver that converts the transactions into pin-level activity to the interface. Virtual interface handle is provided to the driver via the configuration database. Inside each agent containing a driver, there is also a sequencer, that delivers the executed sequences to the driver. Sequence is a component, that consists of sequence items, that model the bus transactions or other sequences. Driver and sequencer are always parametrized to drive one type of sequence item. For monitoring the stimulus and the responses the DUT provides, agent can also contain a monitor. UVM environment also usually contains a scoreboard, that collects the data from the monitors analysis port. Scoreboard can contain a checker that compares the data to the one generated by the reference model based on inputs driven to the DUT. One testbench can contain multiple environments, and environments can be created hierarchically, inside each other. [6]

In UVM, factory is used to construct all components and objects to ease overriding. Overriding means changing a handle pointing to one object to another. A standard *new* constructor method is created for every class/component. Constructors are never used in the testbench code since creation of all components is handled in factory with the *create* method. All components are registered in the factory immediately under the class declaration with a

registration macro that is defined in *uvm_pkg*, that is imported in every package in an UVM project. It is possible to override one instance of an object with *set_inst_override* and all instances with *set_type_override*. Overriding comes in handy when one test requires more strict constraints on some parameters. Then it's possible to just extend a configuration class from the old one and write the constraints with the same names, as done in a previous code sample with the frame classes.[6][5]

2.3 Testbench architecture

UVM classes are typically written in header files that are then included in package file. Package files are created for different levels, for example tests can have their own package file, which is then imported into the testbench top-level file. Environment classes, sequences and RAL usually has their own package as well. DUT signals and VIP components interfaces are declared in the top file. Signals are then connected to the interfaces and then set up in the configuration database. In the figure below, is an example of simple UVM testbench. It is important to remember, that the testbench can contain multiple environments, and each environment can contain multiple agents and other environments. Figure 3. presents a simple UVM testbench architecture with one environment containing one agent.

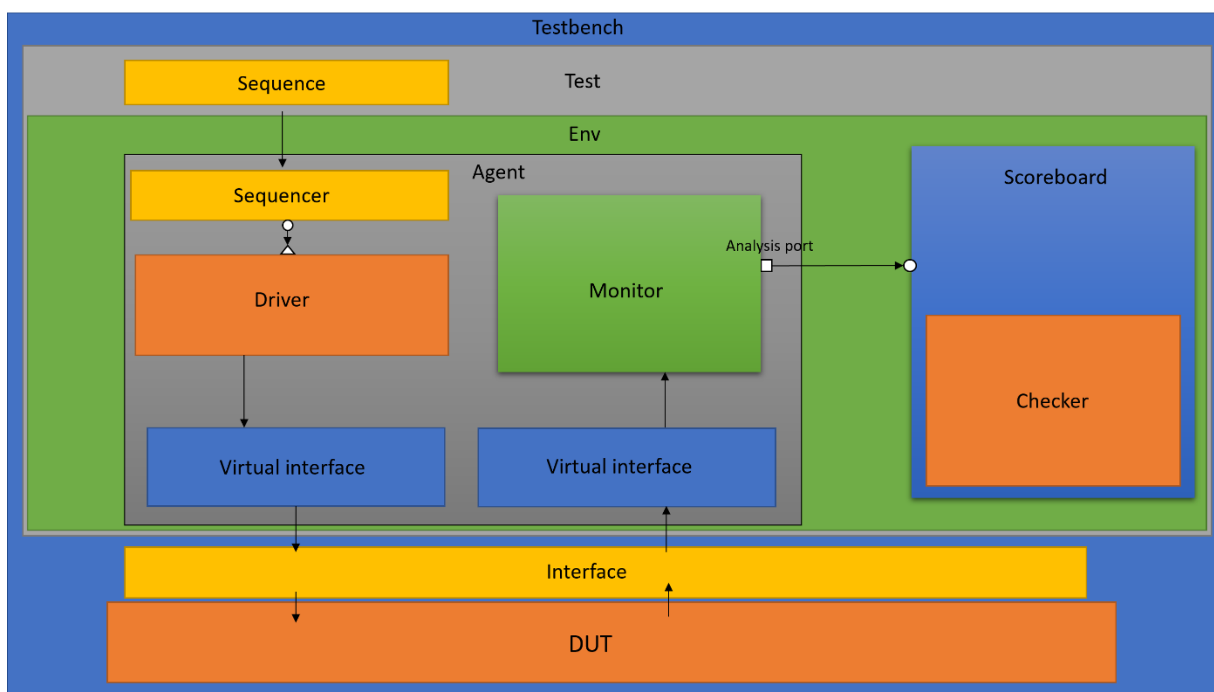

Figure 3. Simple UVM testbench architecture.

UVM uses phases to keep the timing between components in sync. UVM phases listed in the table below. Important insight is that in SV, functions cannot consume simulation time, but tasks can. Phases are always executed in the same order as in the table. It is also possible for the user to create their own phases [6]. In the code, moving to the next phase is restricted with an objection. Objection is raised in the beginning of the implemented phase inside a component. When the needed tasks are done, the objection is dropped, allowing simulation to proceed. Phases are described in table 1. [7].

Table 1. UVM phases

Name	Usage	Category	Type
build_phase	instantiating testbench components	build	function
connect_phase	connecting component's (TLM) ports	build	function
end_of_eloaboration	displaying testbench architecture and executing other functions	build	function
start_of_simulation	setting initial configuration	build	function
run_phase	Simulating the DUT. Executing sequences	run	task
extract_phase	Collecting data from scoreboard and computing expected data e.g based on golden reference model	clean	function
check_phase	Checking data integrity against the reference	clean	function
report_phase	Displaying data from checkers	clean	function
final_phase	Last operations before the simulation ends	clean	function

There are also phases that are executed parallel to the *run_phase.* These are listed in the table 2. [7]. These phases are typically used in the actual test cases instead of the *run_phase*, to split the test into more easily understandable pieces. Usage of the phases not restricted in any way. Inconsistent use of phases can introduce bugs that are hard to detect and restrain reuse of the code.

Table 2. UVM run_phase

Name	**Usage**
Pre_reset_phase	Tasks right before reset
Reset_phase	Resetting DUT
Post_reset_phase	Tasks after immediately after reset
Pre_configure_phase	Tasks before configuration
Configure_phase	Configuring DUT. E.g applying configuration parameters to registers
Post_configure_phase	Tasks after configuration
Pre_main_phase	Tasks right before main_phase
Main_phase	Launching functional sequences to the DUT
Post_main_phase	Tasks after main
Pre_shutdown_phase	Tasks before shutting the simulation down
Shutdown_phase	Shutting down the simulation
Post_shutdown_phase	After shutdown

It is important to note, that all the tasks done in the phases above can also be done in *run_phase.* There is no strict guideline whether should be used, but the ”sub-phases” are a quite recent addition to the class library [6].

2.4 Enabling code reuse

Code reuse can be achieved in two ways. As reuse between abstraction layers, here mainly meaning the reuse of VIP between IP and SoC level, and reuse between different versions of the same IP. Former is called vertical reuse and latter horizontal reuse. This thesis focuses more on the vertical reuse. For instance, a good way to achieve vertical reuse, is to instantiate the IP/subsystem-level UVM environment class in SoC-level environment. Horizontal reuse can be advanced by parametrizing the testbench as much as possible, to make scaling up or down easy. It can be done for example by declaring macros for interface sizes, component counts and bus transaction formats and then using these macros across the testbench.

In telecommunications, the downlink (DL) and uplink (UL) data paths can be verified separately. This is often the case to get quicker simulation times and ease bug spotting. In contrast to this, it makes sense to have a common structure for all verification environments. When verifying an IP, the connections buses between IPs are modeled with VIPs. However, in the SoC level, these VIPs are not needed, since the actual IPs are connected to each other. Therefore, each verification environment should have an easy way of disabling these VIPs. It also makes sense to separate the environment into downlink and uplink parts, when possible. Of course, the digital signal processing (DSP) algorithms are different in uplink and downlink and not all IPs even exist in both paths. When that is the case, the IP could be paired with another or others, when creating the UVM environment for verification. This would enable all the environments to be constructed and controlled the same way, saving time in SoC-level verification.

DUT and environment configuration is another key factor in reusability. If configuration is spread across various *phases* and components, reuse in higher level becomes very difficult. Therefore, separate configuration sequences could be used. One possibility of implementation is to create one base sequence with virtual methods and extend separate base sequences for receiver (RX) and transmitter (TX) from that. They would have the mandatory *new* constructor and a *set_config* method with arguments to the register model and to environment configuration

object. These sequences could be executed in the *run_phase* or *main_phase* in any test, both IP and SoC level. Environment configurations would be normally overridden in *end_of_elaboration,* for example. This would augment vertical reuse considerably.

3 AUTOMATED TESTBENCH GENERATION

This chapter takes a quick look into previous research and implementations on automated UVM testbench and filesystem generation and describes the process of creating a python script for that purpose. As seen in the previous chapter, UVM has many repetitive patterns and mandatory code structures and methods that can be conveniently automated.

3.1 Previous research

Automated testbench creation has been studied in some level before, and there are good results in using such solutions for register class creation in practice. Register classes include maybe the most repetitive code aside the port connections and signal declarations in the entire testbench, so it is intrinsic to automate it [8]. The step can be even combined with the register RTL code generation for the design itself. There also exists one commercial UVM testbench generator by Cadence [9] and couple of amateur open-source solutions [10]. A better solution can be developed for telecommunications SoCs, which can be thought as a DSP pipeline. This, and the knowledge of common architectures of the IPs can be exploited when developing the script.

The Cadences tool is more generally developed to match a wide range of applications. This produces a lot of configuration needs for the tool, which in turn creates overhead for the usage. Other downside with the Cadences tool is license price, upside is that it promises to create a fully functional UVM testbench. The best open-source tool is also a very general solution. However, it does not produce a fully functional testbench, only templates for components. It is easy to use with graphical user interface, but the source code is very low quality and impossible to understand without considerable efforts. Therefore, an implementation of UVM testbench generation script for telecommunications needs to be studied. The script has been made especially with the Digital Front-end (DFE) in mind.

3.2 Implementation

Next, the thinking behind the implementation of the script will be explained. This also serves as a documentation for the tool. The script is implanted with functional programming and a git repository is used for version control. To make future modifications and reuse easier, functions are kept small and parametrized as much as possible.

3.2.1 Filesystem creation

In UVM verification projects, there are typically dozens of folders, and the directory structure can be very deep and complex. Although creating folders manually is not the most challenging task, automating it comes very naturally along the code generation, since it is prominent to know the directory structure while generating files. Therefore, the first challenge in the project was to create a filesystem according to given input.

It was decided that the filesystem is described in a text file with “-“ sign meaning a transition to lower directory. Parsing the data from the input file was implemented in one python function and separate functions were created for the folder generation and UVM component folder mappings. Certain folder names expected to be found from the filesystem were hardcoded to the script. For example, the script expects the user to specify an “env” named folder. When a folder with hardcoded name is found, it’s path is added to a python dictionary, which consists

of key-value pairs. These folder paths are also written to a JavaScript Object Notation (JSON) [11] file for later use.

After a "%" sign in the input file, parameter values are given to the script. These are also saved in a dictionary and in the same JSON file. Parameters contain names and integer values, but they are all saved as strings. Numbers are converted to integers, when used. The functionality is implemented as state-machine in the script.

While generating the UVM-code, script checks that a folder is specified for a group of components. For example, while creating *uvm_env* classes and their configuration objects, the existence of "env" folder is checked from the dictionary. If the folder is not found, the script won't create any files, which are meant to be in that folder.

3.2.2 Parsing signal data from VHDL code

DUTs port widths and names were parsed from the top-level VHDL file with the script. Certain style of coding was expected from the file, e.g. certain number of spaces between *downto* keyword and signal width. These signals are then written into the top level testbench file with SV and also appended to the previously mentioned JSON file.

Clock VIP modules are instantiated, based on port names indicating a clock. Virtual interfaces are created for them and set in the *uvm_config_db.* Standard register access bus protocol VIP is connected to the DUT, and its interfaces are set up too. There is a default connection mode implemented in script, which the user can turn off in the input file. Default connection mode makes port connections between the SV and VHDL signals sharing the same name. However, this is not very handy since many times the names used in the VIPs and the VHDL code don't match. If same names were to be used, there would be greater possibilities for automation.

Polishing the VHDL parsing functionality is decided to be done in the future, since it would require very strict naming and coding rules to be really effective, but the formidable possibilities in it are definitely recognized.

3.2.3 Creating UVM components

The actual UVM components are created based on templates. Templates are written with python as multi-line strings and certain values and names are parametrized in them. In the generation, pythons string formatting is used. Some files are also written line by line based on templates consisting of just one line, or couple of lines. A standard template header is also made, that is used in every generated file. Using python *os* and *time* modules, creators name and date are written into the header. Copyright notices are also written into the header.

In the code, each group of components is created in its own function. Paths to folders and variable values and names are retrieved from the dictionary that was created when parsing the input file. For configuration objects, there is a separate function. Configuration object is automatically created for *uvm_env* components, DUT (rx, tx, and rxtx separately) and for standard bus protocol agents. Table 3. presents all generatable UVM components.

Table 3. Generatable UVM components

Name	Type
{module_name}_tbtop.sv	SystemVerilog file
{module_name}_{env}_env.svh	*uvm_env*
{module_name}_{env}_{component}_config.svh	*uvm_object*
{module_name}_{env}_sb.svh	*uvm_scoreboard*
{module_name}_{env}_sw_model.svh	*uvm_object*
{module_name}_env_pkg.sv	package file
{module_name}_{env}_test_sequence.svh	*uvm_object*
{module_name}_interface_sequence.svh	*uvm_object*
{module_name}_base_config_seq.svh	*uvm_sequence*
{module_name}_sequence_pkg.sv	package file
{module_name}_{env}_base_test.svh	*uvm_test*
{module_name}_test_pkg.sv	package file
{module_name}_ral.svh	*uvm_reg_block*
{module_name}_mem_backdoor.svh	*uvm_reg_backdoor*
{module_name}_ral.pkg	package file

3.2.4 Debugging

The generated UVM code was debugged with DVT Eclipse, and UVM aware IDE. A build file was crated and provided to the IDE for compiling the generated code. Use of the IDE eased the debugging a lot, by providing visual sight of the errors in the code. It would have been a big task to create a dummy DUT and an entire makefile to compile the generated code.

After first compilation, there was quite a lot of errors, but with some time, they were easy to spot and fix. The IDE was an extraordinary tool in debugging, since it showed the syntax errors in the code straight away in a visual manner. Build file had to be modified many times, due to many package files.

At the end, a concept for an entire makefile template was designed for the tool. It would enable the tool to create a fully functional makefile for compiling the generated code. The structure of the makefile is dependent on the directory structure and how the packages are ordered. Common VIP components also need to be addressed in it, and some macros pointing to the DUT and RAL are also usually defined in there.

3.3 Test and sequence generation

After the actual testbench is created, the next step in the project is to create testcases that test different parts of the DUTs functionality. Additional sequences for the testcases to execute also have to be created. Since there is already so much information about the project collected by the script, testcase and sequence generation are reasonable to add.

Two additional scripts were created for this purpose. Scripts take test/sequence name and the parent class as an argument. The purpose of these is to shorten development time and unify coding styles across teams, to make the code easier to understand, debug and reuse. In the tests, aim is especially to standardize phase usage. Scripts also ensure that the sequences and tests are created in the right folders. The scripts create constructors for the generated class and declare prototypes and implement the most used phases and methods in the test cases.

Constructors are fully implemented, and factory registration is also done for both. In the start of implemented phases, script adds raise objection and drop objection in the end of the phase. As an additional functionality, the scripts check that the generated file does not already exist. If it exists, the execution is terminated.

3.4 Development summary

The time used for developing the scripts was about 50 hours and 12 files were written with a total of well over a thousand lines of code. Table 4. describes all files developed during the project. A git repository was used for version control locally. If more people get involved in development in the future, a remote repository needs to be created.

Table 4. All developed files

Name	Description
uvm_generator.py	Main script. Generates the previously listed components and writes data to uvm.json file. Executed from command line, reads parameters from input.txt
make.py	Functions for uvm_generator.py
input.txt	Input parameter file with usage instructions, located in separate source folder.
uvm.json	File for storing data for later use, located in separate source folder.
testbench.py	SystemVerilog templates for uvm_generator.py located in separate templates folder
makefiles.py	Unfinished makefile template, located in separate templates folder
seq_gen.py	Script for generating an individual sequence. Executed from command line with parameters for parent sequence and a name for the generated one.
test_gen.py	Script for generating an individual test. Executed from command line with parameters for parent test and a name for the generated one
testcase.py	SystemVerilog templates for test_gen.py and seq_gen.py located in separate templates folder
__init.py__	Empty file located in the templates folder to enable importing the template files.
default.build	Build file for compiling the project in the IDE.
.gitignore	File that specifies files and folders for version control to ignore.

4 DISCUSSION

The goal of this thesis was to study the possibilities to automate UVM testbench creation. Testbench reusability was an important criteria in the research. Purpose of this was to unify the testbench architectures, decrease errors in repetitive parts of the code, unify the coding style and to reduce human labor. A script was successfully developed for that purpose. The script is able to generate a filesystem and a large quantity of the necessary UVM components and files. It was found out that UVM classes have many repetitive and even rather long parts of code that were simple to write in a template with python multi-line strings with some parameters. With that style of implementation, the parts requiring logic are more difficult to generate, since an entire file is generated at once. If even greater level of automation was desired, the templates would need to be splitted in smaller parts, even to single lines or parts of lines.

Possibilities to analyze the DUTs top-level VHDL code were also thought and experimented on. It was perceived that port names and widths can be parsed from the VHDL code and then declared in the SV top-level testbench file, which saves time, since there are usually hundreds of signals. Successfully automating the signal data parsing was a great success in this study because the data interfaces between IPs are similar, or can be standardized, in many projects.

If the VIP modules interfaces and the VHDL ports had the same names, it could be possible to determine the number of needed VIP components from the signal data. This would also open the door for automatically connecting the components and the declared signals and the whole top level testbench file, which is the longest and most repetitive in the entire testbench aside the register classes, would be fully generatable. That would save a large amount of human labor in the beginning of the project and help to avoid hardly detectable errors. Achieving that would however require much more resources than one trainee working with the tool approximately 10 hours per week. The signal naming would also require commitment across the entire project or maybe even everyone working with the SoCs. The port connection generation should be controllable from the text file, where the parameters are written by the user, in case the tool is used in the SoC level verification. There the interfaces are harder to predict.

The sequence and test generation scripts can be further improved to add more detail to the generated code by using the data collected during the initial testbench creation. It would also be possible for the scripts to include the created files in a corresponding package file. For instance, if a new sequence is generated, it would be automatically included to the package file containing all the sequences. Automated makefile creation would also save time in the verification.

This topic can be further researched, and the more detailed automation can be continued to the lower level in the UVM testbench with the signal data available. The horizontal reusability of the generated testbench can be improved by generating a parameter macro file in the same folder as the top testbench file. To serve the same purpose, an additional feature for re-parsing the VHDL code and modifying the top level testbench accordingly for a possible later version of a same IP.

To conclude, the objectives of this study were met and many ways to automate UVM testbench generation were found, although all not that easy and fast to implement. Increasing the automation in the process of functional verification is definitely achievable, given the appropriate resources, time and commitment.

5 SUMMARY

Due to SoC verification taking such a heavy toll on time and resources, new ways to speed up the process have to be studied. The automated creation of UVM testbenches used in functional verification could be one solution to the problem.

In this thesis, the automation possibilities of UVM testbench generation were studied. First, the necessary theory to understand the workings of UVM was discussed, focusing on SV OOP, on which UVM is built, and the necessary UVM components to build a testbench. Testbench architectures and coding styles allowing reuse of testbenches both horizontally and vertically were also considered.

Automation possibilities were reflected by taking a look into previous research made on the topic, and mainly by developing an automation script with python. Script utilized pythons multi-line strings and string formatting to build UVM components from templates. DUTs port connection parsing from VHDL files was also experimented on. Signal names and widths were successfully parsed from the DUTs top-level VHDL file. With the data parsed, it was possible to automate writing hundreds of lines of code into the testbench top-level file. It was noticed, that this approach has enormous potential for future development, even the majority of the testbench could be created automatically with a script, by utilizing the known interface types between IPs.

The component templates and files generated by the script were compiled to get rid of any errors made when coding up the templates. An IDE was exploited for this task, to make the debugging easier. In the end, the things learned while developing the script were discussed, and future improvement ideas presented.

6 REFERENCES

[1] B. Vineeth and B. B. Tripura Sundari, "UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol," 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018, pp. 307-310, doi: 10.1109/ICACCI.2018.8554919.

[2] N. B. Harshitha, Y. G. Praveen Kumar and M. Z. Kurian, "An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC's): A Review," 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), 2021, pp. 1710-1713, doi: 10.1109/ICAIS50930.2021.9396034.

[3] R. Tymerski, D. Li and X. Wang, "Object oriented design of a power electronics circuit simulator," [Proceedings] 1992 IEEE Workshop on Computers in Power Electronics, 1992, pp. 101-108, doi: 10.1109/CIPE.1992.247288.

[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[5] Accellera, (2014) Universal Verification Methodology (UVM) 1.2 Class Reference, Accessed 14.04.2022. URL: https://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[6] Accellera, (2015) Universal Verification Methodology (UVM) 1.2 User's Guide, Accessed 14.04.2022 URL: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[7] Chipverify, UVM-phases, Accessed 20.04.2022. URL:https://www.chipverify.com/uvm/uvm-phases

[8] Namdo Kim, Young-Nam Yun, Young-Rae Cho, J. B. Kim and Byeong Min, "How to automate millions lines of top-level UVM testbench and handle huge register classes," 2012 International SoC Design Conference (ISOCC), 2012, pp. 405-407, doi: 10.1109/ISOCC.2012.6407127.

[9] Cadence, System testbench generator, Accessed 11.05.2022 URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/system-vip/system-testbench-generator.html

[10] Github, hellovimo uvm_testbench_gen, Accessed 11.05.2022 URL: https://github.com/hellovimo/uvm_testbench_gen

[11] B. Lin, Y. Chen, X. Chen and Y. Yu, "Comparison between JSON and XML in Applications Based on AJAX," 2012 International Conference on Computer Science and Service System, 2012, pp. 1174-1177, doi: 10.1109/CSSS.2012.297.

7 APPENDICES

Appendix 1. Python samples

Appendix 1. Python samples

Hardcoded parameters

```
# parameters
OUTPUT_PATH = 'source/uvm.json'
INPUT_PATH = 'source/input.txt'
ROOT_DIR = 'verif'
LAUNCH_DIR = os.getcwd()
ENVS = ['rxtx', 'rx', 'tx']
```

```
uvm_folders = {   # paths for uvm files
  'sequences': '',
  'env': '',
  'tb': '',
  'tests': '',
  'ral': '',
  'assertions': '',
  'scripts': ''
}
```

Directory generation function

```
def create_dir(int, name):
  """
  Navigates up in the filesystem according to
  given value and then creates a directory
  cd:s to the created direcotry
  """
  if int >= 0:
    back = '..'
    path = back
    for i in range(int):
      path = os.path.join(path, back)
    os.chdir(path)
  os.mkdir(name)
  os.chdir(name)
```

Counter and parser function

```
def count_marks(mark, string):
  """
  Counts given characters in a string
  Returns: count, and string stripped of
  the chars
  """
  count = 0
  for char in string:
    if char == mark:
      count += 1
  return count, string.replace(mark, '')
```

UVM directory finder function

```
def map_uvm_dirs():
  """
  Checks if directory is UVM dir and stores the path
  """
  path = os.getcwd()
  name = path.split('/')[-1]

  if name in uvm_folders:
    uvm_folders[name] = path
    print('Adding folder: ' + name + ' to uvm_folders')
```

Part of a VHDL parser function that also declares signals in the tb

```
if 'downto' in words:
        length = words[words.index('downto') - 1]       # getting width of the
signal
        if length != '':
          bits = '[{}:0]'.format(length)
        else:
          length = 0

      dict_name[name] = length          # saving to dict

      genfile.write('  logic {width} {signal};\n'.format(
        width=bits,
        signal=name
        ))
```

Port connection function

```
def connect_ports(genfile, dict_name, module_name):
  """
  Arguments: File for writing, dictionary containing signal data and module
name
  Makes port connections to ports with the same name
  """
  genfile.write('\n  // Port connections\n  {module_name}_top #()
{module_name}_top(\n'.format(module_name=module_name))
  for i, signal in enumerate(dict_name):
    if i == len(dict_name) - 1:
      genfile.write('     .{signal}  ({signal}  )\n'.format(signal=signal))
    else:
      genfile.write('     .{signal}  ({signal}  ),\n'.format(signal=signal))
  genfile.write('  );\n')
```