FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

MASTER'S THESIS

HIGH-LEVEL VERIFICATION METHODOLOGY FOR UVMF-BASED C++ REFERENCE MODEL TESTBENCH IMPLEMENTATION

Author	Joonas Heikura
Supervisor	Jukka Lahti
Second Examiner	Juha Häkkinen
Technical Advisor	Ercan Kaygusuz

October 2021

Heikura J. (2021) High-level verification methodology for UVMF-based C++ reference model testbench implementation. University of Oulu, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 44 p.

ABSTRACT

This thesis was completed for Nokia and in cooperation with Siemens EDA. In this thesis a UVM Predictor component, which wraps a C++ reference model, was generated with UVM Framework (UVMF) and implemented. The Predictor was generated and implemented to Universal Verification Methodology (UVM) testbench that had HLS generated Design Under Test (DUT). First, the UVMF generated Predictor was implemented for the UVM testbench with a small HLS-generated design to learn the verification flow. After the first trial run, the UVMF-generated Predictor was implemented into an existing UVM testbench with a bigger subsystem as a DUT. The subsystem contained two manually written RTLs and one HLS-generated RTL.

First, this thesis presents the UVM theory and the UVM technologies that are used in the thesis work. The third chapter introduces code coverage, different coverage metrics, and the coverage metrics used in this thesis.

After theory, practical work is presented. Chapter four explains the devices under test, UVM components, testbench connections with a UVM Predictor, Predictor generation, functionality testing, and simulation. Measured coverage metrics, tools, and technologies are also presented. Finally, coverage results from thesis work with testing strategies are presented. The results of coverage closure are discussed in chapter 6, and the thesis is summarized in chapter 7.

Applying a UVMF-generated Predictor to the UVM testbench for verification flow showed promising results for obtaining a faster verification process as well as produced the possibility of using various versatile verification techniques with the Predictor, such as stimulus generation with constrained random (CR).

Keywords: HLS, RTL, UVM, Predictor, DUT, UVM Framework

Heikura J. (2021) Korkeatason verifiointi metodologian testipenkki-implementaatio UVM Framework pohjautuvalla C++ referenssi mallilla. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 44 p.

TIIVISTELMÄ

Tämä diplomityö on tehty Nokialle yhteistyössä Siemens EDA:n kanssa. Tässä diplomityössä UVM Framework työkalulla generoitiin ja toteutettiin UVM-prediktori komponentti, joka sisältää C++ referenssimallin. Generoitu prediktori integroitiin universaalin varmennusmenetelmän testipenkkiin, joka sisälsi HLS:llä luodun testattavan suunnitelman. Ensiksi UVMF:llä generoitu prediktori implementoitiin UVM-testipenkkiin pienellä HLS generoidulla alilohkolla, jotta verifiointivuo saatiin opeteltua. Ensimmäisen testivedoksen jälkeen, UVMF generoitu prediktori implementoitiin olemassa olevaan UVM-testipenkkiin, jossa varmennettavan suunnitelmana oli suurempi osajärjestelmä. Osajärjestelmä sisälsi kolme alilohkoa, joista kaksi oli manuaalisesti kirjoitettua RTL:ää ja yksi HLS generoitu RTL.

Ensiksi tässä työssä käydään läpi UVM:n teoriaa, sekä käytettävät UVM teknologiat, joita sovelletaan diplomityössä. Kolmas kappale esittelee koodin kattavuutta ja erilaisia kattavuus parametreja. Teoriaosuuden jälkeen esitellään käytännön työn asiat. Kappale 4 esittelee varmennettavat suunnitelmat, UVM komponentit, testipenkkikytkennät prediktorin kanssa, sekä prediktorin generoinnin, testauksen ja simuloinnin. Myös työssä mitattavat kattavuusparametrit, sekä käytettävät työkalut ja teknologiat esitellään. Lopuksi esitellään diplomityössä saavutetut kattavuustulokset, sekä suunnitelmien varmennusstrategiat. Diplomityössä saavutetut tulokset käydään läpi seuraavassa kappaleessa, minkä jälkeen kappaleessa 7 tiivistetään koko diplomityö. UVMF generoidun prediktorin ottaminen mukaan osaksi UVM testipenkin verifiointivuota antoi lupaavia tuloksia verifiointiprosessin nopeuttamiseksi, ja mahdollisuuden käyttää erilaisia monipuolisia verifiointitekniikoita kuten testiherätteiden luontia rajoitetun satunnaisuuden menetelmällä.

Avainsanat: HLS, RTL, UVM, Prediktori, DUT, UVM Framework

TABLE OF CONTENTS

FOREWORD

The thesis aimed to study and adopt new verification methods for HLS-generated designs. This thesis was done at Nokia in cooperation with Siemens EDA. I would like to thank Sami Leskelä and my technical advisor, Ercan Kaygusuz, for providing me with the topic and direction for my thesis. I would also like to thank all my colleagues for their general UVM support. I would like to thank Jukka Lahti for supervising this thesis as well as Juha Häkkinen for the second examination. I would also like to thank Esa-Matti Turtinen and Juho Järvinen from Siemens EDA for their general support of the project. Lastly, I would like to thank my friends, family, and fiancée for all their support.

Oulu, October 5, 2021

Joonas Heikura

LIST OF ABBREVIATIONS AND SYMBOLS

ASIC	Application Specific Integrated Circuit
API	Application Programming Interface
AVM	Advanced Verification Methodology
AXI4	Advanced Extensible Interface 4
CR	Constrained Random
CCOV	Catapult Coverage
COM	Component Object Model
DPI	Direct Programming Interface
DUT	Device Under Test
DMRS	Demodulation Reference Signal
EDA	Electronic Design Automation
FIFO	First In, First Out
FEC	Focused Expression Coverage
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HW	Hardware
HLS	High-Level Synthesis
HLV	High-Level Verification
HVL	Hardware Verification Language
HDL	Hardware Description Language
IP	Intellectual Property
I/Q	In-phase/Quadrature
I/O	Input/Output
N/A	Not/Available
OVM	Open Verification Methodology
OOP	Object-Oriented Programming
PUSCH	Physical Uplink Shared Channel
QVIP	Questa Verification IP
RTL	Register Transfer Level
RAM	Random-Access Memory
RMS	Regression Management System
SoC	System-on-Chip
SV	SystemVerilog
TB	Testbench
TLM	Transaction-Level Modelling
TLM1	Transaction-Level Modelling 1
TLM2	Transaction-Level Modelling 2
TLM GP	Transaction-Level Modelling Generic Payload
URM	Universal Reuse Methodology
UVM	Universal Verification Methodology
UVMF	Universal Verification Methodology Framework
UVMC	Universal Verification Methodology Connect
VIP	Verification Intellectual Property

VMM	Verification Methodology Manual
VM	Verification Management
VRM	Verification Run Manager

1 INTRODUCTION

Developing verification methods for making the verification process faster and more reliable is an ongoing process. Methods can vary from using different Electronic Design Automation (EDA) tools, generated designs, and how Design Under Test (DUT) is verified. High-Level Synthesis (HLS) Register Transfer Level (RTL) is now part of DUTs more than ever before. Even though manually written RTL is still part of company designs, generated RTL is replacing it bit by bit. The study [22] from the year 2015 indicated that generated RTL cuts off a few months of design time when compared to manually written Field Programmable Gate Array (FPGA) systems-on-chip (SoC). The same study showed that using generated RTL offered a little less performance with shorter design time, but with generated RTL, the verification process could already have been begun by the time the manually written RTL is ready. [22, p.5] Coverage closure is an important target when verifying FPGA or Application Specific Integrated Circuit (ASIC). Coverage closure measures the design to verify all its features and functions. One important aspect when simulating the DUT is to have the testbench self-checking that the output of the DUT matches with expected data. One way to do this is using a reference model (e.g., Matlab) to generate reference data files, which are then used for automated comparison.

One potential method to enhance the verification process is to replace the traditional reference file comparison approach for the reference model that is integrated into the testbench as a Predictor component. A predictor is a Universal Verification Methodology (UVM) component that is used to calculate the expected outputs of the design [4, p.7]. The Predictor can be manually written with SystemVerilog (SV), but it takes time away from other verification tasks, which makes it so that it can also be generated by HLS tools using a reference model of the design as an input [4, p.129]. When verifying complex SoC systems, writing reference models manually can be difficult and time-consuming work. There is also an increased risk of errors in large designs because while writing a large amount of code, engineers are likely to include bugs. That is why moving to a generated reference model or design saves time and there is a better chance for it to work properly both in the verification and the product.

High-Level Verification (HLV) flow approaches HLS RTL coverage in verification from a higher level of abstraction by covering part of the verification at the C++ level. When code at the C++ level is well-covered, repeating the same tests at the RTL level will result in high code coverage as a starting point and the focus in the verification is on the RTL components added by the HLS process. In many cases, generated HLS RTL is not easily human-readable, and the coverage gaps become more understandable at the C++ level on a higher level of abstraction.

This thesis was completed to study an enhanced verification flow that uses a UVM Framework (UVMF) generated Predictor for the possibility of replacing Matlab reference files as golden data and the possible difficulties with HLS-generated RTL over manually written RTL in coverage closure [20]. The UVM testbench (TB) for small designs was created from scratch to prototype and to learn the setup. This flow was repeated twice: first with a small design to learn the flow and then for a bigger subsystem. The goal was to develop an enhanced verification flow or methodology for HLS RTL when moving from Matlab-generated reference files to a generated reference model.

In this thesis, a UVM TB is created and a C++ reference model is wrapped to a UVM Predictor component generated with the UVMF. The generated Predictor is then integrated into the written UVM testbench, and coverage closure is performed for HLS-generated RTL.

The second chapter presents the theory for the evolution of UVM, UVM phases, and the UVM technologies that are adapted for this thesis. The third chapter presents coverage metrics. The

fourth chapter presents the practical work as it describes the used DUTs, UVM components, testbench connections with the Predictor, Predictor generation, testing, and simulation of the Predictor. Used coverage metrics, tools, and technologies are also discussed. The fifth chapter presents RTL coverage closure flow, the used testing strategies for coverage closure, and the results. The sixth chapter contains the result analysis, and the seventh chapter summarizes the whole thesis.

2 UVM VERIFICATION WITH SYSTEMVERILOG

2.1 Universal Verification Methodology

Universal verification methodology (UVM) is the first-ever verification methodology to be standardized by the IEEE Standards Association, and it is developed by Accellera. Figure 1 shows the overall development of the UVM methodology. Standardization was possible because of the work that was done between EDA vendors and customers. UVM is a combination of technologies and knowledge that is inherited from Mentor's Advanced Verification Methodology (AVM), Mentor and Cadence's Open Verification Methodology (OVM), Verisity's eReuse Methodology, and Synopsys Verification Methodology Manual-Register Abstraction Layer. [4, p.7] OVM is an open-source verification methodology containing base classes written in SystemVerilog. It is inherited from Mentor's AVM and Cadence's Universal Reuse Methodology (URM). OVM has features of integrating component architectures, Transaction-level modeling with stimulus transactions, phasing of testbench execution, and Object-Oriented Programming (OOP). [4, p.538, 7]

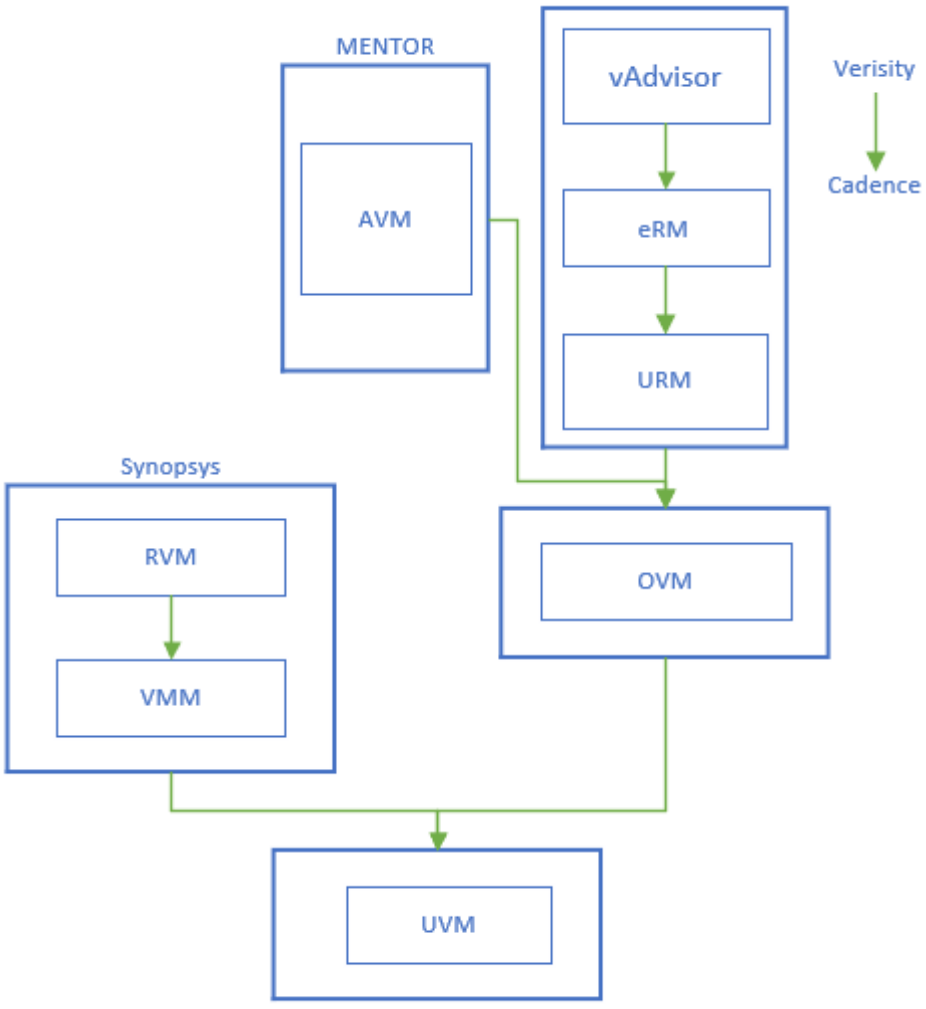

Figure 1. Development of UVM methodology.

UVM uses OVM as its base code, which contains the technology for OOP and combines it with Synopsys's VMM, as shown in figure 1. UVM has a flexible way of executing testbench phases and using and creating UVM components and testbenches. UVM TBs are written with hardware description languages (HDL), such as Verilog or SystemVerilog. Between these two HDLs, Verilog does not include OOP features like SystemVerilog. [4]

2.1.1 UVM phasing

The testbench execution flow is divided into three main phases to keep execution straightforward. The phases are build phases, run phases, and cleanup phases, and they are also executed in that sequence. [4, p.17] At build phases, TB and all UVM components are configured, constructed, and connected before the beginning of the simulation. Simulation time is not consumed at build phases due to its methods being functions. Run phases start the stimulus generation execution of the test case. Simulation time starts running and all phases from start_of_simulation to post_shutdown are executed. Phases executed during run phases can be seen in figure 2. Mostly just reset, configure, main, and shutdown phases are necessary to implement on a testbench, but their post- and pre-phases can be implemented if needed. The last main phases to be executed are the cleanup phases. During cleanup phases, information from the testbench scoreboards and monitors is gathered and reported. The collected data contains the details of whether the test case is passed or failed, as well as what kind of coverage results are gained during the process. [4, p.17–20]

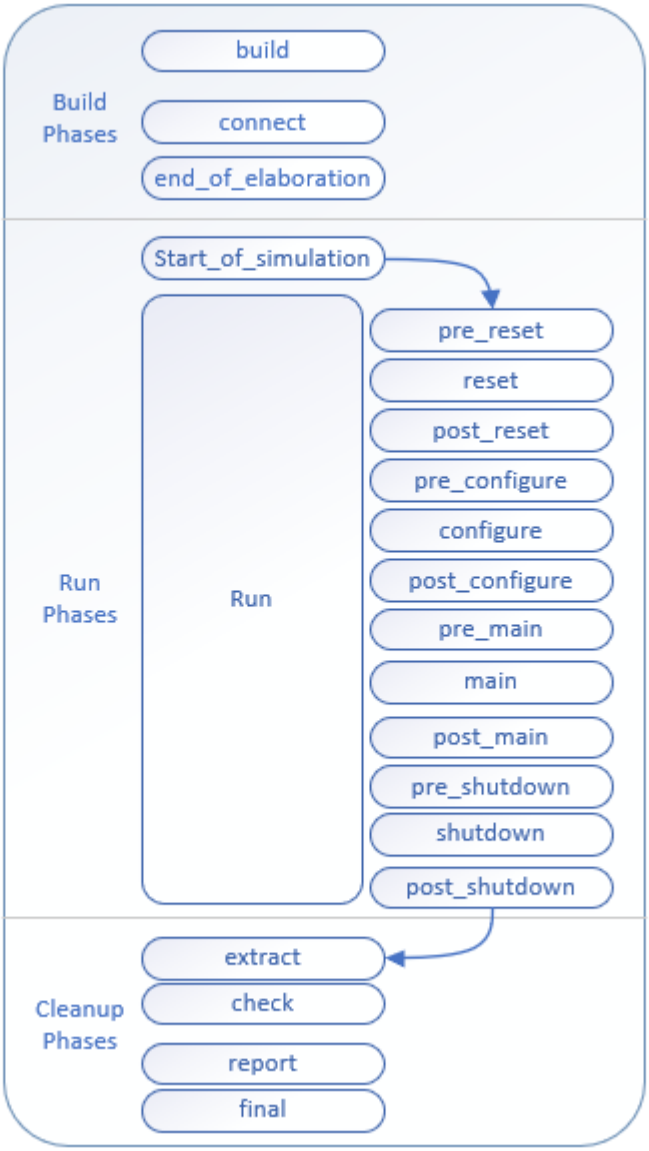

Figure 2. UVM Phases.

Starting the execution of UVM phases is done by calling the run_test() method that is normally implemented and called in the top level of the testbench. [4, p.18] Phases that are executed during the calling of the run_test() method are shown in figure 2.

2.1.2 UVM Testbench Architecture

Figure 3 represents the typical UVM testbench architecture, and figure 4 represents the testbench hierarchy used in this thesis. The highest level of a testbench is the testbench top where other UVM components are then constructed. Tb_top contains an initial block where the run_test() method is called and test execution is begun. [4, p.10] The top-level UVM component test encapsulates the environment, configures it, and provides a stimulus from the environment to the DUT. The testbench has one base test, and a TB can contain separate tests with various stimuli, which are all extended from that base test. The environment instantiates and configures other testbench components that are connected with a DUT, such as scoreboards and agents.

Usually, there is only one environment instantiated, but it is possible to have as many as preferred in parallel. [6, p.8]

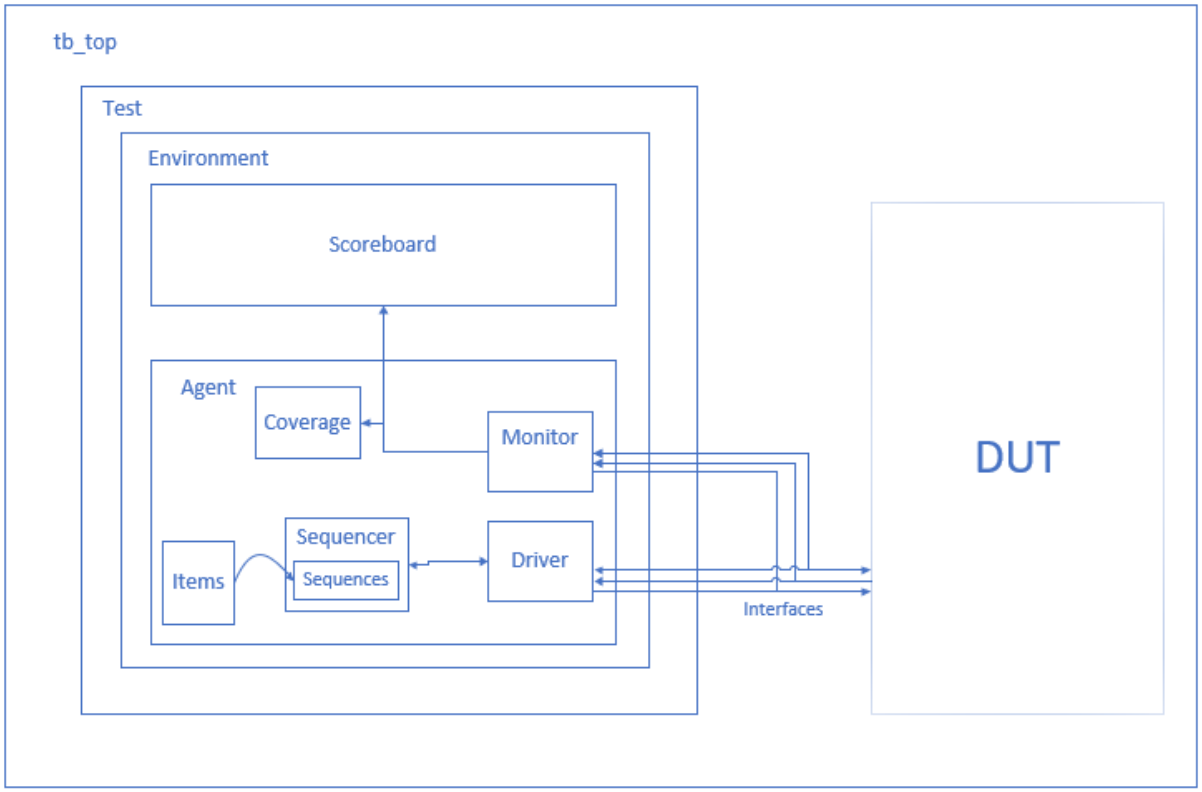

Figure 3. Typical UVM testbench hierarchy.

UVM scoreboards are UVM components whose task is to check if the design is working properly. The scoreboard takes transactions from the DUT's outputs and reference data from a reference model or reference files and compares them to confirm that the design works properly. The scoreboard is connected to the UVM Agent's Monitor. [6, p.8] The UVM Agent configures and instantiates UVM components and connects them together. The components inside a UVM Agent are the sequencer, driver, and monitor. These three components are connected to each other and the DUT's interfaces. There may also be other components inside a UVM Agent, such as coverage collectors or Transaction-Level Modelling (TLM) models. UVM Agents can be configured to be passive or active. An active Agent has all three components mentioned above and their transactions are driven. However, passive Agents contain only the Monitor component, so no stimulus driving happens. [4, p.25–26, p.28, 6, p.8–9]
The UVM Sequencer handles transaction flow and sends transaction items created by UVM Sequences to the UVM Driver. There can be several UVM Sequences handled by just one Sequencer. The UVM Sequences create the stimulus for the Sequencer and control how many data items the sent package is holding. The transaction from a UVM Sequencer is collected by a UVM Driver which then forwards the transaction towards the DUT's interface and the DUT itself. The DUT's interface is also connected to a UVM Monitor which observes the DUT's behavior at the transaction level. The Monitor is a passive component, so it only monitors the DUT's behavior and does not drive anything. [4, p.22, p.32, 6, p.9] The testbench hierarchy shown in figure 4 was used in this thesis work.

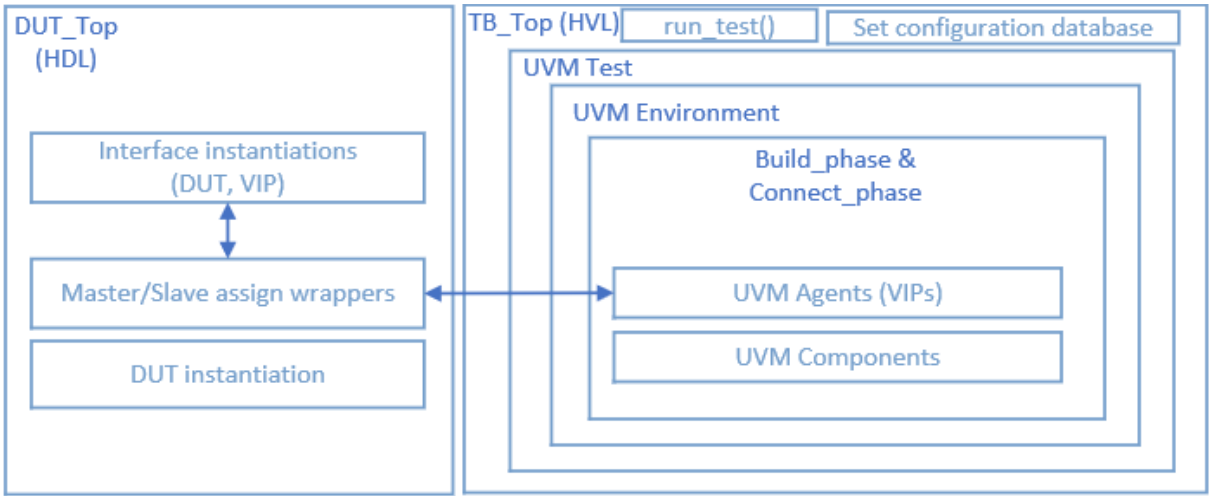

Figure 4. The testbench hierarchy used in thesis work.

The testbench adopted a dual top testbench architecture with a Hardware Description Language (HDL) and Hardware Verification Language (HVL) top. DUT_Top instantiates the used interfaces, Master and Slave side assigns to the interface signals and DUT instantiation. All integrations to HDL_Top are associated with the clock signal level activity of the DUT. The TB_Top wraps the OOP testbench with used Agents and UVM components. The run_test() method is instantiated inside the initial block which calls the method and starts the execution of the test. Also, all Agent and interface database configurations are set in HVL_Top to the configuration database. [26]

2.2 UVM Technologies

2.2.1 UVM Connect

UVM Connect (UVMC) is an open-source library that makes it possible to perform TLM transactions and object passing between foreign languages and SystemVerilog modules, TLM1 and TLM2 connectivity, and components. The supported programming languages are C, C++, and SystemC.
UVMC offers IP and VIP reusability for reusing stimulus generation VIPs in SystemVerilog to verify SystemC models. It is also a possible to access and control UVM simulation from SystemC through the UVM Command Application Programming Interface (API). With API one can control, for example, UVM transitions, prints, and UVM factory overrides. UVMC has a built-in feature for the TLM generic payload (TLM GP). [4, p. 433–434]

2.2.2 UVM Framework

Fast UVM testbench implementation offers many possibilities for verification. Users may also want different variants of the designed chips. Having a separate testbench for each DUT is time-consuming, which is why the UVM framework is a great solution for faster UVM TB development. This thesis generated only the Predictor UVM component, while other parts of the testbench and components were disabled from generation. UVMF enables recyclable UVM methodology and rapid automated UVM TB generation. [1, 2, p.8] It can automatically generate a complete TB architecture with interconnectivity for interface packages, agents, scoreboards, and TLM ports by using SystemVerilog as the output Hardware Description Language.

The first part of UVMF flow is to design and picture the UVM TB that needs to be verified and measured. After TB planning has been made, all information is put into a configuration file, such as a YAML file or a Python API-based file. YAML is a data serialization language which the UVMF generator uses as an input to generate testbench environments and Predictors with a shell script. [20] In this thesis, the YAML file was used to define Predictor, TLM ports, pinpoint paths to different submodules, and C++ model files. Figure 5 depicts an example of a YAML file structure used in this thesis, where used parameters, TLM ports, and design files needed to be declared. The configuration file is then fed into a script that generates a TB using UVM Framework. [20]

```
uvmf:
  util_components:
    "leka_rx_pusch_dmrs_predictor" :
      type: "tlm2_sysc_predictor"
      parameters:
      - name: "s_axis_dynamic_control_rsc_WIDTH"
        type: "int"
      - name: "m_axis_dynamic_control_rsc_WIDTH"
        type: "int"
      - name: "m_axis_selected_ri_rsc_WIDTH"
        type: "int"
      analysis_exports:
      - name: "s_axis_dynamic_control_rsc_ae"
        type: "ccs_transaction#(.WIDTH(s_axis_dynamic_control_rsc_WIDTH))"
      - name: "s_axis_dmrs_rsc_ae"
        type: "ccs_transaction#(.WIDTH(s_axis_dmrs_rsc_WIDTH))"
      analysis_ports:
      - name: "m_axis_agcShift_rsc_ap"
        type: "ccs_transaction#(.WIDTH(m_axis_agcShift_rsc_WIDTH))"
  environments:
    "leka_rx_pusch_dmrs" :
      parameters:
      - name: "s_axis_dynamic_control_rsc_WIDTH"
        type: "int"
        value: "8"
      imports:
      - name: "ccs_pkg"
      config_constraints: []
      config_vars: []
      config_variable_values: []
      analysis_exports:
      - name: "dmrs_dynamic_control_ae"
        trans_type: "ccs_transaction#(.WIDTH(m_axis_dynamic_control_rsc_WIDTH))"
        connected_to: "leka_rx_pusch_dmrs_pred.s_axis_dynamic_control_rsc_ae"
      - name: "dmrs_data_rsc_ae"
        trans_type: "ccs_transaction#(.WIDTH(s_axis_dmrs_rsc_WIDTH))"
        connected_to: "leka_rx_pusch_dmrs_pred.s_axis_dmrs_rsc_ae"
      analysis_ports:
      - name: "dmrs_m_agcShift_rsc_ap"
        trans_type: "ccs_transaction#(.WIDTH(m_axis_agcShift_rsc_WIDTH))"
        connected_to: "leka_rx_pusch_dmrs_pred.m_axis_agcShift_rsc_ap"
      analysis_components:
      - name: "leka_rx_pusch_dmrs_pred"
        type: "leka_rx_pusch_dmrs_predictor"
```

Figure 5. YAML file syntax structure.

Several things in the YAML file are non-mandatory, so the user can define what kind of environment and TB to generate. [2] There are also many integrated tools and technologies that

come with UVMF. They are shown in Table 1, which is also where the purpose of the tool or technology is explained. [2] In this thesis, a Visualizer simulator as well as a Regression Management System (RMS) based on Siemens EDA's Verification Run Manager (VRM) system were used.

Table 1. Integrated tools and technologies inside UVMF

TOOL/TECHNOLOGY	PURPOSE
Questa Verification IP (QVIP)	IDE instantiates, configures, and connecting QVIP components
inFact	UVM/SV importer that accelerates coverage closure
Verification Management (VM)	Coverage data and metrics ranking, merging, analyzing, and reporting
Verification Run Manager (VRM)	Regression testing optimization, integration into Jenkins ecosystem
Vista	Hybrid SystemC/SV that accelerates SoC simulation/emulation
Visualizer	Advanced design simulator and reconstruction technology increasing debug speed
Veloce	Hardware (HW) assisted TBX verification for reusing UVM test benches

2.2.3 Direct Programming Interface

A direct programming interface (DPI) is an interface that is situated between the foreign programming language layer, usually C or C++, and the SystemVerilog layer that is the surface of the UVM. DPIs on both sides are isolated. [13, p. 939] A DPI can be compared to a black box: the specifications and implementations of components are fully isolated and are transparent to the rest of the system. Because of this, the instantiation of programming languages is transparent as well. The isolation between the foreign language and SystemVerilog is performed by functions. These functions can be instantiated into the SystemVerilog side or the foreign language side and called from either side to another. [13, p. 939]
DPI offers an easy way to use function calls between different programming languages and to move data between programming layers. Functions that are called from the SystemVerilog side are called imported functions, and function calls happening on the foreign side are called export functions. Task calls between languages are also possible. Imported tasks called from SystemVerilog act in the same way as in standard SystemVerilog tasks, so they never return any value and can consume the run time of the simulation. In DPI all functions are supposed to finish their execution immediately and consume zero simulation run time. [13, p. 939-940] DPI can be written either manually or automatically by the tool.

2.2.4 SystemC

The complexity and sizes of modeled hardware and software are growing, which is where SystemC comes into the picture. SystemC is an open-source C++ class library that was made for hardware modeling purposes. It can model complex hardware and software on many levels

of abstraction. SystemC supports all C++ data types as well as some that are native only to SystemC. [14, p.8, p.388]

3 CODE COVERAGE AND METRICS

3.1 Coverage

Coverage measures the design during the stimulus and communicates which part of the code is executed and completed during the simulation. Coverage can be split into implicit coverage and explicit coverage, as shown in figure 6. An example of explicit coverage is functional coverage. Functional coverage is a manually written and defined coverage model in which the metrics and features that need to be covered are usually discussed with the designer. [9, p.8] Coverage measurement can also be automatically built by a tool, which is called an implicit coverage metric. Measuring coverage in a testbench is an automatic process that collects data from the DUT during the stimulus. All design features and requirements that need to be verified are documented in the test plan. [9, p.6–8]

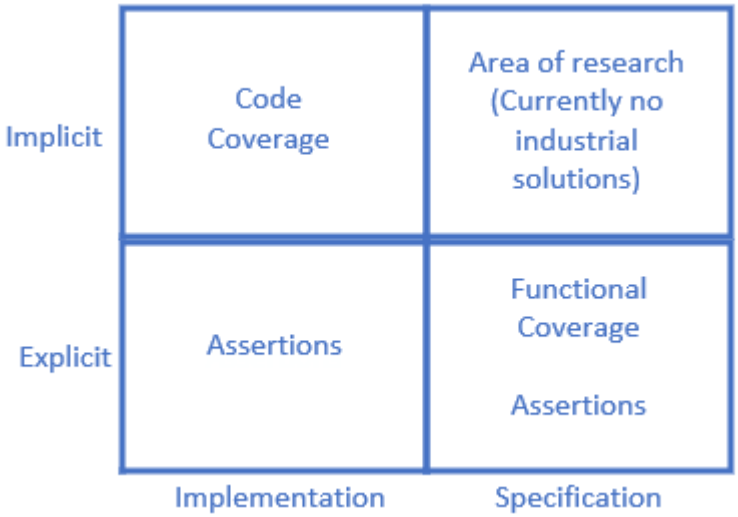

Figure 6. Coverage metrics categories.

The target for code coverage is always 100%, but this is often not possible due to design limitations or how the IP is designed and how it works together with other IPs. This might leave some code coverage gaps that are reviewed together with design and verification teams to decide which gaps are acceptable. There are eight different code coverage metrics that can be measured: toggle coverage, line coverage, statement coverage, condition coverage, branch coverage, expression coverage, focused expression coverage (FEC), and finite-state machine coverage. [9, p.12]

3.1.1 Line Coverage

Line coverage measures all source code lines that do and do not execute during the simulation of the design, including how many times they are executed. Measuring line coverage is a good way to find bugs, lines that need another test to be covered, and unused or dead code. Dead code or unused code is a part of code that is not executed during simulation or supported in the current IP and which show as gaps in coverage. Line coverage helps a verification engineer to decide if the minimum line execution threshold has been achieved. [9, p.11]

3.1.2 Statement Coverage

Statement coverage measures all statements that are reached during the simulation and how many times they are executed. It is similar to line coverage, but many verification engineers prefer statement coverage over line coverage. This is because statement coverage can analyze several statements in a single line of the code. This is one of the most basic coverage metrics that are measured. All statement lines are analyzed independently. [9, p.11, 10, p.927]

3.1.3 Branch Coverage

The branch coverage metric measures different statement branches as if–or cases. The results from all true and false statements that are evaluated from the Boolean expression table are reported. Even states that cannot happen are measured, so the results need to be discussed with the designer of the IP. [9, p.11]

3.1.4 Condition Coverage

The condition coverage metric measures if and ternary statements. [11, p.894] Ternary statements contain the ternary operator ?: which is a condition operator in the SystemVerilog language. The ternary operator has multiple expressions which are chosen by the first expression executed. If the first expression condition is 1, then the second expression is executed, but if the first expression condition is 0, then the third expression condition is executed. The ternary operator is usually used with multiplexers because multiplexers have multiple inputs and with the ternary operator it is easy to choose which input is in use. [12, p.206] Condition does not have any logical operators, so it is a Boolean operand. [9, p.12]

3.1.5 Expression Coverage

Expression coverage is close to condition coverage, and it also measures conditions that are evaluated as true or false. It uses a Boolean expression table and checks each expression that could happen. [9, p.12] Expression analyzes all assignment statements that are executed during simulation. [11, p.894]

3.1.6 Focused Expression Coverage

The focused expression coverage (FEC) metric measures all inputs. When input expressions and conditions hit 100%, then that input is fully covered. FEC is used by the DO-178B and DO-254 standards. DO-178B is a standard for safety-critical software certification. DO-254 is a standard for formal airborne electronic hardware certification. [9, p.12, 10, p. 939]

3.1.7 Toggle Coverage

Toggle coverage is a metric that measures which bits of a signal are registered or hit. There are different requirements depending on the signals and projects. Sometimes some signal bits are not registered because those signal bits are not in use on a higher-level testbench. Analyzing toggle coverage may be more harmful than helpful when there is little knowledge of which bits should toggle. However, this can usually be discussed with the designer. This is why toggle

coverage can sometimes be left unanalyzed if other metrics add more value. Toggle coverage can help connectivity checking between IP blocks. [9, p.11]

3.1.8 FSM Coverage

The finite state machine (FSM) coverage metric measures all states and transitions that occur during simulation. All states that happen in state machines and all transitions from one state to another are measured and included in the final results. [9, p.12, 10, p.1023]

3.1.9 Functional Coverage

Functional coverage is a model that covers different design features or specifications that are defined by the designer. If some feature is determined as a functionality in the design with a functional coverage model, a verification engineer can verify that the design supports these features. Functional coverage is explicit coverage, so the model is not automatically generated and requires manual implementation by a verification engineer. Functional coverage is divided into two main types of how to measure it: cover groups and cover properties. [9, p.14]
A verification engineer cannot implement cover groups and properties blindly. The engineer also needs to know when the measurement trigger should happen. Cover groups contain values from interfaces, registers, and buses that are under observation during cover group sampling. Sequences that occur during RTL simulation are measured with cover properties. This is a good way to verify block functions. The most effective way to measure functional coverage is to use groups and properties together. [9, p.15]

4 TESTBENCH IMPLEMENTATION WITH A UVMF-BASED C++ REFERENCE MODEL

The goal of this thesis was to build a UVM TB for a smaller design, wrap a C++ reference model to a UVM Predictor component automatically generated by UVMF and integrate it with a TB, and do coverage closure for an HLS-generated RTL. Predictor implementation was done with two different IPs. First with a smaller design to test and learn new verification methodology, and then by implementing the reference model to a much larger subsystem called DMRS. The smaller design is named Submodule0. The DMRS DUT contained three subblocks: HLS_Block, Manual_RTL1, and Manual_RTL2. The UVM testbench for Submodule0 was performed by using Nokia's verification platform. The platform acts as a skeleton for UVM TB to enable the faster implementation of the testbench.

4.1 Designs Under Test

4.1.1 Submodule0

Submodule0 was the first design to be verified as a DUT and to learn UVMF base Predictor generation and integration. It is a submodule and part of a much bigger subsystem called Physical Uplink Shared Channel (PUSCH) and is a fully HLS-generated design. Figure 7 shows the block diagram of Submodule0. Submodule0 has two data input and output ports. One for streaming I/Q data and the other channel for streaming control data. Each input and output had VLD and RDY handshake interfaces in order to let the design know when to start or stop feeding data.

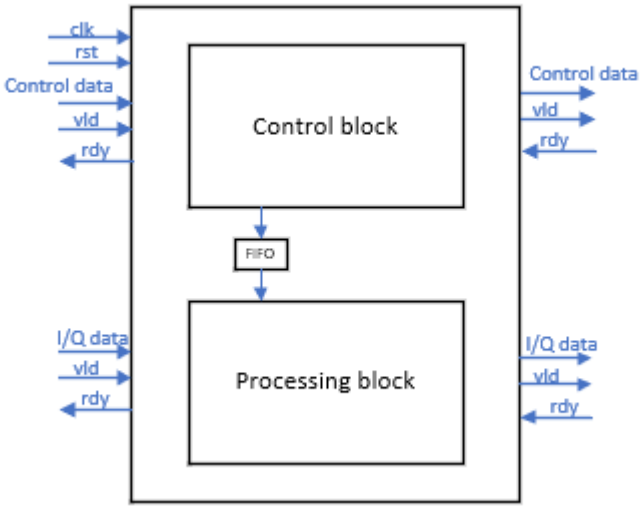

Figure 7. Block diagram of Submodule0, consisting of control and processing blocks.

Control and I/Q data are fed through the block with Advanced Extensible Interface 4 (AXI4)-Stream type interfaces. Data feeding starts by feeding the first set of dynamic configurations into the block. Used configurations can be seen in table 2. Submodule0 is a frame-based design, so it takes one set of I/O configuration parameters and the right amount of I/Q data. After the first configurations come from the block, the amount of fed I/Q data samples are calculated from the configurations and fed through the processing block of Submodule0.

4.1.2 Demodulation Reference Signal

The second DUT to be verified was the Demodulation Reference Signal (DMRS), which is also a frame-based design, so it takes a set of configuration data with the right amount of I/Q data. Figure 8 shows the block structure of the DMRS and its subblocks. DMRS forms from three subblocks where two are manually written RTLs and the third is an HLS-generated RTL. DMRS writes outputs to buffers and another output block.

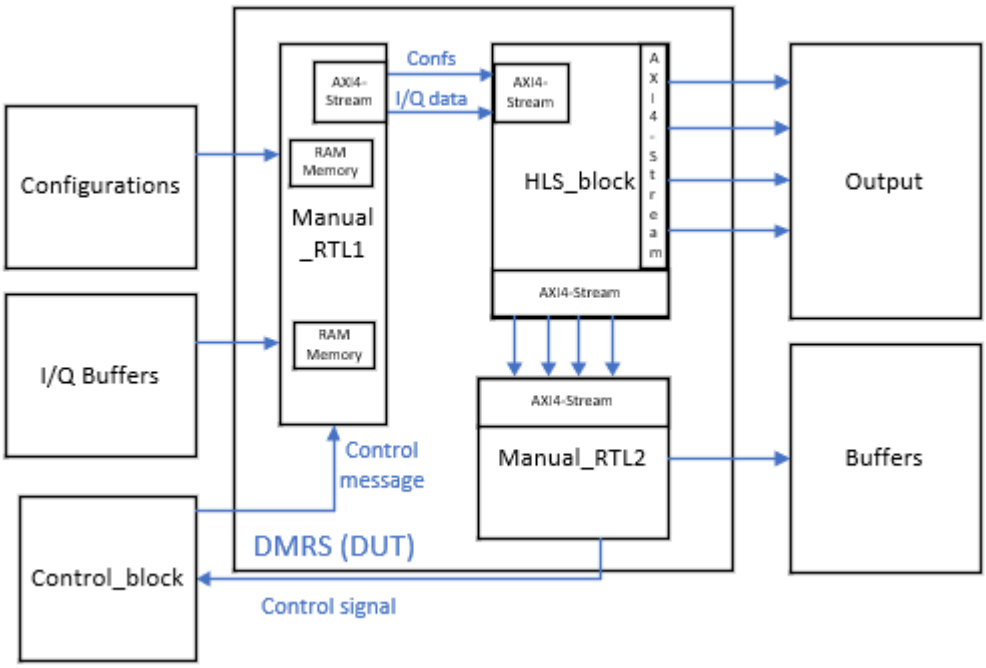

Figure 8. The DMRS DUT and its subblocks.

Configurations and I/Q data are written to the DMRS block with memory interfaces to Random-access memory (RAM) blocks, which are inside Manual_RTL1. The AXI4-Stream bus is in use for all internal interfaces. Manual_RTL1 sends collected configurations and I/Q data through the AXI4-Stream interface to the HLS-generated design. HLS_block processes data, and output is collected to the measurement block and forwarded to Manual_RTL2. Manual_RTL2 streams data to buffers and sends a control signal to control_block. Control_block sends a message to Manual_RTL1. The message contains information about the form of data.

4.2 UVM Components

In this chapter, the UVM components used in the practical work are introduced.

4.2.1 Transformers

The UVM TB needed transformers because the instantiated Predictor component used a different datatype than the testbench itself. Transformers were instantiated as UVM components, extending from the uvm_subscriber class. Transformers transform one type of data into another. Figure 9 shows the example code of the transformer component. In that example, Transformer takes a ccs_transaction data type input from its analysis_port and assigns it to a data item that is an AXI data type. Transformer components can be made reusable because they

can be parameterized. In this thesis, data widths of the incoming data are parametrized. Because data width is parametrized, an unlimited number of transformers can be created with different data widths.

```
2 ///Parametrized transformer from Predictor to Scoreboard.
3 class Transformer_predictor_to_scoreboard#(int WIDTH, type T = ccs_transaction#(WIDTH)) extends uvm_subscriber#(T);
4   `uvm_component_param_utils(Transformer_predictor_to_scoreboard#(WIDTH, T))
5
6   //Declaring analysis_port and data objects where data is written.
7   typedef axi_stream_data_logic_seq_item predictor_to_sb_item;
8   predictor_to_sb_item to_sb_item;
9   uvm_analysis_port#(axi_stream_data_logic_seq_item) d_to_sb_ap;
10
11    //Constructing data object.
12  function new(string name, uvm_component parent);
13    super.new(name, parent);
14
15    d_to_sb_ap    = new("d_to_sb_ap",this);
16
17  endfunction
18
19 ///Building data item.
20  function void build_phase(uvm_phase phase);
21    super.build_phase(phase);
22
23    to_sb_item = axi_stream_data_logic_seq_item::type_id::create("to_sb_item");
24
25  endfunction
26
27  ///uvm_subscriber class write function where data is collected to object and forwarded to analysis_port.
28  function void write(T t);
29
30       to_sb_item.valid = new[1];
31       to_sb_item.data = new[1];
32
33       to_sb_item.data[0][`MAX_DW-1:0] = 0;
34       to_sb_item.valid = '{1{1}};
35       to_sb_item.data[0][WIDTH-1:0] = t.rtl_data;
36       d_to_sb_ap.write(to_sb_item);
37
38  endfunction
39 endclass : Transformer_predictor_to_scoreboard
```

Figure 9. A transformer component extending from the uvm_subscriber class.

Each input and output stream needs a Transformer. The transformer used analysis_port, which is part of the uvm_subscriber class, to collect input or output data, which is then collected from the write function and forwarded to another analysis_port that is declared into the transformer's output side. The transformer class contains a construction function where data objects are constructed with a new build_phase function for building data items as well as a write function, which obtains data from the subscriber's analysis_port and assigns data to another object, which is a different type of data, and forwards data to the output side analysis_port.

4.2.2 Predictor

A predictor predicts the output value of the verified DUT. It is a verification component that is used to model reference data to the scoreboard where the output of the DUT is compared to the expected data coming from the predictor. Predictors can have several input and output data streams depending on the HLS part of the design. The predictor takes the same observed data that is going to the DUT and transforms it into the expected data from the predictor's output. A predictor is usually part of the scoreboard, but it can also be its own UVM component. As all verification components, a predictor has TLM ports that are used to connect the predictor to the testbench. Data is streamed to these analysis ports and fed through the predictor.
Predictors are usually written in a foreign language such as C, C++, or SystemC, but they can also be written with SystemVerilog. The predictor can also act as a proxy DUT. Usually, a predictor is written at a higher level of abstraction which is why writing data through a predictor and the availability of data takes less time than a DUT. [4, p. 129]

The reference model in this thesis work is the UVMF-generated UVM Predictor component which contains a wrapped C++ reference model. The wrapped C++ reference model only contained the HLS part of the DUT. Figure 10 presents the Predictor hierarchy and its layers when it is fully generated by the UVMF.

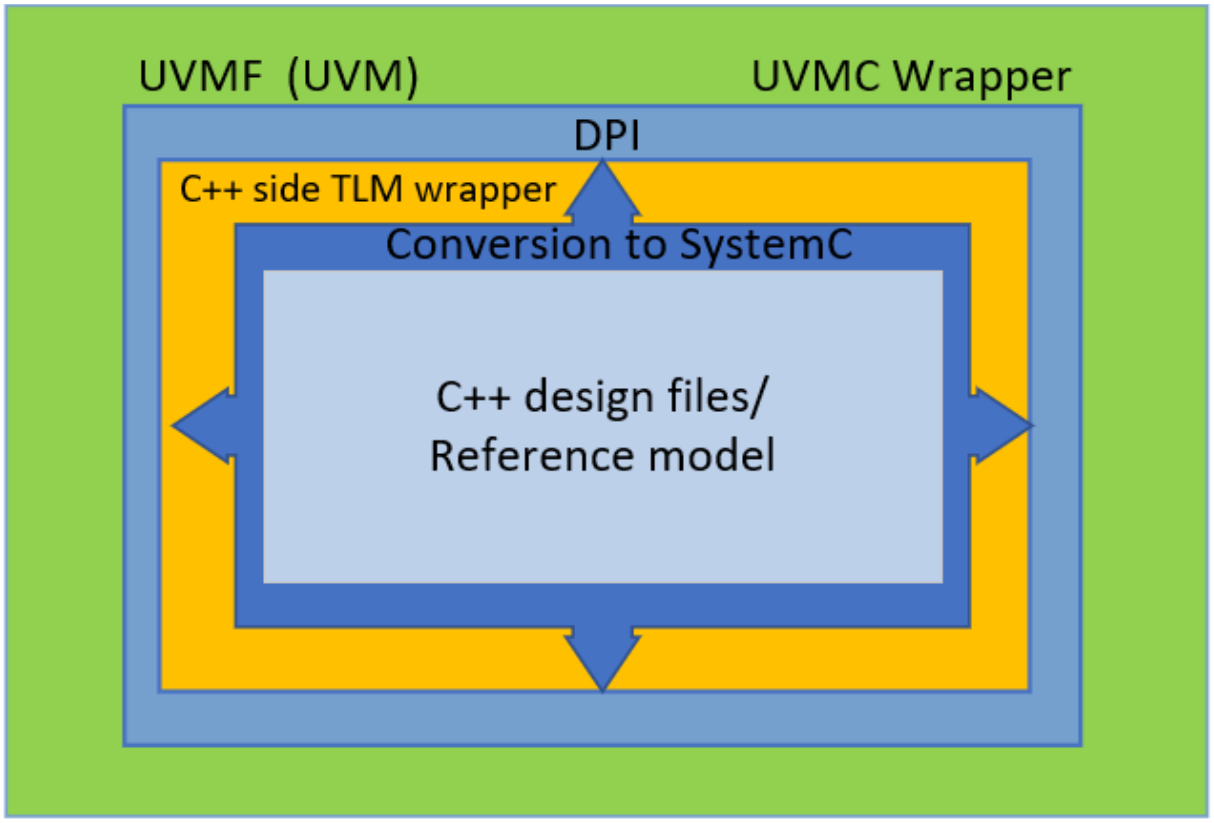

Figure 10. Predictor hierarchy.

The Predictor has four different layers. The lowest layer and its core are C++ design files that are wrapped inside the Predictor. The C++ reference model models the behavior of the DUT. C++ is then automatically transformed into SystemC form. Between the C++ side TLM wrapper and the SystemVerilog layer is the DPI. DPI allows C function calls from the SystemVerilog side, which makes it so the foreign-SystemVerilog line can be crossed. The highest layer is the UVMF layer which itself is UVM code. The UVMF layer has a wrapper that uses the UVMC library that includes function calls for TLM packet transactions from the SystemVerilog-SystemC interface. These transactions are sent through TLM1 and TLM2 connectivity and Analysis Ports. [23]

The Predictor has as many input and output ports as the DUT, so Submodule0's predictor had one data and control data input port as well as two output ports.

The DMRS Predictor had two inputs for data and control as well as several outputs. The Predictor is only for the HLS part of the design and does not include manual RTL. This is expected since only the HLS part of the DMRS design is modeled in C++, which is used to generate wrappers for the Predictor component. It is not unusual for the whole design or DUT to include some subblocks that are manually written and some that are generated through HLS. To support manual RTL verification when using Predictor, alternative methods must be included when the entire design needs to be verified and some design parts are not supported by the Predictor. An example of such a method would be using a reference file for the data stream coming from manual RTL. Similarly, if the data stream combines HLS and manual RTL

signals, the solution could be to combine HLS data from Predictor with reference data created by engineers.

4.3 Testbench Connections with Predictor

The starting point of this thesis was the fact that Nokia's UVM test benches used traditional Matlab-generated reference files to check that the outputs of the DUT match with expected data. In order to help speed up the design verification process, an enhanced method was requested. To help with this goal, it was preferred that the reference model should be a part of the UVM testbench. Using the Predictor as a reference model offers various different verification techniques that can be used with it. One such technique is the Constrained Random stimulus as it can generate tests on a much bigger scale than the traditional file Input/Output (I/O) stimulus. It is also faster to generate and execute the Predictor component and it takes less disc space than Matlab files.
The Predictor is integrated with similar connections on both UVM test benches. Figure 11 shows the testbench connections to both the Predictor and DUT. The Predictor is integrated as an outside component into UVM TB. It is connected to transformer components that are instantiated into the UVM environment. In this thesis, TLM ports were used upon implementation to the UVM testbench as well as for data streaming to the Predictor. The squares and circles in figure 11 represent TLM ports and connectivity between the Predictor and UVM TB.

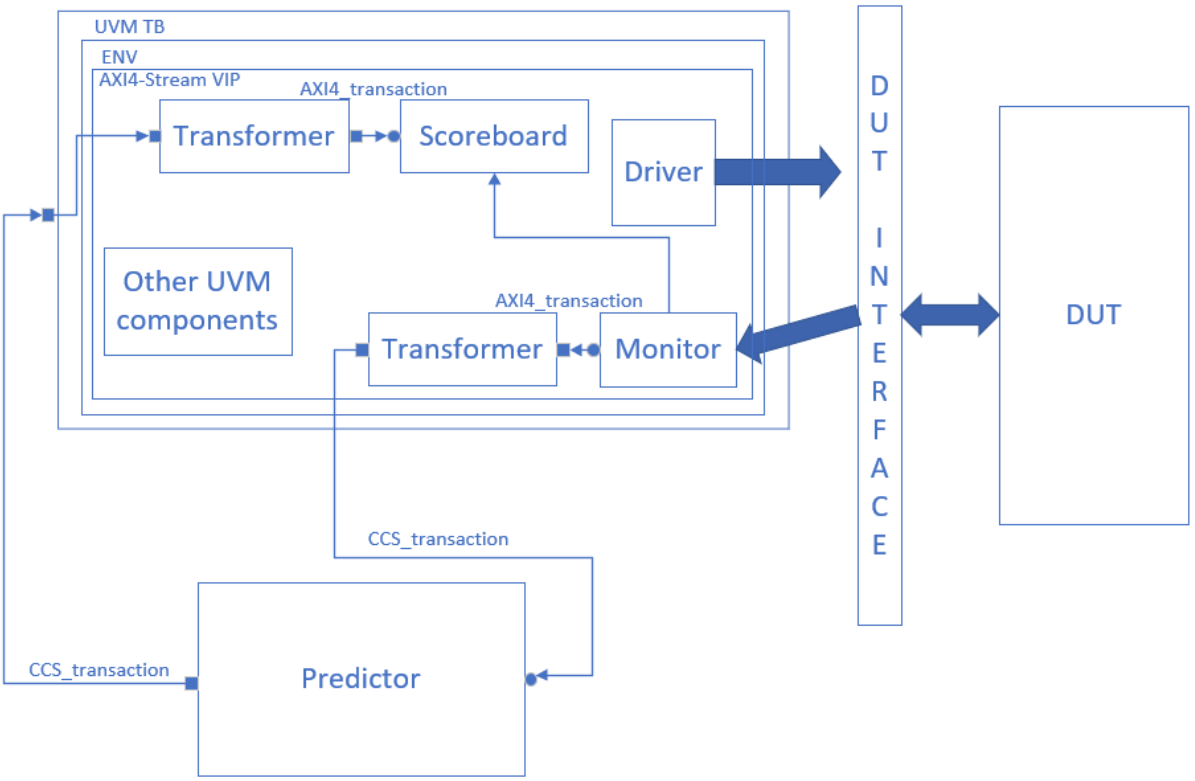

Figure 11. Predictor connections with UVM testbench.

The testbench's AXI4 data types are based on ARM's AXI4-Stream Protocol which is integrated into Nokia's in-house developed VIP [27]. CCS_transaction is a data type generated by the UVMF for the C++ reference model that is wrapped inside the Predictor.

CCS_transaction is generated as a class where an incoming transaction is converted into a generic payload by a function that is integrated inside the class.
All components for Predictor integration are instantiated, built, and connected in the UVM Environment at build_phase and connect_phase. An example of this can be seen in figure 12. The reference model uses the CCS data type, and the testbench for Submodule0 uses the AXI4-Stream data type. The DMRS TB uses AXI4-Stream with the design's internal signals or memory data type through memory interfaces. Verification Intellectual Property (VIP) with the AXI4-Stream bus handles data streaming through the DUT and Predictor with the same control and data channels. The DMRS Predictor's inputs are connected inside the DUT between Manual_RTL1 and HLS_block to control and I/Q data streams. Predictor outputs are connected to transformers and, finally, to the UVM Scoreboard. Connections are made with TLM analysis ports and analysis exports.

```
class env  extends uvm_env;
  //INSTANCES
  Predictor#(8,27,48,38,46) predictor_inst;
  data_transformer#(WIDTH,AXI4-Stream#(WIDTH))  data_to_predictor;
  Configuration_transformer#(WIDTH, AXI4_Stream#(WIDTH)) controltf_to_predictor;
  predictor_to_scoreboard_tf#(WIDTH, ccs_transaction#(WIDTH)) Transformer_toscoreboard;

  extern function new(string name = "env", uvm_component parent = null);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);
endclass

function env::new(string name = "env", uvm_component parent = null);
  super.new(name, parent);
  //CONSTRUCTING MASTER AND SLAVE SIDE AGENTS
endfunction

function void env::build_phase(uvm_phase phase);
  super.build_phase(phase);
  //BUILDING PREDICTOR AND OTHER UVM COMPONENTS
  predictor_inst  = Predictor#(8,27,48,38,46)::type_id::create("predictor_inst", this);
  data_to_predictor = Configuration_transformer#(WIDTH, AXI4_Stream#(WIDTH))::type_id::create("data_to_predictor", this);
  controltf_to_predictor = dmrs_dynamic_control_tf_to_predictor#(WIDTH,AXI4_Stream#(WIDTH))::type_id::create("controltf_to_predictor", this);
  Transformer_toscoreboard  = dmrs_predictor_to_scoreboard_tf#(WIDTH, ccs_transaction#(WIDTH))::type_id::create("Transformer_toscoreboard", this);
endfunction: build_phase

function void env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  `ifdef PREDICTOR
  ///TLM connections AGENT-TRANSFORMER-PREDICTOR
  axi_stream_agt_inst.slave_aps[CONFIGURATION_DATAHLS].connect(this.controltf_to_predictor.analysis_export);
  this.controltf_to_predictor.control_to_pred_ap.connect(predictor_inst.dmrs_dynamic_control_ae);

  axi_stream_agt_inst.slave_aps[IQ_DATAHLS].connect(this.data_to_predictor.analysis_export);
  this.data_to_predictor.data_to_pred_ap.connect(predictor_inst.dmrs_data_rsc_ae);

  //TLM connections PREDICTOR-TRANSFORMER-SCOREBOARD
  predictor_inst.output_ap.connect(this.Transformer_toscoreboard.analysis_export);
  this.Transformer_toscoreboard.d_to_sb_ap.connect(this.scoreboard_inst.scoreboard_export);
  `endif
endfunction:connect_phase
```

Figure 12. Predictor TLM connections to the UVM testbench.

In the study completed in 2020 [24], the C++ reference model was implemented into a UVM TB, where reference model implementation to the testbench and data streaming to the reference model were performed by using packages that included structure and function declarations in the SystemVerilog code. The structures contained fixed-size arrays where configurations and I/Q data were assigned. The function calls between DPI wrappers were manually implemented. [24, p.30–32] In both the 2020 study and this one, the C++ reference model was used but implemented differently.
The thesis completed in 2020 implemented the HLS C++ model directly to the UVM TB, which was wrapped between the SystemVerilog side and the C++ side wrappers. Between those wrappers, DPI was manually implemented in order to allow function calls between SystemVerilog and the foreign language C++. However, this thesis wrapped the C++ reference model inside the Predictor UVM component which allowed the use of TLM connectivity. The implementation of the Predictor is easier to do with TLM ports than connecting it by function calls, as its connections on the UVM side work like any other UVM component. Also, the Direct Programming Interface was implemented manually in the year 2020 thesis, but in this thesis, it was done automatically by the UVMF generator.

4.3.1 Predictor Generation

In this thesis, only the Predictor part was generated with the UVM framework, which takes YAML files as an input. YAML is a readable data serialization language that is usually used in cases where data needs to be transmitted. [8] Figure 13 shows this generation flow. It also shows which tool is used in which part of the flow.

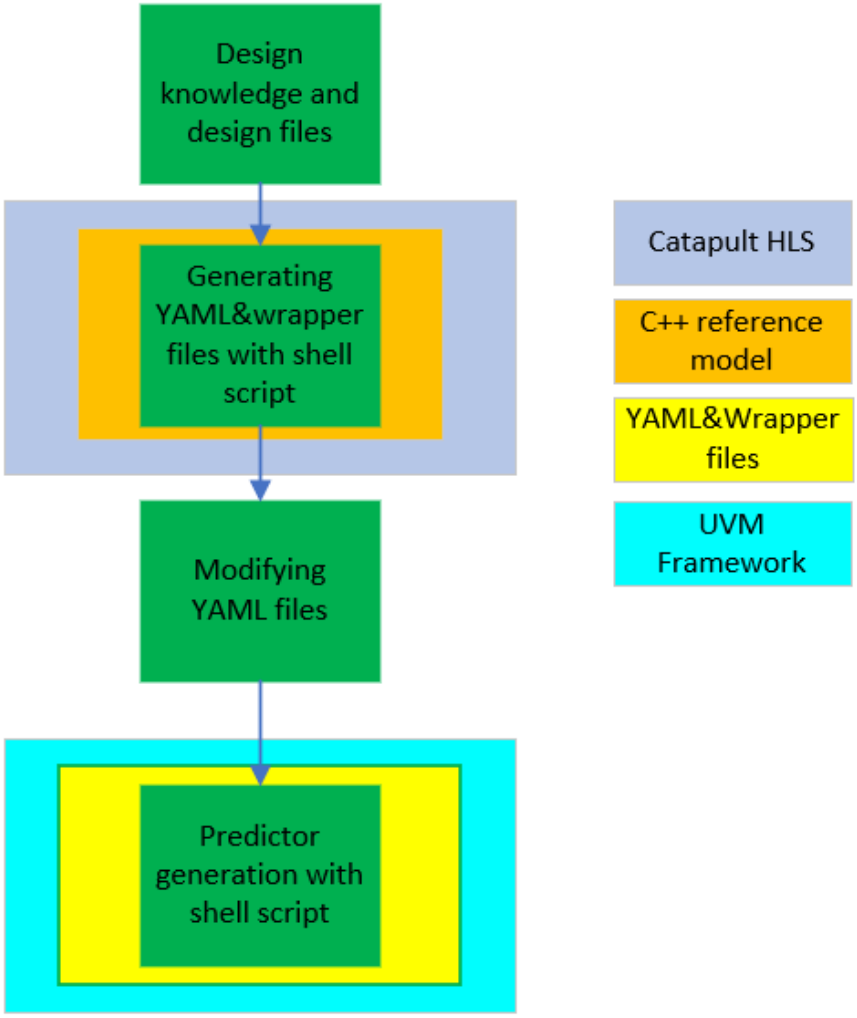

Figure 13. Predictor generation flow.

YAML files and TLM wrapper files are generated using Catapult. After YAML and wrapper files are generated, paths to C++ design codes and TLM ports are defined to the YAML file. Shell script generates the Predictor that has a C++ reference model wrapped inside the component. The C++ reference model works as an input for Catapult, and the modified YAML and wrapper files are inputs for the UVMF generator. The generated Predictor's packages and files are compiled with a separate Makefile, which is included inside the main Makefile. This makes it so that both the whole UVM TB and the Predictor are compiled with the same compilation. The Predictor uses generated CCS and Predictor environment packages to compile a generated Predictor at the same time with the UVM testbench as these packages are imported to the UVM test and environment packages. The CCS package imports all used packages and files that contain information about various ccs datatype interfaces. Predictor environment packages contain source files for Predictor configuration and structure.

4.3.2 Verifying Functionality of the Predictor

The assumption at the verification phase is that the C++ reference model which is wrapped inside the Predictor is working correctly because the C++ model has already been verified by the HLS flow in the C testbench. The UVMF-generated Predictor functionality was verified with DUT during the simulation process. First, the used C++ design files needed to be the same version as the verified design. This was to make sure that the Predictor worked, and if problems were to occur, they would not happen because of incompatibility problems between the design and the C++ design files. Secondly, the Predictor Makefile was integrated into the main Makefile and compiled with DUT. After the Reference model was fully integrated into the UVM testbench, the model's endpoint was in the Scoreboard component where the UVM comparator was instantiated.
The comparator was class uvm_in_order_comparator which organized incoming data from the DUT and Predictor into lines, and when there was data on both lines, it compared the data from the DUT and Predictor. The Comparator contained a counter that that gained in value each time when a new output sample came to the comparator. If DUT output samples did not match the output samples coming from the Predictor, the counter would issue a notification to alert to the disproportion of the samples. Finally, DUT and Predictor output samples are compared to each other, and if matching, it then verified that the Predictor was working.

4.3.3 Simulation of the Reference Model

The reference model was based on a frame-based design, which takes one set of configuration data and the right amount of I/Q data. The Predictor was created at build_phase and connected at connect_phase to the AXI4-Stream VIP, transformers, and Scoreboard. Catapult uses Kahn's Process Networks for computation modeling. With Kahn's Process Networks, it is possible to model parallel processes with sequential language (C++). [25, p.1]
Data feeding to Predictor is done in parallel by streaming the same I/Q data and control data that is going to DUT through TLM ports. From the Predictor side, accepting data into the C++ reference model is done with available checks. Available checks block the Predictor from reading the empty channel and checking that all inputs are available. Figure 14 presents an example of available checks inside C++ design files. I/O configuration and stimulus data signals to the design can be seen. Available checks check that the correct amount of configuration data exists in the channel, after which the design runs stimulus data available checks before the design continues processing. Available() is a method of the ac_channel library. Checks calculate values for the right amount of control and I/Q data, and when those values are true, the Predictor takes data through a block. Each subblock of HLS_block needed its own available check. Using the available method was discussed with Siemens EDA.

```
1 // HLS top
2 #pragma hls_design top
3 void DESIGN (
4         ac_channel<ConfigurationStream >                &CONTROL_INPUT,
5         ac_channel<DESIGN_fxps::designIn_complex_t >    &INPUT_DATA,
6         ac_channel<ConfigurationStream >                &CONTROL_OUPUT,
7         ac_channel<DESIGN_fxps::designOut_complex_t >   &OUTPUT_DATA0,
8         ac_channel<DESIGN_fxps::designOut_complex_t >   &OUTPUT_DATA1,
9         ac_channel<DESIGN_fxps::designOut_complex_t >   &OUTPUT_DATA2)
10 {
11     static ac_channel<DESIGNCtrl::ctrlStruct_t > intCtrlStream;
12
13   #ifndef __SYNTHESIS__
14   if(CONTROL_INPUT.available(configuration_samples))
15   #endif
16   {
17     DesignControl (CONTROL_INPUT, CONTROL_OUPUT, intCtrlStream);
18   }
19   #ifndef __SYNTHESIS__
20
21   if(intCtrlStream.available(1) && intCtrlStream[0].numPRBs>0 && INPUT_DATA.available(SAMPLES))
22   #endif
23   {
24     DESIGNProcessing (
25         intCtrlStream,
26         INPUT_DATA,
27         OUTPUT_DATA0,
28         OUTPUT_DATA1,
29         OUTPUT_DATA2);
30   }
31 }
```

Figure 14. Example of available() checks inside the design C++ files.

The stimulus was generated by using either Matlab-generated stimulus files or by generating data with constrained random (CR) stimulus. The UVM testbench for the bigger DMRS design used both of these methods, while Submodule0 used only Matlab stimulus files. The Predictor was concealed with wrappers that contained the UVMC library, which was implemented as a SystemVerilog package and a SystemC namespace. This enabled sending and receiving TLM data packets from the SystemVerilog-SystemC interface. Configurations and data exited from the Predictor's output side analysis_ports, which are connected to transformers at the connect_phase of the UVM Environment class. Data and configurations were sent to Scoreboard, where they were compared with the DUT's output.

4.4 Coverage Metrics used in this Thesis

In this thesis, Measured Coverage metrics varied and metrics were chosen with mutual understanding with Nokia and Siemens EDA. The functional coverage model was made for the Submodule0 testbench and measured with cover groups. The used coverage metrics for Submodule0 testbench were: Branch, Statement, FEC Expression and Condition, and cover groups. The DMRS testbench measured the Branch, Statement, FEC Expression and Condition, FSM, and Toggle coverage metrics. DMRS added the FSM and toggle coverage metrics, and no functional coverage model was taken into coverage closure flow for DMRS because such a model was not implemented.

4.5 Tools and Technologies

Multiple tools and technologies were used in this thesis, and they are categorized here.

4.5.1 Questa SIM

Questa SIM is a simulation tool from Siemens EDA for Verilog, VHDL, SystemVerilog, SystemC, and mixed-language designs. It offers automatic design optimizations, design file compiling, simulation loading and running, and debugging environment tools for tracking problems such as waveform viewing, stimulus generation with Waveform editor, and design connectivity testing. [15, p.21–25] In this thesis, Questa SIM was used for simulation run and debugging purposes with Waveform editor. Other features were not tried during thesis work.

4.5.2 Questa CoverCheck

Questa CoverCheck is a formal verification analysis tool from Siemens EDA for faster automatic coverage closure. CoverCheck examines the coverage database and items in an RTL design, analyzing reachability and unreachability through Post-simulation or Pre-simulation Analysis. The tool can be run in Graphical User Interface (GUI) mode or with Tcl commands. During Post-simulation Analysis, Questa CoverCheck receives coverage results from Questa SIM. This makes the analysis process faster when the CoverCheck tool does not have to process data that is already covered. CoverCheck analyzes all coverage items to define which ones are unreachable. Unreachable code is part of the code that cannot be covered or reached during simulation. [16, p.11–12]

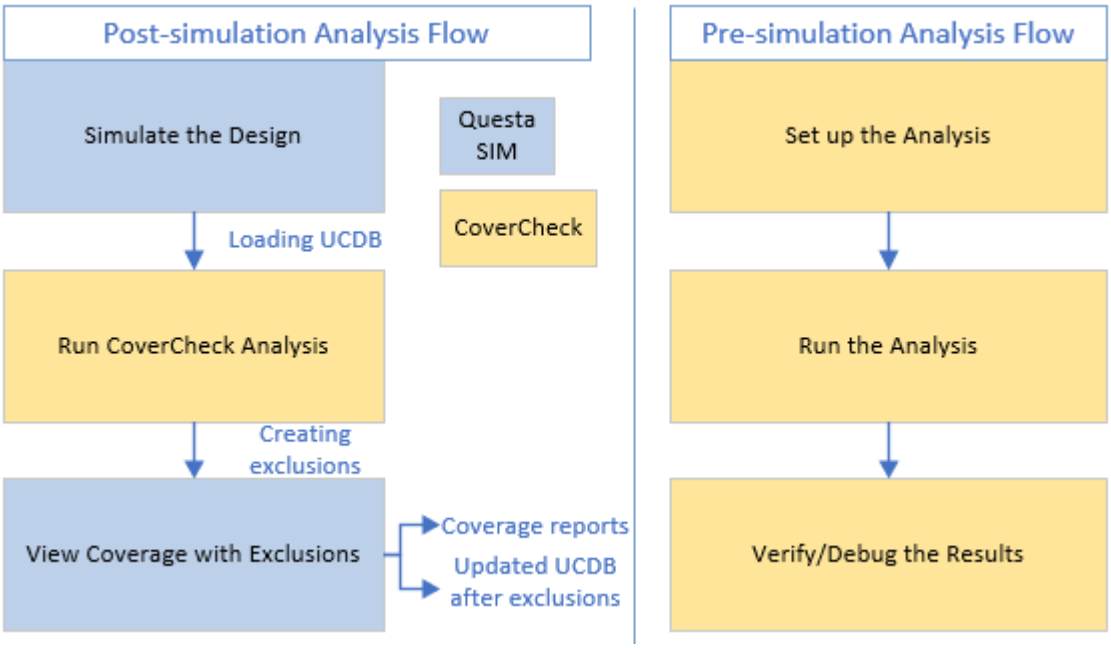

Figure 15. Post- and Pre-simulation Analysis Flows.

Figure 15 depicts the Post- and Pre-simulation flows during a Questa CoverCheck run. In Post-simulation flow, the Questa database is loaded by the Questa SIM and simulated. CoverCheck also runs the analyzing tool and automatically creates an exclusion file.
After Post-simulation Analysis, CoverCheck automatically creates Post-simulation phase reports, and exclusions generated by CoverCheck are applied to UCDB and unreachable code is now excluded from coverage analysis. In the Pre-simulation Analysis flow, only formal verification analysis for the whole design coverage items with CoverCheck is included [16, p.12–13] In this thesis, the Post-simulation Analysis flow was used to check unreachable code

from the Questa database. Questa CoverCheck commands were run by a separate Makefile Makefile_CoverCheck.mk which was included to base Makefile. To start the CoverCheck tool to check unreachables, the following Unix command was run:

- make cv DUT=<DESIGN> UCDB=<database> -f Makefile_CoverCheck.mk

In the Unix command, the verified DUT and the used coverage database needed to be defined.

4.5.3 Catapult

Siemens EDA's Catapult is an HLS and verification tool for automated ASIC and FPGA RTL implementation. It is compatible with C, C++, and SystemC design specifications as an input language to create VHDL or Verilog RTL. Catapult HLS supports multi-block synthesis with a Bottom-up or Top-down synthesis strategy. Both Bottom-up and Top-down are synthesized from source code. In the Bottom-up strategy, Catapult uses presynthesized units to map blocks of the design one block at a time, and after that a separate top-level build is executed. In the Top-down method, the whole design is built with one synthesize run.
Synthesized RTL is already set for simulation running and gate-level synthesis. Catapult also offers RTL power optimizations for lower power usage and tools for resource allocation handling, such as loop unrolling, merging, and pipelining. Catapult can be used with various procedures: in GUI mode for graphical experience, with the command line using scripts, and using source code as input for the tool. [17, p.24–25]

```
1 #!/bin/tcsh
2
3 setenv MGC_HOME <PATH_TO_CATAPULT_TOOL>
4
5 set arr = ("subblock0" "subblock1" "subblock2" "subblock3" "subblock4" "DMRS")
6
7 foreach block ($arr)
8   cd ./$block/c/catapult
9   rm -fr Catapult* catapult.* design_checker*
10  $MGC_HOME/bin/catapult -p prime -f *_10_5c.tcl -shell
11  cd ../../..
12 end
13
```

Figure 16. Shell script for synthesizing design and generating YAM and wrapper files

This thesis used Catapult HLS-generated RTL as a DUT. Catapult was also used to generate YAML and wrapper files from C++ design files. A flow where Catapult was used can be seen in figure 13. Figure 16 shows the shell script which was used to synthesize a design with the Bottom-up method to generate YAML and wrapper files.

4.5.4 Design Analyzer

Design Analyzer is part of the Catapult tool. Catapult Design Analyzer makes it easier to understand what is happening during the HLS process at different stages. With a ready Catapult-generated design project, Design Analyzer can be used to look at schematics and preview generated RTL code. It lets users trace RTL code lines back to the C++ level. This can make debugging possible because schematic locations can be pinpointed from generated RTL. [17, p.1173] This was attempted when trying to identify HLS coverage gaps. However, this was

found not to be helpful during the thesis work because no coverage gaps were able to be identified with it. Design Analyzer continues to evolve, which will make it an even better tool in the future.

4.5.5 Matlab

Matlab is a matrix-based language platform for programming and computing. It manages to verify big wireless communication systems, analyze big designs, automate test processing, machine and deep learning, and signal processing. Matlab supports several programming languages such as Python, C/C++, Fortran, Java, and many Component Object Model (COM) components. [18, 19, p.10] In this thesis, Matlab was used for generating test vectors for the Submodule0 block and the DMRS subsystem. Test vectors are different test scenarios that contain various values of system parameters. There are some restrictions on what kind of test vectors and on what signal levels they can be generated. The maximum and minimum signal levels might not be possible to achieve in real life, but to close the coverage gaps, it might be necessary to produce a test artificially in order to find out whether the design block can handle such a test.

4.5.6 Visualizer Debug Environment

The Visualizer Debug Environment is Siemens EDA's back-end graphical user interface simulator for the Questa SIM Simulator tool and Veloce emulator. It is a tool for viewing, analyzing, and debugging designs. The Visualizer contains several supported debugging features, and it has two working modes. It supports features such as live tracking of currently executing code and hierarchical UVM debugging, including sequences, threads, locals, and configurations. It also has waveform debugging of design signals, class objects, and transactions like the Questa SIM tool. Users can define the Visualizer's working mode from the following selections: Post-simulation mode or Live-simulation mode. In Post-simulation mode, the tool loads and views results from a previous simulation. In Live-simulation mode, the Visualizer uses the current (live) simulation. [28, p.21–22]
The Visualizer has an interesting feature that captures C++ side signal level activity to the same wave window as RTL level waves. Adding this feature would have required further tool integration on the Makefile side and was not tried in this thesis, but it would have made debugging Available() checks a faster process than adding prints for I/Q and control data at the C++ level.

4.5.7 Regression Management System

In this thesis, the tool's name is changed to Regression Management System (RMS) because it is a tool developed in-house at Nokia. RMS makes it possible to run basic shell commands in a flexible way. It is Nokia's inner regression management tool, and it is based on Siemens' VRM system. Table 2 shows the files belonging to the RMS system and their descriptions.

Table 2. Files that belong to the RMS system and their descriptions

File	Description
RMS	Python-based front-end to use the RMS system. Imports rms_base.py
regression.xlsx	Executable command definitions for regressions
rmsbase.py	Python-based regression system implementation
rms.rmdb	Contains rules for Siemens VRM to execute commands in parallel in GRID
xlsx2csv.py	Converts the XLSX file to CSV format for parsing

The RMS is a great tool for running multiple test vectors in parallel, and it is used in this thesis. It contains a regression.xlsx file that has multiple regression commands. In regression.xlsx it is possible to declare the used simulation tool, memory size, seed number, commands, and different execution phases. Each command belongs to a phase, and the phases are executed sequentially. After regressions are run, the tool indicates whether the regression passed or failed and generates reports from simulation results.

4.6 Problems with Generated RTL

HLS RTL generation can save time from manual RTL writing and speed up verification, but some problems that do not occur with manual RTL may arise. It often happens that machine-generated RTL is not easily readable, even though the RTL structure is reiterative and understandable, which makes understanding coverage gaps easier. This is why it may be difficult to start looking for coverage gaps in the code by reading the RTL. When analyzing HLS RTL, the best way to start is to first aim for high coverage at the C++ level with a high level of abstraction. After that goal is achieved, it is recommended to move to the RTL level, where parts that are not visible in the untimed description are covered.

The HLS tool might generate some unused code which causes coverage gaps. Unused code is a part of code that is never used. If the verification engineer needs to know how to toggle some bits or how some finite state machine state shift is happening, the information might be difficult to come by because even the designer of the block might not be able to tell from reading the generated RTL. Even though coverage closure for HLS-generated RTL is harder at the RTL level, it does not mean that it should not be done. Developing tools to help analyze generated code and obtaining knowledge on how to do it may bring about some productive results.

The year 2013 study showed that HLS-generated RTL used a similar number of resources as manual RTL, but generated RTL cut the design's cycle, decreased latency by a factor of 30x, and offered similar results when comparing to manually written RTL. [21, p.1,4] The advantage with generated RTL is that it will most likely meet the requirements in area or time execution, but manual RTL might not, which is why it can consume more time to manually write working RTL.

5 COVERAGE CLOSURE AND RESULTS

5.1 RTL Coverage Closure Flow

Figure 17 shows the flow used for RTL coverage closure. The whole coverage closure flow starts from a higher level of abstraction by covering code at the C++ level. The coverage closure process was done by a colleague, covering the C++ reference model using Catapult Coverage (CCOV) flow. A colleague found and ran a set of tests that offered a good coverage percentage in statement, branch, and FEC at the C++ level. Knowing which tests delivered good coverage at the C++ level created a good starting point for testing at the RTL level. The RTL level focus was more geared toward covering the remaining HLS RTL gaps, such as FIFOs and stalls.

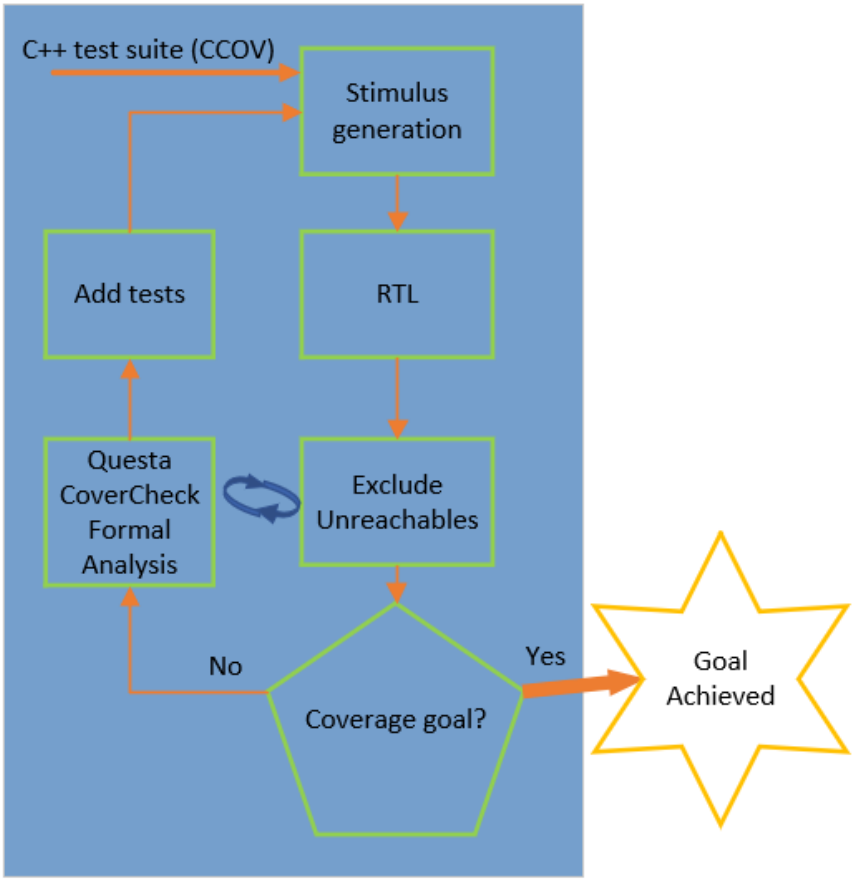

Figure 17. RTL coverage closure flow.

RTL was fed with stimulus, and coverage databases were run iteratively with the CoverCheck tool. If the coverage goal was not achieved, more tests were added depending on the coverage gaps. A key part of this flow was to cooperate closely with the designers. If coverage gaps could not be identified, the designer of the block could always be consulted on how to identify the gaps and the potential methods of covering them.

5.2 Testing Strategy

The testing strategies presented here were used to achieve as close as possible to 100% code coverage.

5.2.1 Submodule0 Testing Strategy

Matlab was used to create a set of test vectors for Submodule0 to obtain results of first baseline coverage. These vectors contain tests, and one vector can have multiple sets of tests that increase code coverage. Information about the vectors that needed to be generated was received from the designer of Submodule0. There were about 600 vectors that were run with the regression manager tool (RMS) and generated with Matlab. These vectors contained different system parameters in order to obtain good coverage over parameter ranges. The vectors had control and IQ data files that were used as input for feeding the DUT. Some of the author's own stall tests were also added to verify the block. The idea of stall tests is to feed control data or IQ data to the channel so that it fills Submodule0's inner control and processing blocks.
Stall conditions happen when the block's output rdy and vld signals are forced down and input data feeding continues. After the stall has happened, the block does not accept any new control or stimulus data. Subsequently, those outputs are released so the block can recover and continue processing. If a block does not recover, it is not working correctly. The Submodule0 testbench used only file I/O-based test sequences.
After getting baseline coverage for Submodule0, Questa Covercheck was used to check unreachables and states that were impossible to occur in the RTL simulation and exclude them from the Questa coverage database. This step was repeated as many times as new unreachable coverage holes were found. When new unreachable holes were no longer found, the database was opened in Questa GUI to check for coverage gaps, and the block designer was consulted in order to remove some unnecessary code.
The most effective tests were found with the test ranking feature in Questa. The test ranking feature analyzes which tests and vectors improve code coverage and which do not. There were only three tests that improved RTL code coverage, so simulations were run again with tests that increased coverage and other test vectors were not used. At this point, code coverage was almost the desired 100%. To reach 100% coverage saturation, conditions had to be exercised. This was achieved by modifying the stimulus data by manually altering in-phase/quadrature (I/Q) data by changing I and Q pairs to theoretical maximum and minimum.

5.2.2 DMRS Testing Strategy

The testing strategy and coverage closure flow for the DMRS subsystem was similar to the Submodule0, but with some differences. There were two regression runs executed. One regression run was performed with Matlab-generated stimulus as the DUT's input, which included stall tests for the DMRS's HLS subblock. The test vectors contained various values of system parameters. Another regression run was performed with added Constrained Random (CR) stimulus for control data and I/Q. These two regression coverage databases were merged to cover some test gaps that were not tested during constrained random regression or when using Matlab files. CR made it possible to do a larger variety of tests on different signal levels. Unreachable states and unused code were analyzed with Questa Covercheck iteratively, and project exclusion files were examined with the designer and excluded from the coverage results. Also, few coverage gaps were closed by looking into the C++ coverage results. When the best

coverage results with the CoverCheck tool were achieved, most of the remaining gaps occurred because of stall testing. Some of the stall conditions were impossible to generate due to how the DMRS block's subblock's datapath was implemented together with the manual RTL's, even though this was not identified as unreachable by the CoverCheck tool. The DMRS's HLS block has fifteen subblocks and to verify all stall conditions, these subblocks would have to have been fed full of IQ data and configuration data to get them to stall. The problem with this was how the previous Manual_RTL1 worked with the HLS block. It was possible to close some gaps by doing specific testing that was artificially aimed directly at the gap.

5.3 Coverage Results

These results were achieved with the testing strategies presented in previous chapters. Table 3 shows the coverage results of Submodule0. Table 4 presents coverage results when only the baseline vectors from a designer are run. In table 5, the final results of the DMRS are listed when stall tests and exclusions from the coverage database are included. The values are presented as percentages. In table 5, a positive value under the percentage signifies an improvement in percentage. For example, (+5.07) indicates that the coverage percentage improved by that much. In the coverage report, N/A means Not Available, which happened to Manual_RTL1 in FEC Expression coverage.

Table 3. Coverage results of Submodule0

Block\Coverage Metric	Branch %	Statement %	FEC Expression %	FEC Condition	Covergroups %
Submodule0 (DUT)	100	100	100	100	100

Table 4. Coverage results of DMRS and its subblocks before stall tests and exclusions

	Block\Coverage Metric	Branch %	Statement %	FEC Expression %	FEC Condition %	FSM %	Toggle %
DMRS (DUT)	HLS_block	78.84	94.92	79.31	36.06	79.75	89.20
	Manual_RTL1	89.81	89.00	N/A	54.83	70.95	80.37
	Manual_RTL2	93.03	97.36	100	66.66	78.88	89.81

Table 5. Coverage results of DMRS and its subblocks after all tests and exclusions

	Block/Coverage metric	Branch %	Statement %	FEC Expression %	FEC Condition %	FSM %	Toggle %
DMRS (DUT)	HLS_block	99.91 (+21.07)	99.99 (+5.07)	93.74 (+14.43)	93.15 (+57.09)	100 (+20.25)	93.96 (+4.76)
	Manual_RTL1	100 (+10.19)	100 (+11.00)	N/A	100 (+45.17)	100 (+29.05)	98.70 (+18.33)
	Manual_RTL2	99.06 (+6.03)	100 (+2.64)	100 (+0.00)	100 (+33.34)	100 (+21.12)	97.17 (+7.36)

Submodule0 was a small subblock, so it did not require many tests to achieve the 100% coverage target, and thus no constrained random stimulus was added into the testbench. The HLS part of DMRS was not able to achieve a 100% coverage because of the limitations of how Manual_RTL1 was designed and integrated with the HLS part. This caused stall tests to not work properly. Stall tests were directed only at the HLS block. Manual_RTL1 fed configurations and I/Q to the HLS block, and stall tests began to delay the HLS block's outputs. The problem was that when the first subblocks inside the HLS began to stall and Manual_RTL1 noticed that it is not possible to feed data through the outputs, it stopped feeding data. This made it impossible to keep feeding configurations and data to the HLS block because the previous Manual_RTL1 block did not allow data feeding if, at the same time, that data could not also be fed out. This problem did not occur with Submodule0 because data was fed directly to the HLS block and Submodule0 was very small. In DMRS some interface stalls froze data feeding, and as a result, it was not possible to feed more data. DMRS was a more complex design than Submodule0.

Coverage results were good. The 100% coverage goal was achieved for Submodule0, and overall coverage results were also acceptable for the DMRS block, even though the 100% coverage goal was not reached. It would have been possible to achieve 100% code coverage for the DMRS's HLS part. To get the HLS part of DUT fully verified, the HLS block would have to be verified separately by generating stimulus directly to it. Coverage closure at the RTL level is also possible even though it is more difficult. However, fortunately there were many tools and methods to help in that process.

6 DISCUSSION

This thesis aimed to generate and implement a UVM Framework-based Predictor that wraps the C++ reference model in the UVM testbench with HLS-generated RTL and to do coverage closure to the DUT. In this thesis, a newly adopted verification method was taken into account during the verification process. As a result, the UVM Framework-generated Predictor with the wrapped C++ reference model was successfully integrated for a smaller IP as well as a larger subsystem containing multiple of such IPs.

Using a UVM Predictor generated by UVMF as a golden reference instead of Matlab-generated files showed promising results. A generated UVM Predictor takes less disc space than vector files and it is faster to generate a working model than vectors that are generated with Matlab. Reference models are usually implemented with foreign programming languages such as C, C++, or Python. One benefit of the RTL coverage closure flow used in this thesis is that it enables the possibility of fast coverage results, and it also works well with HLS-generated RTL, where coverage gaps might not be so easily recognized.

Using Nokia's in-house verification platform offered a good starting point for UVM testbench development for Submodule0. This provided the skeleton for building a testbench. The only downside to the platform was that a lot of files and code were received that were not needed by the smaller IP. The platform is definitely more suitable for bigger IPs. Also, the UVM agent that was used was an AXI4-Stream type VIP, which would have been more suitable for IPs that use the full AXI4-Stream bus. The agent had to be modified to get configuration and stimulus data through. Looking back, it might have been easier and perhaps faster to make a simpler agent from scratch.

The goals laid out for this thesis were achieved, and a functional UVMF Predictor was generated with a wrapped C++ reference model and integrated into the testbench. It is a feasible option to make the verification process smoother.

UVM Framework is used as a generator for UVM code. Because it can generate large test benches very quickly, it is also a good tool for demonstrating things. In this thesis work, UVMF was used to generate the UVM predictor component which acts as the reference model for the verified design. Reusing design files for generating a golden reference model is time-saving and practical as it most likely works for verifying the design due to a reference model already having been verified at a higher level of abstraction. However, it has to be ascertained that design files are up to date with the RTL design that must be verified. In addition, the Predictor responds better to different verification methods as well as different stimulus generation approaches, such as Constrained Random (CR) stimulus generation. CR allows doing testing on a bigger scale and obtaining similar or better coverage results faster than with Matlab files, which was also demonstrated in the 2020 thesis [24]. Also, simulation times will decrease compared to running a simulation with Matlab-generated files.

The ideal method would be to start the analysis of the design at the C++ level with the Catapult Coverage tool, which brings the RTL coverage metrics to the C++ level, and then take those coverage results and move to the RTL level to analyze the remaining gaps. This has usually been the case due to generated code not being easily human-readable, but in which structures are repetitive which then makes gaps more understandable. This made it possible to identify all stall and FIFO fill level gaps. A colleague ran CCOV, identified the C++ reference model's gaps, and documented them. This made it possible to identify some of the gaps in the HLS code as gaps that are happening at a higher level of abstraction at the C++ level as ones that are also happening at the HLS RTL level.

There are different varieties of Predictor components. They can be made manually or automatically and have different foreign languages at their core. It is time-consuming to create reference models manually, and with deadlines closing in, every second counts. In this work, the reference model was developed by engineers, but the UVM Predictor was generated by UVMF.

When moving to bigger and more complex IPs writing Predictor, its interfaces and connectivity can be a difficult task to do manually. That is why UVMF is a great tool to generate a Predictor as it generates the whole UVM Predictor with a wrapped C++ reference model and automatically uses DPI between the foreign language and SystemVerilog. It also integrates a bunch of useful tools and technologies such as Visualizer.

Machine-generated RTL and unfamiliarity with it caused some difficulties during the coverage closure process, but most of the problems were able to be solved by either asking a designer or by looking at the VCS tool's exclusion files from the project. Sometimes even the designer of the block may not know what is causing the gap or how some generated FSM states happen in RTL code. There might be also some code that is not in use but is generated in RTL. These lines of code should be removed from the final generated RTL because they cause unnecessary coverage gaps. The UVM Predictor's generation and integration into UVM Testbench for a smaller IP took longer than for a larger subsystem's testbench. The smaller IP's UVM testbench was done from scratch, and Predictor generation and integration flow were also learned.

The process to integrate Predictor into testbench was easy to do because it obeyed the basic functionality of the UVM component connectivity because data could be transferred through the Predictor via TLM analysis ports. The generation process and integration did take some time to learn and did not go as smoothly for the Subblock0 testbench as the DMRS testbench.

For the purposes of future projects, it would be interesting to find out what kind of results could be achieved if the whole UVM testbench was generated with UVMF because it makes it possible to generate even large-scale IPs in just a few hours. Additionally, it would also be interesting to determine the differences between the fully generated UVM TB and the manually written UVM TB. Some parts of the UVMF Predictor generation flow could be affected by human error and cause the Predictor to not work properly. For example, this kind of error could happen when writing on a YAML file. The human errors are mostly misspellings or typos. These could be solved by automating the generation process even further, even though some level of knowledge of the design should be obtained.

Awareness of the design in design teams should be increased, so that design engineers could plan ahead or determine which interfaces can or cannot stall due to design implementations and verification engineers would be made aware if a gap is excludable. The available checks that were made so data could be fed through Predictor could also be made by a designer because a designer would know best when a block should begin taking data and configuration.

7 SUMMARY

Designs will become both larger and more complex in the future, which is why it is good to develop different verification methods and tools to accelerate the verification process. The goal of this thesis was to generate a working UVMF-generated UVM Predictor component which wrapped the C++ reference model inside the predictor. The predictor is then integrated into the UVM testbench. Integration is performed first for an HLS Submodule0 as DUT and then for a bigger subsystem DMRS, which had two manually coded RTL subblocks with one HLS subblock. After integration coverage, closure is done for DUT. The achieved levels of code coverage for HLS RTL were satisfactory in this thesis, and it was found that design style can impact how a design works when testing certain conditions, such as stall conditions.

First, this thesis presented the general verification-based UVM theory and the UVM components that were used. Chapter three presented code coverage metrics. In chapter four DUTs were introduced with testbench connections with the Predictor along with generation, simulation, and testing of the Predictor. Used coverage metrics, tools, and technologies were also presented. Chapter five represented RTL coverage closure flow with results gained from coverage closure. After that results of the thesis are discussed.

The generated Predictor offered techniques to be used with the verification flow, such as constrained random stimulus. The Predictor worked well with a constrained random stimulus, which allowed similar or better coverage results with a shorter simulation time to be obtained. Achieved coverage results were good. The smaller IP, Submodule0, was able to be fully verified by code coverage, up to 100%. DMRS achieved a good overall coverage result, even though some subblocks of the DMRS were not fully verified. The HLS block of DMRS could have been verified to full 100% if it could have been verified separately without the manual RTLs connected to it. Some gaps were left due to design limitations or simply because there was no information available on whether the gap should be happening or not. This is true at least of the toggle coverage. HLS designers should consider design testability and how design acts during testing. From a coverage point of view, using a Predictor is a way of verifying design and not increasing coverage.

In closing, when verifying a DUT, the golden reference that is used has several different variations. Some may take more time to integrate or offer more functions with other techniques. Using a generated Predictor instead of Matlab reference files showed great promise to be used in future projects.

8 REFERENCES

[1] Oden B., Craft J., Aerne D., Horn M., (2019-2020) UVM Framework- One Bite at a Time

(Accessed 3.6.2021) URL:

https://verificationacademy.com/courses/UVM-Framework-One-Bite-at-a-Time

[2] Siemens, (2021), Version 2021.1, UVM Framework Users Guide, p.148

[3] Konale S., Bheema Rao N., (2014), C-based Predictor for Scoreboard in Universal Verification Methodology

[4] Allan G., Chidolue G., Ellis T., Foster H., Horn M., James P., Peyer M., Mentor Graphics, (2019), Universal Verification Methodology UVM Cookbook, pdf, p.547

(Accessed 4.5.2021) URL:

https://verificationacademy.com/cookbook/uvm

[5] Siemens, (2021), Version 2021.1, UVMF YAML Reference Manual, p.45

[6] Accellera, (2015) Universal Verification Methodology (UVM) 1.2 User's Guide, p.190

(Accessed 21.6.2021) URL:

https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[7] Cadence, Open verification methodology

(Accessed 22.6.2021) URL:

https://www.cadence.com/ko_KR/home/alliances/standards-and-languages/open-verification-methodology.html#

[8] YAML

(Accessed 22.6.2021) URL:

https://yaml.org/

[9] Horn M., van der Schoot H., Peyer M., Fitzpatrick T., Stickley J., Mentor Graphics, (2019), Coverage Cookbook, pdf, p.95

(Accessed 23.6.2021) URL:

https://verificationacademy.com/cookbook/coverage

[10] Mentor Graphics, (2020, Version 2020.2), Questa SIM User's Manual, p.939

[11] Mentor Graphics, (2020, Version 2020.2), ModelSim SE User's Manual, p.1900

[12] Sarah L. Harris & David Money Harris, (2016), Digital Design and Computer Architecture ARM Edition, p.732

[13] IEEE. (2017) IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, p.1315

[14] IEEE. (2012) IEEE Standard for SystemC Language Reference Manual, IEEE Std 1666-2005, p.1163

[15] Mentor Graphics, (2021, Version 2021.1), Questa SIM Tutorial, p.298

[16] Mentor Graphics, (2021, Version 2021.2), Questa CoverCheck User Guide, p.76

[17] Mentor Graphics, (2020, Version 10.6), Catapult Synthesis User and Reference Manual, p.1258

[18] MathWorks, Matlab

(Accessed 23.7.2021) URL:

https://se.mathworks.com/products/matlab.html

[19] MathWorks, (2021, R2021a), Matlab, p.164

[20] Pumar M., Siemens, (2019), UVM Framework + Questa Verification IP, A Winning Combination

(Accessed 26.7.2021) URL:

https://verificationacademy.com/seminars/u2u-silicon-valley/uvm-framework-plus-questa-verification-ip-a-winning-combination

[21] Winterstein F., Bayliss S., George A., (2013), High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS, p.4

[22] Damak T., Ayadi L., Masmoudi N., Bilavarn S., (2015), HLS and manual Design methodology for H.264/AVC Deblocking Filter, p.5

[23] Siemens, UVM Connect

(Accessed 28.7.2021) URL:

https://verificationacademy.com/topics/verification-methodology/uvm-connect

[24] Liakka M., (2020), MASTER'S THESIS CONSTRAINED RANDOM IMPLEMENTATION FOR UNIVERSAL VERIFICATION METHODOLOGY TESTBENCH WITH C/C++ REFERENCE MODEL, p.50

[25] Ceponis J., Kazanavicius E., Mikuckas A., (2002), Design and analysis of DSP systems using Kahn process networks, p.4

[26] Siemens, DualTop

(Accessed 16.8.2021) URL:

https://verificationacademy.com/cookbook/dualtop

[27] ARM, (2010, Version 1.0), AMBA 4 AXI4-Stream Protocol Specification, p.42

[28] Mentor Graphics, (2020, Version 2020.2), Mentor Visualizer Debug Environment User's Manual, p.506