CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Universal-Verification-Methodology-Based Verification Strategy for High-Level Synthesis Design

Master's thesis in Embedded Electronic System Design

Chi Zhong
Haonan Shen

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A Universal-Verification-Methodology-Based Verification Strategy for High-Level Synthesis Design

Chi Zhong
Haonan Shen

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Universal-Verification-Methodology-Based Verification Strategy for High-Level Synthesis Design
Chi Zhong, Haonan Shen

Supervisor:
Per Larsson-Edefors, Department of Computer Science and Engineering

Industrial Mentor:
Ion-lucian Sosoi, Ericsson

Industrial Supervisor:
Stefano Olivotto, Ericsson

Examiner:
Lena Peterson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

A Universal-Verification-Methodology-Based Verification Strategy for High-Level Synthesis Design

Chi Zhong, Haonan Shen
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

High-level synthesis (HLS) enables automatic translation from high-level language code to register transfer level (RTL) code, which could be a potential method to improve the development efficiency for hardware design. However, a different verification flow should be introduced to guarantee the HLS design fulfills design specifications. The purpose of this thesis is to explore and develop a new universal-verification-methodology-based (UVM-based) verification workflow particularly for HLS design. A universal strategy is developed in two different vendors' tools, Cadence Incisive™/Xcelium™ and Siemens Mentor Graphics QuestaSim™/Visualizer™, including automatic scripts and highly-reusable code to verify the HLS C++ design and the HLS RTL design generated by Siemens Mentor Catapult™. The design under test is a complex 5G communication block design from Ericsson. We first investigated the existing Cadence-based UVM verification environment, and updated the current flow into an HLS-specialized flow. Then, we explored how the verification strategy is realized with Cadence tools including Incisive/Xcelium. After that, we developed an entire flow in Siemens Mentor Graphics environment, by migrating the existing UVM verification environment architecture with Siemens Mentor Graphics supported libraries and developed the automation process for compilation, optimization and simulation in Siemens Mentor QuestaSim/Visualizer. Furthermore, to test the feasibility of the Siemens Mentor Graphics flow we designed, the flow is applied to collect results of functional coverage and code coverage of an Ericsson's IP block. In the process of reaching coverage closure, the intermediate results indicate the demands of developing a series of additional direct tests, and suggest potential changes in the test plan for regression test. Finally, based on results in practicing the HLS verification flow in Cadence and Siemens Mentor Graphics environment, summarized suggestions are given to Ericsson for further improvements in the HLS verification flow.

Keywords: High-level synthesis, UVM, IP verification, Multi-language verification.

Acknowledgements

We would like to thank our supervisor, Professor Per Larsson-Edefors and our examiner, Professor Lena Peterson for their guidance and suggestions in this project. We would like to thank our advisor from Ericsson, Ion-lucian Sosoi, for precious help and guide during this thesis. We are also thankful to our supervisor in Ericsson, line manager Stefano Olivotto, for his administration and constant support throughout the thesis.

Chi Zhong, Haonan Shen, Gothenburg, June 2022

Contents

Acronyms

API application programming interface. 43
ASIC application-specific integrated circuits. 1, 3–5, 16

DPI direct programming interface. 11, 34, 39, 44
DUT design under test. 4, 11, 13–22, 24, 26, 28, 29, 32, 34, 40, 46–48, 57, 59, 60, 63

EDA electronic design automation. 2, 3, 10, 12, 18, 37, 43, 49, 59, 60, 63

FEV formal equivalence verification. 6
FPGA field-programmable gate arrays. 1
FSM finite state machine. 3, 25

GUI graphical user interface. 60

HDL hardware description language. 1
HLS high-level synthesis. 1–4, 7, 15–21, 31, 32, 34, 37, 40, 46, 47, 49, 51, 57, 59, 63

IP intellectual property. 3, 6, 16

LSF load sharing facility. 60

ML multi language. 32–35, 42, 43, 59

OOP object-oriented programming. 9

RTL register transfer level. 1, 3, 7, 8, 10, 12, 15, 18–22, 32, 34, 37, 40, 46–48, 56, 57, 59, 60, 63

SoC system-on-chip. 1, 10, 13, 63
SVA SystemVerilog assertions. 6

TCL tool command line. 37, 38
TLM transaction-level modeling. 5, 11–13, 17, 22, 24, 29, 33, 34, 42–44

UCDB unified coverage database. 47–49, 55, 60, 61
UVM universal verification methodology. 2–6, 10, 13, 15–24, 26, 28, 29, 32–35, 37, 38, 42–44, 46, 47, 52, 59, 61, 63

VME verification makefile environment. 61, 62

1
Introduction

The complexity in system-on-chip (SoC) design has been dramatically increasing in recent decades, which forces designers to shift toward adopting higher abstraction levels than register transfer level (RTL). Expediting the automation of both synthesis and verification processes is increasingly important nowadays where time to market is a crucial factor for competitiveness of companies that are developing sophisticated subsystems of SoC designs [1, 2]. Therefore, from a perspective of design productivity, high-level synthesis (HLS) is an alternative choice, compared to using a traditional hardware description language (HDL) such as Verilog and VHDL. HLS enables automatic translation of abstract, untimed or partially timed specifications written in high-level languages such as C++, SystemC to the hardware design described by cycle-accurate RTL code, which can be efficiently deployed in application-specific integrated circuits (ASIC) and field-programmable gate arrays (FPGA). Besides, the translation process can be customized depending on the performance, power and cost requirements, which allows designers to explore the design space according to their particular systems [3]. HLS is rapidly gaining popularity in industry because it is a promising approach to reduce the design efforts and help overcome time-to-market challenges while maintaining comparable performance and energy efficiency of the produced hardware design to those that are manually coded in classic HDLs such as Verilog and VHDL [4].

However, current HLS tools are unable to always guarantee that the automatically generated hardware designs are equivalent to the source high-level specifications. For example, Herklotz et al. [5] revealed through their fuzzing testing approach that among four commercial HLS tools, all of them could generate erroneous designs based on the given high-level programs. In fact, 1191 test-cases out of total 6700 failed in at least one of the tools. Thus, HLS tools are not fully trustworthy and the resulting hardware design generated by HLS tools can be buggy as well. To ensure the functional correctness, functional verification must be performed, which is usually carried out by simulating the generated hardware design against an extensive testbench. Yet the shortcoming of this technique is that unless all inputs are covered exhaustively in the testbench, which is generally unrealistic, there is a risk that some bugs still remain in the design as they are not detected by the limited input test cases [4]. There are some other approaches that can be harnessed to check the functional correctness of the generated design but each of them has advantages and downsides. Therefore, an efficient workflow for verifying the functional equivalence between the high-level behavioral description and the corresponding RTL code created by HLS tools needs to be explored.

1.1 Thesis goals

This thesis aims at devising an efficient flow based on universal verification methodology (UVM) for HLS verification, in which UVM is a framework of SystemVerilog classes for building fully functional testbenches [6]. We investigate and update the existing verification strategy in a Cadence environment provided by Ericsson; then we develop our own entire verification strategy in a Siemens Mentor Graphics environment; finally the limitations and further improvements for the flows in the two environments should be evaluated. To be more specific, the goals for different stages are listed below.

(1) *Assess the current UVM-based verification flow in Cadence provided by Ericsson and update the flow into an HLS-specialized verification flow:*

Cadence tools are widely used in Ericsson for design verification purpose. The architecture of the existing UVM testbench in Cadence is investigated. Moreover, the functional coverage and the code coverage produced by random test cases based on the UVM testbench in the Cadence-based environments are examined.

To update the current flow into an HLS-specialized verification flow, UVM components in the UVM testbench and the connection for the language boundary (SystemC and SystemVerilog) should be modified for the particular HLS design. Moreover, since it takes much more time to build a desirable verification environment utilizing various components compared to using equivalence checking performed by electronic design automation (EDA) tools, it is essential for us to figure out all the verification details before we move to our own development in Siemens Mentor environment.

(2) *Develop an entire UVM-based HLS verification flow in Siemens Mentor Graphics verification environment for the same HLS design generated by Siemens Mentor Catapult High-level Synthesis Platform:*

We aim to develop an entire UVM-based HLS verification flow based on Siemens Mentor Graphics tools. Based on the Cadence testbench architecture, the new Mentor-based testbench is modified to be compatible with Mentor-provided libraries. A script or makefile should be developed for the compilation, simulation and regression process to improve process automation. UVM environments are highly reusable and able to provide flexibility and ease for maintaining testbenches. More importantly, UVM is an open-source standard so it is free of charge. The flow we develop should be tested with an Ericsson's IP block, there would be three major verification metrics for our case in this phase:

- Test Pass Rate
- Functional Coverage

- Code Coverage

If the metrics yield 100%, then we can sign off the verification stage. If the metric is $< 100\%$, then we need to investigate and analyze the missing coverage and decide how to implement new tests to cover the missing part.

(3) *Evaluate the pros and cons of carrying out the same verification flow in Cadence and Siemens Mentor Graphics environments:*

Several key factors for benchmark are compared: the simulation time, the C++ code coverage capability, the C++ to SystemVerilog communication complexity, the debugging tools capability for C++ errors, the complexity of the new UVM components needed for HLS verification and the regression flow. Then we provide suggestions of whether the Siemens Mentor Graphics tools for HLS verification can be an alternative choice for Ericsson, based on our careful consideration and assessment.

1.2 Challenges

For functional coverage, the test vectors for simulation should be applied to the HLS-generated RTL and the functional coverage should attain 100%, whereas the percentage for code coverage is considerably lower, around $60 - 80\%$ [7]. The high-level language has some undefined behavior. For example, divide by zero is defined in RTL but undefined in C code [8]. Therefore, the RTL model contain a larger scale of finite state machine (FSM). Consequently, the RTL code generated by HLS has some redundant architecture. In order to reach a higher coverage rate, new tests for FSM states are essential to verify the correctness of each state [7].

Although HLS design tools are gradually applied in the industry, the verification methodology for HLS still remains immature, which demands continuing research and investment efforts. Furthermore, the quality of automatically generated RTL is heavily dependent on the HLS tools provided by the EDA vendors, which might affect the cost of verification. From Ericsson's perspective, Siemens Mentor Graphics Catapult is used to generate the HLS RTL design. For common practice, Ericsson uses Cadence verification tools for ASIC/IP projects. If the UVM-based verification flow could be performed by Siemens Mentor Graphics verification tools, Ericsson could be less dependent on one specific EDA vendor. Therefore, Ericsson can have more choices for setting up a verification environment.

Another significant issue is that a compilation script or a makefile system needs to be designed from scratch, which would be a significant disadvantage due to the large number of files involved and the numerous factors to be considered. Otherwise a subtle problem like the compatibility of the compilers, libraries and simulators could hinder us from achieving more progress for a long period.

1.3 Thesis outline

This thesis is organized in the following way: Chapter 2 describes the technical background of ASIC verification and one of the most the popular verification methodology in this field. Chapter 3 describes the three-phase flow for HLS design verification. Chapter 4 illustrates the UVM-based verification testbench for different types of design under test (DUT). Chapter 5 and Chapter 6 explore the manners in which to connect SystemVerilog-based verification environment and the high level SystemC DUT to enable the communication between the DUT and the UVM testbench in Cadence and Siemens Mentor Graphics environments separately. Then Chapter 7 presents and compares the coverage results. Eventually Chapter 8 and 9 contains the conclusion based on the results and some further discussion.

2
Technical Background

In this chapter, the context of ASIC verification emphasizes the importance of verification and demonstrates the trend in recent decades. Besides a bird's eye view, several key concepts that will be brought up in this thesis are presented to lay a solid technical background of the thesis topic, including two design and verification languages SystemVerilog and SystemC, the multi-language communication protocol transaction-level modeling (TLM) and the key methodology UVM.

2.1 ASIC verification

In an ASIC design flow, the main goal of the verification process is to guarantee the design correctness before the design is taped out. In the 1990s, the infamous Pentium FDIV bug occurred in the floating point multiplication lookup table of the processor, which cost Intel $475 million for processor replacements [9]. Thus, a trial-and-error approach should be prevented since it is difficult to alter the hardware and it will cause amplified expense of replacement and reputational damage if a problem is discovered by consumers.

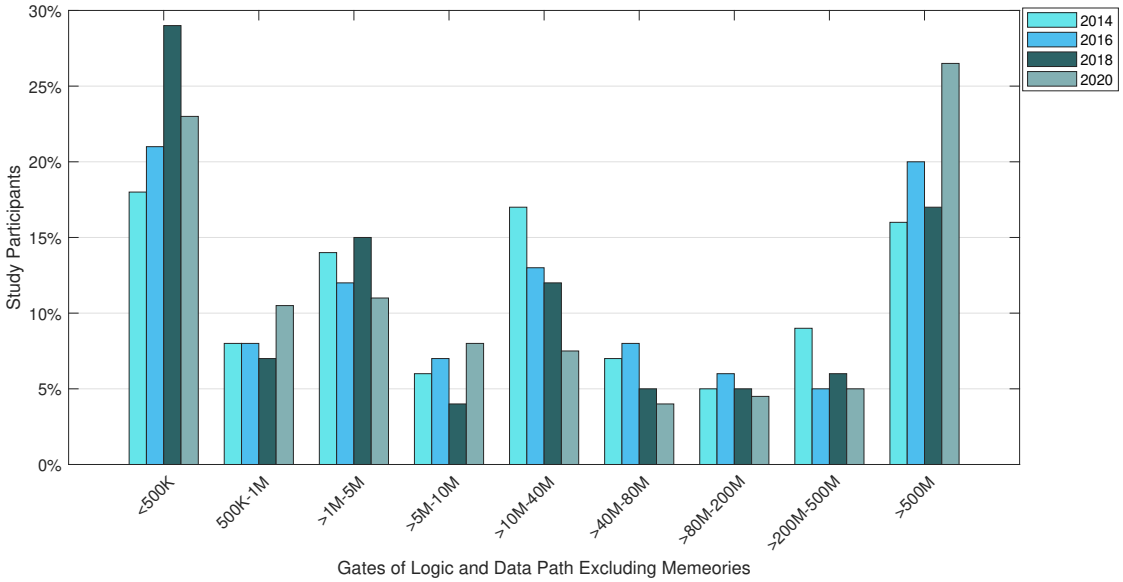

Figure 2.1: IC/ASIC study participants by design scale (reproduced from [10]).

As shown in Figure 2.1, the design complexity in the electronic industry is growing rapidly from 2014 to 2020. In addition, designs containing more than 500M gates take the largest market share in 2020.

As a result, the overall development time for the verification process is increasing. The designs which are based on existing pre-verified intellectual property (IP) only need a very short time for verification, while the other designs which are based on newly developed IPs require longer time for verification. Figure 2.2 indicates that for complex chip designs, there is a trend that the verification process takes longer time than the design process [10]. Around 50% of the total effort is devoted to the verification process to make sure the design is mature and ready for market [11].

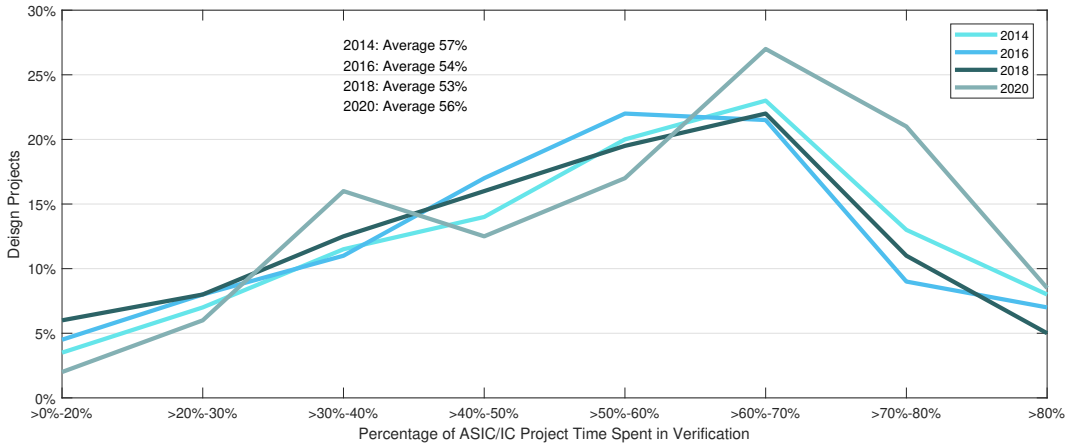

Figure 2.2: Verification time percentage of IC/ASIC project (reproduced from [10]).

Nowadays, product teams invest massively in design verification including testbench simulations, SystemVerilog assertions (SVA), formal equivalence verification (FEV), coverage metrics checking, debugging etc. In regards to the verification methodology standards, Figure 2.3 shows that the UVM standard developed by Accellera Systems Initiative has predominance over other methodologies in the market, and it has gained increasing adoption over recent years [12].

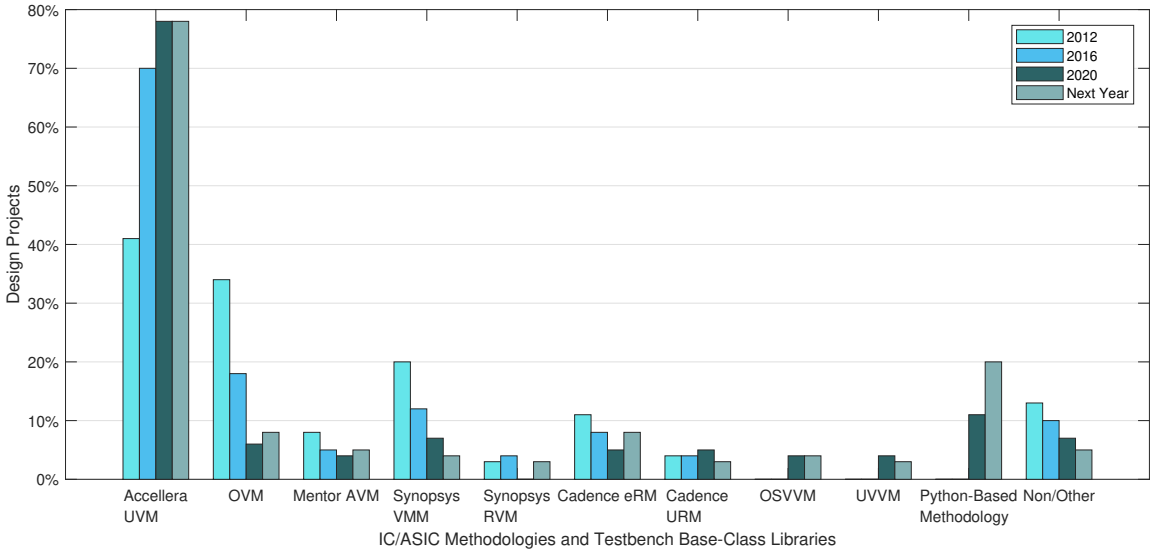

Figure 2.3: IC/ASIC methodologies and testbench base-class libraries (reproduced from [12]).

HLS tools contribute to a 5X to 10X code reduction for a design written in C++ or SystemC, which significantly decreases the design efforts [13]. Maturing the verification process for an HLS design is important as the potential market of HLS becomes scalable. HLS verification can be classified into two categories: The first one is the functional verification of the source specification, the other is the verification of the generated RTL [14]. To make HLS more practical in a real design flow, an efficient methodology for HLS verification is crucial and beneficial to enhance the production efficiency and curtail the verification cost.

2.2 Coverage metrics

There does not exist one single metric that can completely characterize the verification process [15]. For example, during a series of simulation regressions, 100% code coverage is attained. However, a 100% code coverage result cannot imply that all the functionalities have been verified. Moreover, if a 100% functional coverage result is reached, it does not mean that a 100% code coverage can be achieved at the same time. If the code coverage is not 100%, it indicates that one of the key design behaviours is missing in the coverage model, or some design features that have been implemented are not originally stated in the testplan. Therefore, multiple metrics are essential to obtain a thorough picture of a project's verification process.

Table 2.1: Different categories of coverage metrics [15].

Implicit	Code Coverage	Area of Research (Currently no industrial solutions)
Explicit	Assertions	Functional Coverage Assertions
	Implementation	Specification

From Table 2.1, the coverage specification is demonstrated. Different coverage types are specified by the method of creation (e.g. explicit vs. implicit) or the origin of source (e.g. specification vs. implementation) [15]. Two commonly used forms of coverage metrics in industry are code coverage metric which is implicit coverage and functional coverage metric which is explicit coverage.

2.2.1 Functional coverage metric

Functional coverage is a typical example of an explicit coverage type. The aim of a functional coverage metric is to determine whether the design specifications are working as intended or not [15]. A functional coverage model is created manually by the user. To create a functional coverage model, first the functionality to be measured should be identified, then the machinery to measure the functionality should be implemented accordingly.

A typical functional coverage flow can be summarized into four main steps, which is shown in the following Table 2.2.

Table 2.2: A typical four-step functional coverage flow [15].

Step Number	Contents
1	Create a functional coverage model
2	If using assertions, instrument the RTL model to gather functional coverage
3	Run simulation for multiple times to capture and record the functional coverage metrics.
4	Report and analyze the functional coverage results.

During the last step in the functional coverage flow, the coverage holes are identified and the analysis for the identified uncovered code should be carried out. There are two major reasons of resulting coverage holes. The first reason is missing input stimulus for uncovered functionality. If the current testing stimulus is insufficient for activating the uncovered functionality, additional directed testing stimulus should be written to generate the direct input for the uncovered functionality. The second reason is the design has some internal bugs that prevents the input stimulus from activating the uncovered functionality. In this case, moving back to work together with the design engineers is essential to fix the bug in the design.

2.2.2 Code coverage metric

Code coverage measures the percentage of structures in the source code that have been activated during simulation [15]. The structural coverage model will be generated in an automatic process, allowing the structures that have been activated or those that have not been activated to be detected automatically. Therefore, the integration of the code coverage into an existing simulation flow is simple and does not have an impact on the design or the verification flow [15]. As an implicit method, the code coverage might not detect some bugs in the design due to the insufficient amount of input stimulus, and the functionalities have been tested cannot be specified by the coverage result.

The typical code coverage flow is similar to the typical functional coverage flow. A typical code coverage flow can be summarized into three main steps, which is shown in the following Table 2.3. In an industrial practice, the RTL implementation should be close to completion before gathering and analyzing code coverage.

Table 2.3: A typical three-step code coverage flow [15].

Step Number	Contents
1	Instrument the RTL code to gather code coverage.
2	Run simulation for multiple times to capture and record the code coverage metrics.
3	Report and analyze the code coverage results.

During the last step in the code coverage flow, the coverage holes are identified

and then analysed. There are two major reasons of resulting coverage holes. The first reason is missing input stimulus for inactivated structures. If the current testing stimulus is insufficient for activating the missing structures, additional directed testing stimulus should be written to generate the direct input for the uncovered part. Another approach is to alter the randomization constraints for a larger amount of testing stimulus. The second reason is the design has some internal bugs that prevents the input stimulus from activating a certain part of the structure. In this case, a discussion between the verification team and the design team is required to fix the bug in the design.

2.2.3 Regression tests

Regression test is a process of running a series of tests repeatedly in some cycle, in order to verify the design and to guarantee that earlier debugging has not influenced the overall functionality. Because it is quite likely that a change in a design that is made to fix a problem and may be detected by a debugging test case, would jeopardize the functionality that was previously verified. Therefore, regression testing should be distinguished from debugging. In addition, another benefit of regression testing is a reassurance that the design is backward compatible with the original design. In addition to that, the results of regression testing can be captured and accumulated so the simulation data can be presented to verification resources to aid their productivity or their ability to achieve coverage closure [15].

2.3 SystemVerilog UVM

2.3.1 Testbench from VHDL/Verilog to SystemVerilog

The conventional verification testbenches are written in VHDL/Verilog. All the test vectors are decided by designers. These test vectors are specific to the design and they are not reusable. Since the design features vary and there are no standard practice, each time a new testbench should be developed for a new design.

VHDL/Verilog can be a simple and convenient solution to verify the functionality of designs which are relatively small and not really complex. However, as the design complexity increases, there is a need to develop a language with more object-oriented programming (OOP) features.

SystemVerilog is a unified hardware design and verification language extended from Verilog with many verification features such as assertions and coverage. SystemVerilog is a better solution to VHDL/Verilog because it is able to support complicated testbenches due to its capabilities to use OOP features in assertions and functional coverage, etc [16].

2.3.2 Testbench from SystemVerilog to UVM

In order to tackle the increasing complexity of validating RTL designs, a well-supported methodology, UVM, is proposed to enhance the verification efficiency and code re-usability. UVM is a popular verification standard which can be considered as a set of class libraries implemented by SystemVerilog. It provides a standardized manner for developing modular and reusable testbench structures and verification environments, which is increasingly important when the complexity of SoC designs grows and more engineers with different skill domains are involved in the verification process because standardized testbenches and environments enable portability and compatibility of various verification components among multiple projects. This will significantly boost the verification efficiency from the perspective of industry [17].

By means of UVM, it is convenient to perform coverage-driven simulation based on constrained random stimulus. In addition to that, assertion-based verification and hardware emulation could also be easily implemented using UVM and it is supported by major EDA vendors, making it an efficient and productive alternative for companies to verify complex SoC designs [18].

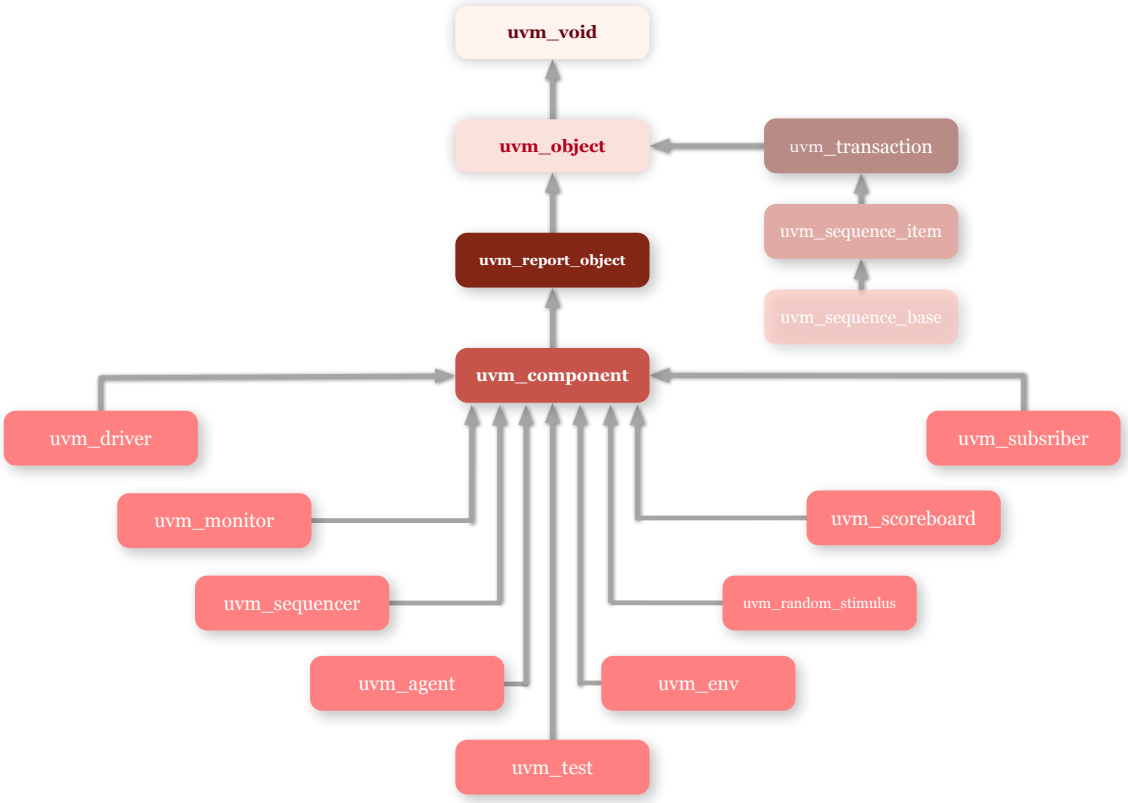

Figure 2.4: Structure of the UVM hierarchy.

As shown in Figure 2.4, there are a few essential components in a UVM hierarchy. The various blocks of the UVM-based testbench are discussed below.

- *Driver*: A driver is used to drive signals to the DUT. To be more specific, it converts the transaction level data received from the sequence to bit level data that conforms to the DUT ports.

- *Monitor*: A monitor is responsible for capturing signal level information from DUT output ports and translate it into transaction level data in order to send the data to other components like the scoreboard.

- *Sequencer*: A sequencer generates data transactions as class objects and sends it to the driver. It also controls the flow of request and response sequence items between sequences and the driver.

- *Agent*: An agent simply contains a sequencer, a driver and a monitor. It can be configured to operate in active or passive mode. In the active mode, it enables required data via a driver to the input ports of DUT and also monitors the output ports of the DUT, whereas in the passive mode, it only monitors the output ports of DUT.

- *Scoreboard*: A scoreboard is an essential component that checks and verifies the correctness of the DUT. Generally it receives transaction level data captured from the interfaces connected to the DUT through transaction-level modeling (TLM) analysis ports. After receiving data objects, it can either implement calculations and produce the golden values or send the data to a reference model to get the expected values. The validation of the DUT is achieved by comparing the expected results with the actual output data from the DUT.

2.3.3 Multi-language feature: direct programming interface (DPI)

Modern verification environments need to be built by SystemVerilog together with other languages, so DPI is used for interfaces between SystemVerilog and foreign languages such as SystemC, C and C++.

Since the data types are different in SystemVerilog and foreign languages, data needs to be interpreted in a uniform way for both ends. In the simulator installation path, a default header named `svdpi.h` is available for constructing the interfaces and the data type mapping between SystemVerilog and the foreign languages when doing the compilation with DPI-related flags.

Two DPI methods are implemented for calling functions between two languages. The first one in Source Code 2.1 is the import method, which allows the functions implemented in the foreign language environment to be called from SystemVerilog side. The second one in Source Code 2.2 is the exported method, which allows the functions implemented in SystemVerilog to be called from the foreign language side.

Source Code 2.1: The DPI implementation on SystemVerilog side.

DPI in SystemVerilog.

```
module Bus(input In1, output Out1);
  import "DPI" function void slave_write(input int address,
                                         input int data);
  export "DPI" function write;
  // Note - not a function prototype

  // This SystemVerilog function could be called from C
  function void write(int address, int data);
    // Call C function
    slave_write(address, data); // Arguments passed by copy
  endfunction
  ...
endmodule
```

Source Code 2.2: The DPI implementation on SystemC side.

DPI in SystemC.

```
#include "svdpi.h"
extern void write(int, int);
// Imported from SystemVerilog
void slave_write(const int I1, const int I2)
{
  buff[I1] = I2;
  ...
}
```

2.4 SystemC

SystemC is a unified system design and verification language, which extends from standard C++ with several methodology- and technology-specific open-source class libraries [19]. Nowadays, SystemC is used by leading system design companies and EDA vendors to make early-stage architectural exploration. Given that the design usually includes complex architecture or algorithms, such kind of highly abstract features make it possible to carry out architectural analysis tasks significantly faster and more productive than in an RTL model.

TLM is integrated into SystemC so that it can transfer data between the SystemC design and the SystemVerilog testbench. The application of SystemC TLM models will enhance the simulation performance compared to using RTL designs.

2.5 TLM

TLM is a modeling paradigm used to address the problems of increasingly complexity of SoCs by building highly abstract models of components and systems [20]. In this methodology, data is transmitted via data transactions that are represented as class objects containing protocol-specific information. Data information flows in and out of various components through specific ports called TLM interfaces.

In a transaction-level model, components are isolated from each other so that the changes in a component would not affect other components. Therefore, unnecessary details of communication and computation are hidden and may be added later, which promotes reusability and flexibility because a component can be swapped with another that also has a TLM interface. This mechanism allows engineers to explore and validate design alternatives at a higher level of abstraction, which will expedite the simulation in today's verification environment because of the large number of signals associated with various protocols [20].

TLM-1 and TLM-2.0 are two TLM modelling styles that are developed to build transaction-level models. TLM-1 is primarily used for passing messages. For example, In a UVM-based verification environment, the monitor is usually connected to the scoreboard via a TLM-1 analysis ports so that the data packets captured in the monitor can be sent to the scoreboard. However, TLM-1 interfaces are either untimed or rely on the target for timing. TLM-2.0, while still enabling transfer of data and synchronization between independent processes, is mainly designed for high performance modeling of memory-mapped bus-based systems [21]. Nevertheless, TLM-2.0 features are not fully implemented in UVM and there are also several limitations in the current UVM implementation. For instance, the key sockets in standard TLM-2.0, *tlm_initiator_socket* and *tlm_target_socket* have no direct counterparts in UVM. In addition, UVM does not fully implement TLM-1 non-blocking interfaces. Calls to these interface methods by connected SystemC-side ports will produce a run-time error [22].

2.6 Description of the "Pterodactyl" DUT

The real name of the design in Ericsson is not supposed to be revealed, so we created a nickname for our DUT called Pterodactyl. The "Pterodactyl" DUT consists of five blocks, which can be seen in Figure 2.5. There are two main input data signals, `TX0` and `TX1` respectively and these two input data signals called `in_data` could be added to two additional data signals called `test_stim` according to two configuration bits after passing through the add block. The output data after the add block will be multiplied with a value called gain in the multiplication block. In this case the gain is specified as one so the output from the multiplication block is the same as the output from the add block. Furthermore, the hclip block does nothing but bypasses the input so that the output would be identical with the input. Hence, if we call these two configuration bits as `data_en` and `test_en` respectively, the four types of

output signals can be concluded as shown in Table 2.4.

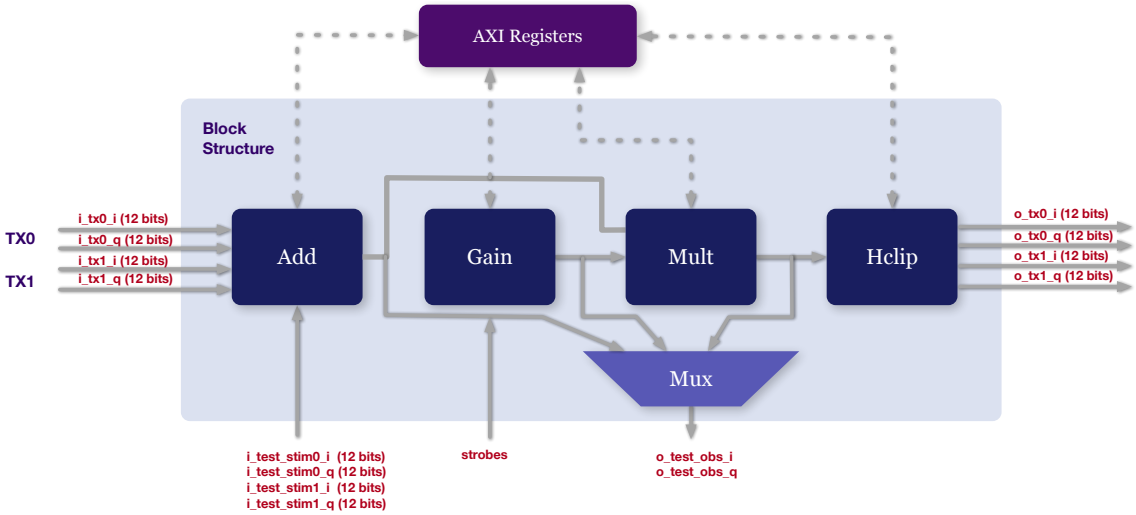

Figure 2.5: Block diagram of the "Pterodactyl" DUT.

Table 2.4: DUT output based on configuration bits.

data_en	test_en	output
0	1	zeros
1	0	in_data
1	1	in_data + test_stim
0	1	test_stim

3

Universal HLS verification flow

In this chapter, a three-phase UVM-based verification flow is introduced for general designs realized by HLS tools. Regardless of the UVM verification testbench, there are three different HLS-related DUTs in three phases. According to the HLS design flow in the left part of Figure 3.1, the first HLS DUT is written in higher level language, which is a C++ design. The second HLS DUT is an RTL level design translated to Verilog/VHDL by HLS tools, e.g. Siemens Mentor Graphics Catapult. The third DUT is an optional integrated design, which merges the HLS RTL design and other potential RTL blocks.

Therefore, the entire HLS verification flow is divided into three phases by verifying different forms of HLS DUTs. The middle part of Figure 3.1 shows the universal three-phase HLS verification flow. The first phase is to verify HLS C++ design, the second phase is to verify HLS RTL design and the third phase is to verify the integrated design.

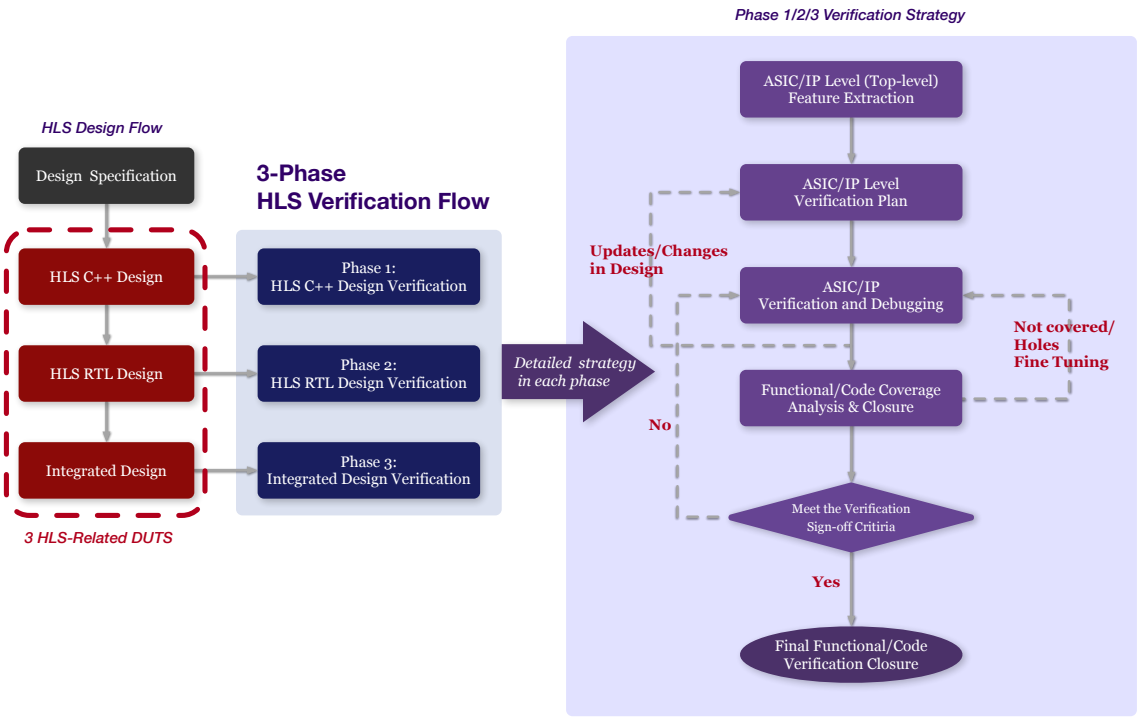

Figure 3.1: The top view of the 3-Phase UVM-based HLS verification flow.

The verification strategy in each phase is shown in the right part of Figure 3.1, which

is described step by step in the following enumeration.

1. *ASIC/IP level (Top level) feature extraction:*
   The first step is to do the top-level design feature extraction. A design specification study is carried out to investigate the functionality and architecture.

2. *ASIC/IP level verification plan:*
   In the second step, a verification plan needs to be created, modified and finalized before starting the verification process in the latter steps. In the verification plan, the intended behaviour of the ASIC/IP DUT should be defined, the functionalities of sub-IPs should be identified and the coverage collection requirements should be determined. If extra features are added to the design, the verification team will move back to update the verification plan.

3. *ASIC/IP level verification environment development and debugging:*
   In the third step, after the verification needs are reviewed and confirmed with the verification team, the reusability of the existing components is taken into account, such as the integration of sub-IPs and the legacy verification environment. For example, the sub-blocks UVM environment and test scenarios can be reused if they are integrated with top-level ASIC/IP design. In the top level environment development, the standard verification components should be redeveloped, e.g. top level driver, monitor, scoreboard, checker, and reference model. After that, the top level test scenarios needs to be developed. Finally, the debugging process will be a time-consuming task for the verification team. The verification engineers need a solid understanding of the functionality of the DUT and a clear view of the verification components and interconnections.

4. *Functional/Code coverage analysis and closure:*
   In the fourth step, the tests are run according to the verification plan and functional/code coverage is collected. Further coverage analysis contributes to reaching the goal of 100% coverage. Carrying out consecutive regression tests also helps to be more close to the 100% goal.

5. *Verification sign-off criteria check:*
   The last step before sign-off is to check whether the verification criteria is met or not. If all the listed verification criteria has been successfully fulfilled, the functional verification process can be concluded and the final functional verification closure can be claimed.

The verification environments are different in three phases depending on three different DUTs, which will be explained in the following three sections. In the first half of each section, the HLS-related DUT is introduced accordingly. In the second half of each section, the corresponding verification environment that includes both UVM testbench and DUT in each phase is illustrated. Moreover, the connection and the interaction between UVM testbench and DUT will be demonstrated.

3.1 HLS C++ design verification environment

Starting from the first phase in the verification flow, when verifying the HLS C++ design, the actual DUT is a design written in C++ by the design team. As shown in Figure 3.2, a SystemC top which includes a SystemC wrapper is generated, and the SystemC wrapper calls the HLS DUT by calling a specific function.

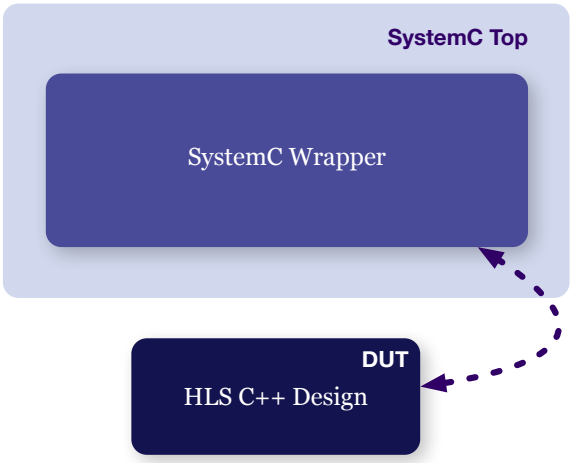

Figure 3.2: HLS C++ design DUT in phase 1.

Figure 3.3 illustrates the connection and data transaction between UVM testbench and SystemC top. The detailed UVM objects and components in the UVM environment will be discussed in the following chapter. Only the most important components are listed in Figure 3.3. Sequences of test stimulus are provided to multiple agents. The scoreboard receives output signals from the DUT via virtual interfaces. Virtual interfaces are pointing to actual interfaces, which contain the encapsulated signals. A Matlab model called by the scoreboard is used to validate the DUT. The Matlab model gets the input data from interfaces and computes the expected output data for reference. The scoreboard compares the output from the DUT and the outputs from the Matlab reference model. Finally, the functional coverage is collected.

The key difference between the two UVM environments is that the UVM environment for C++ DUT includes a SystemC adapter. The input data for DUT is encoded and sent via the TLM-2.0 protocol from SystemC adapter to SystemC wrapper. The TLM-2.0 protocol application between SystemC and SystemVerilog will be illustrated in detail in the following Chapter. The SystemC wrapper calls the HLS C++ DUT and gets the actual outputs from the DUT. All the inputs and outputs from SystemC adapter are acquired via virtual interfaces.

In the first phase, tests should be implemented, and 100% functional coverage should be achieved. Meanwhile, the code coverage should reach 100% for HLS C++ code.

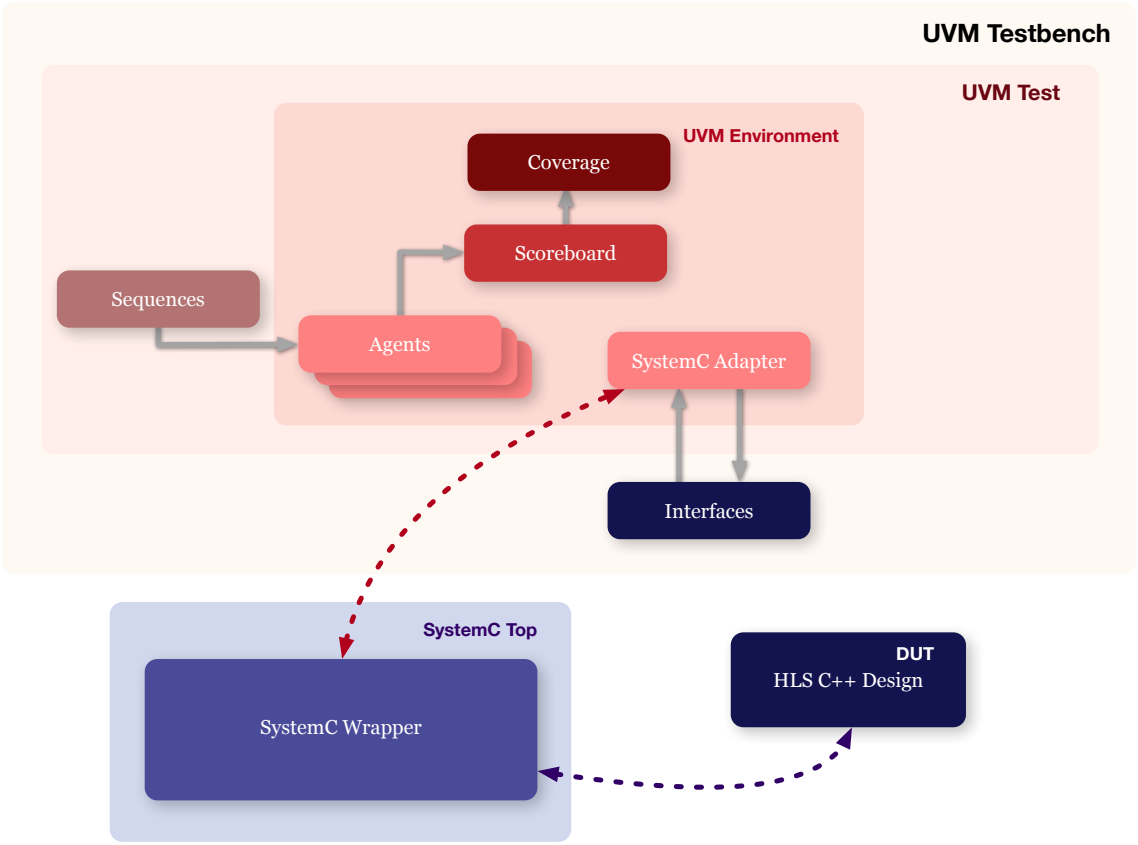

Figure 3.3: HLS C++ design verification environment in phase 1.

3.2 HLS RTL design verification environment

In the second phase, the resulting RTL design from the synthesis of the HLS C++ code is verified. It needs to be mentioned that the design team gives the verification team both the HLS C++ design and the corresponding RTL design generated by EDA HLS tools at the same time.

Figure 3.4: HLS RTL design DUT in phase 2.

The DUT in phase 2 is a regular RTL design as shown in Figure 3.4. Therefore, the verification strategy is similar to the conventional UVM-based verification.

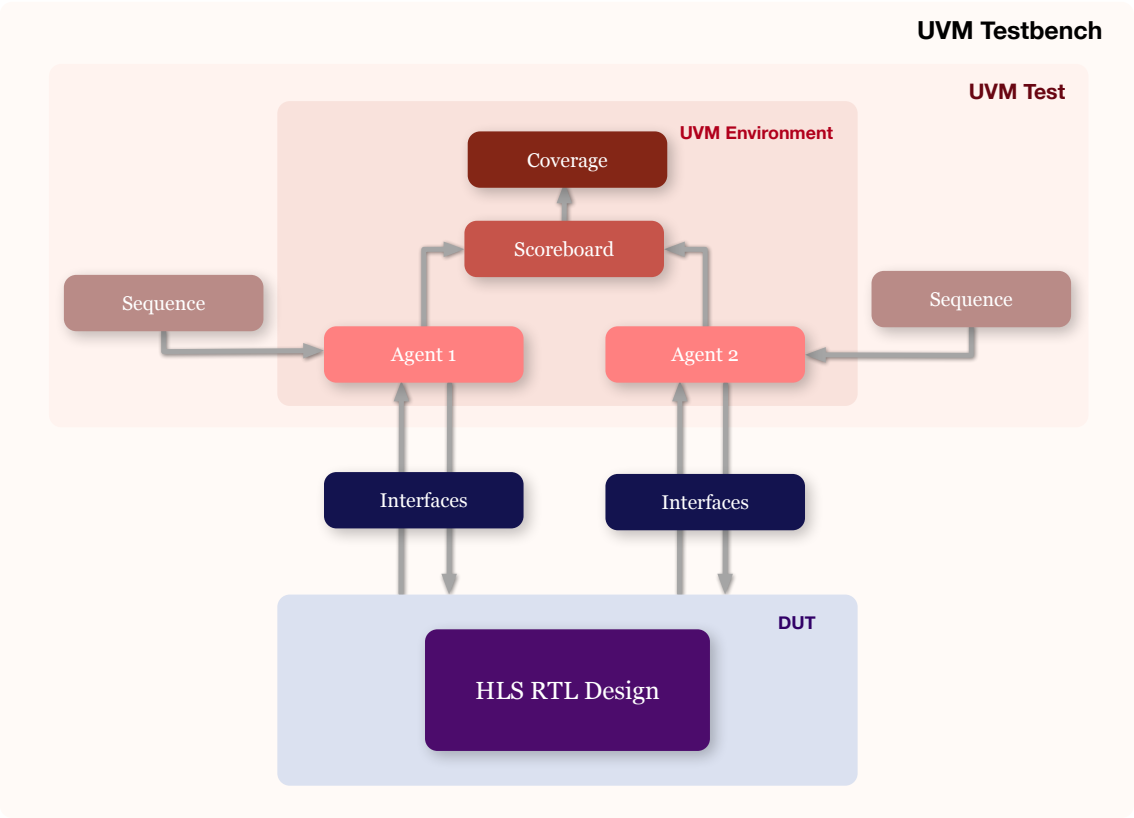

Figure 3.5: HLS RTL design verification environment in phase 2.

Figure 3.5 shows that the previous DUT is replaced with the resulting RTL design in the UVM testbench developed in phase 1.

After the testbench is set up, the phase 1 tests will be re-run to check the equivalence. According to the design specifications, timing features are implemented on existing predictors and Matlab checks.

If the functional coverage and the code coverage cannot reach 100%, then investigations of missing coverage should be carried out. By implementing new tests which cover the missing coverage, or developing a new test plan can be an alternative.

3.3 Integrated design verification environment

Phase 3 can be regarded as an optional stage. If some extra RTL design blocks are added into the HLS RTL design, the new DUT will be regarded as an integrated design including the previous HLS RTL design and extra logic designs.

The DUT in phase 3 is shown in Figure 3.6. The previous HLS RTL design and extra logic designs (e.g. AXI registers) will be packed into an RTL wrapper. Depending on the pinout differences, the verification engineers will decide whether to re-use the existing UVM testbench or not.

Figure 3.6: HLS integrated design DUT in phase 3.

Assuming that we can use the previous testbench, then Figure 3.7 shows the DUT modification in the UVM testbench. An RTL wrapper including the HLS RTL design and the extra logic designs is used as DUT in phase 3. Then, the verification process is similar to the previous phase, in this case only the extra logic designs need to be verified. Standard UVM verification is performed on extra logic. Therefore, the tests will be performed, then the functional coverage and code coverage will be collected.

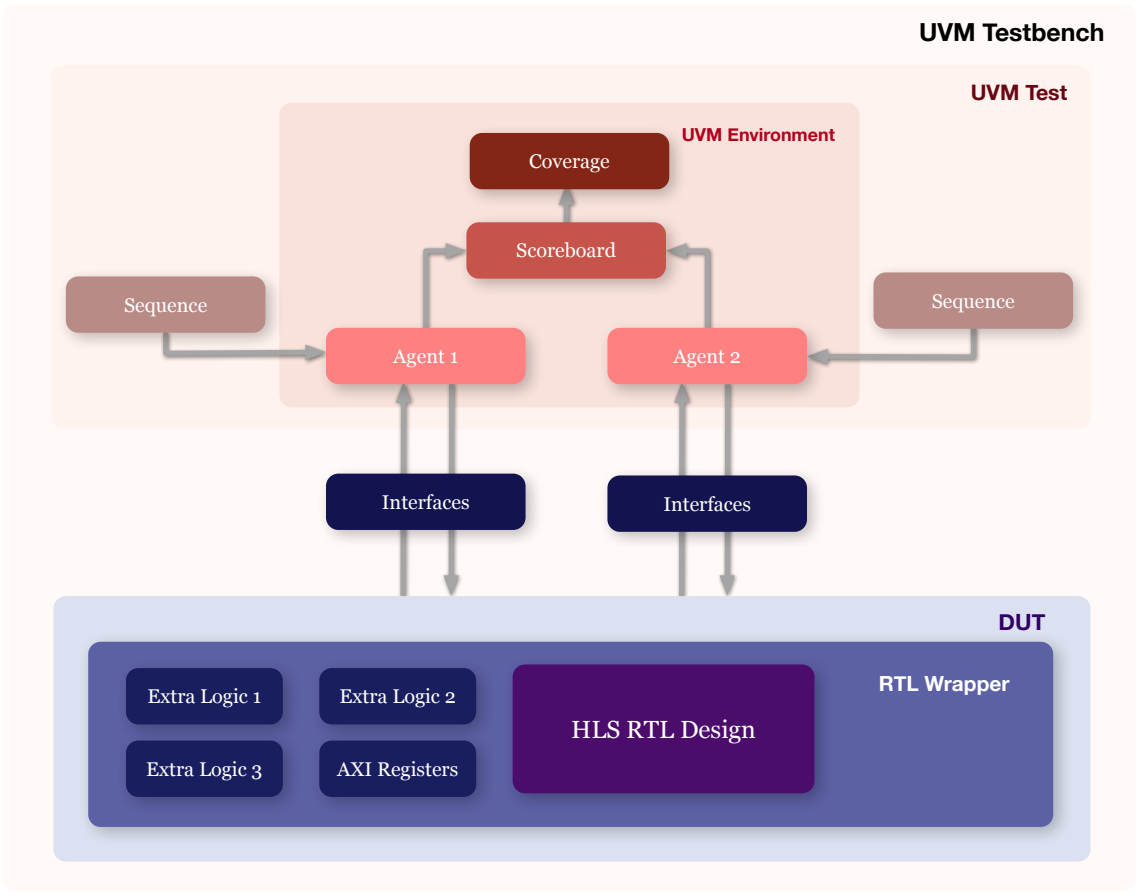

Figure 3.7: HLS integrated design verification environment in phase 3.

4

Implementation of the UVM testbench for "Pterodactyl" DUT

This chapter describes how the UVM testbench for the "Pterodactyl" DUT is implemented. The following sections illustrate the functionality of main components in the UVM testbench structure separately. A special component called SystemC adapter is needed to verify the HLS C++ DUT, which is described in detail in the last section.

UVM consists of extensive class libraries which can be inherited to develop well-organized and highly reusable verification testbenches. The same testbench code is used for both the RTL "Pterodactyl" DUT and the C++ "Pterodactyl" DUT, but the testbench will be compiled into different architectures by parsing a specified DUT type. Our UVM-based verification testbenches for the RTL DUT and the C++ DUT are presented in Figure 4.1 and Figure 4.2 respectively. The architectural difference will be discussed in the following paragraphs.

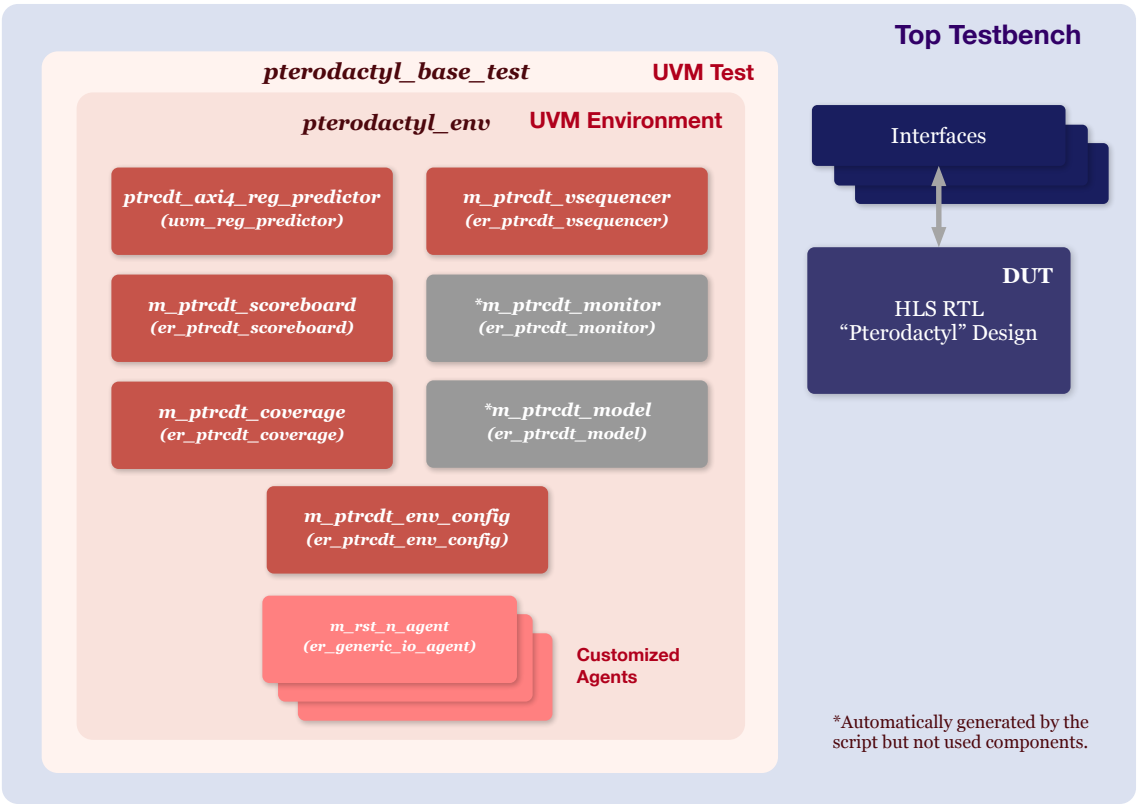

Figure 4.1: Overview of UVM-based testbench for the RTL DUT.

As Figure 4.1 shows, for the RTL "Pterodactyl" DUT, the testbench is built in a traditional way, which means that the testbench has a classic UVM hierarchy, from the test level to various agents from a perspective of top-down. At the top level, the RTL DUT is connected to the testbench through interfaces containing input and output pins in a typical way.

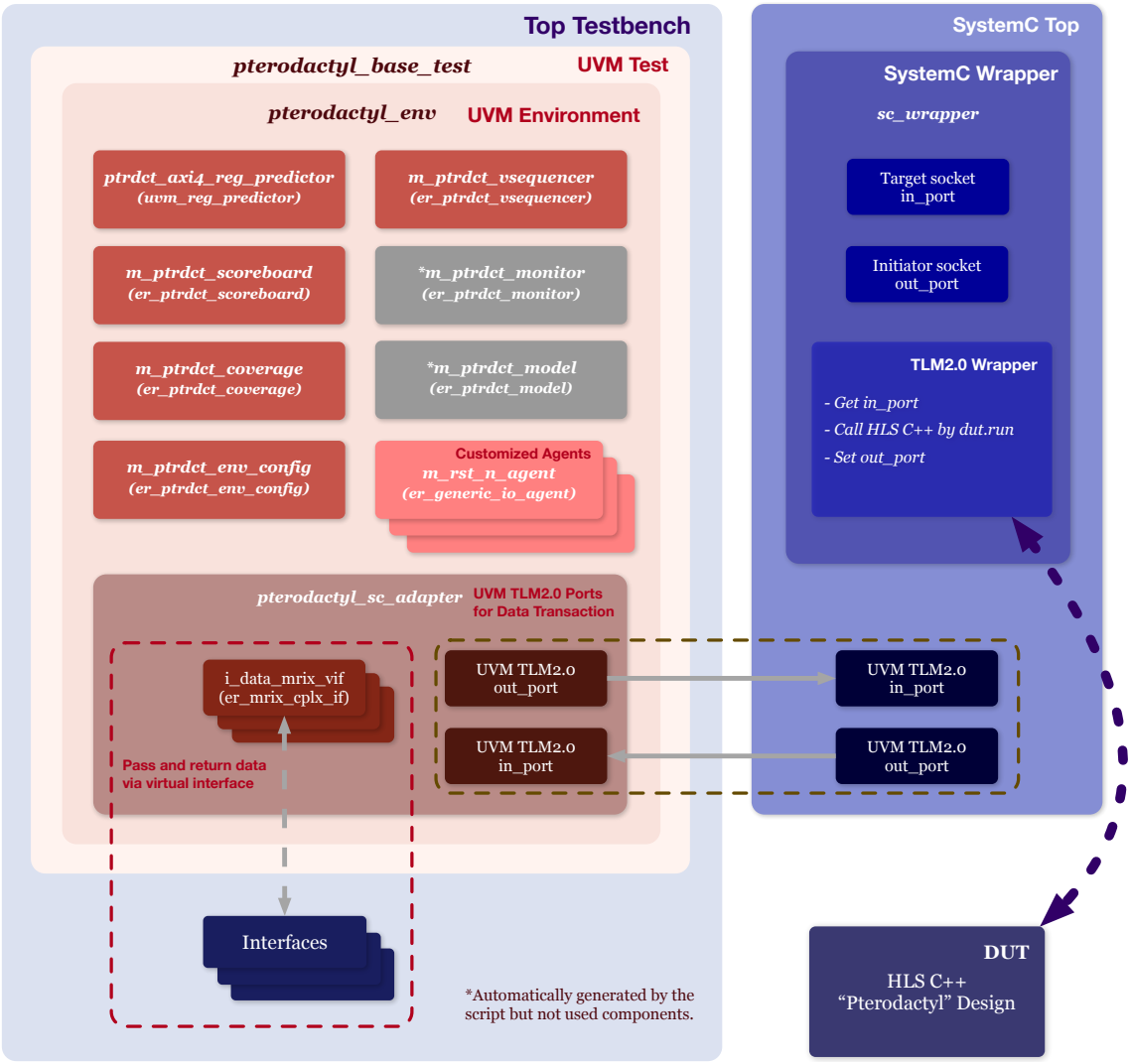

Figure 4.2: Overview of UVM-based testbench for the C++ DUT.

While for the C++ "Pterodactyl" DUT shown in Figure 4.2, besides the fundamental verification components, an additional component called adapter is also created in the verification environment. It functions as a bridge to enable communication and data transaction between the UVM-based testbench and the C++-based "Pterodactyl" DUT via TLM-2.0 ports. The C++ "Pterodactyl" DUT is also wrapped up in a SystemC wrapper. In this case, the UVM-based testbench has no actual connection with the C++ "Pterodactyl" DUT wrapper but they communicate with each other via the TLM-2.0 protocol using the TLM generic payload data type. The data from UVM environment will be encapsulated into a TLM generic payload and

the payload will then be sent to the wrapper and vice versa.

4.1 UVM environment configuration

An environment configuration object is created to configure the entire verification environment. It contains various agent configuration objects because agents play a significant part in the composition of a UVM-based environment. In addition, the structure of an agent is dependent on its configuration and the configuration can differ from one test to the another using a different configuration object for the same agent. For example, an agent can act in active or passive mode. The status of an agent can be configured either in active or passive mode by a parameter in the agent configuration object. Therefore, through configuring the agents contained in the environment by agent configuration objects, the entire environment can be configured by a environment configuration object which consists of various agent configuration objects and other components such as virtual interface and register model.

4.2 Interfaces

In Verilog, the communication between blocks is specified using module ports. SystemVerilog adds the interface construct which encapsulates the communication between blocks. An interface is a bundle of signals or nets through which a testbench communicates with a design.

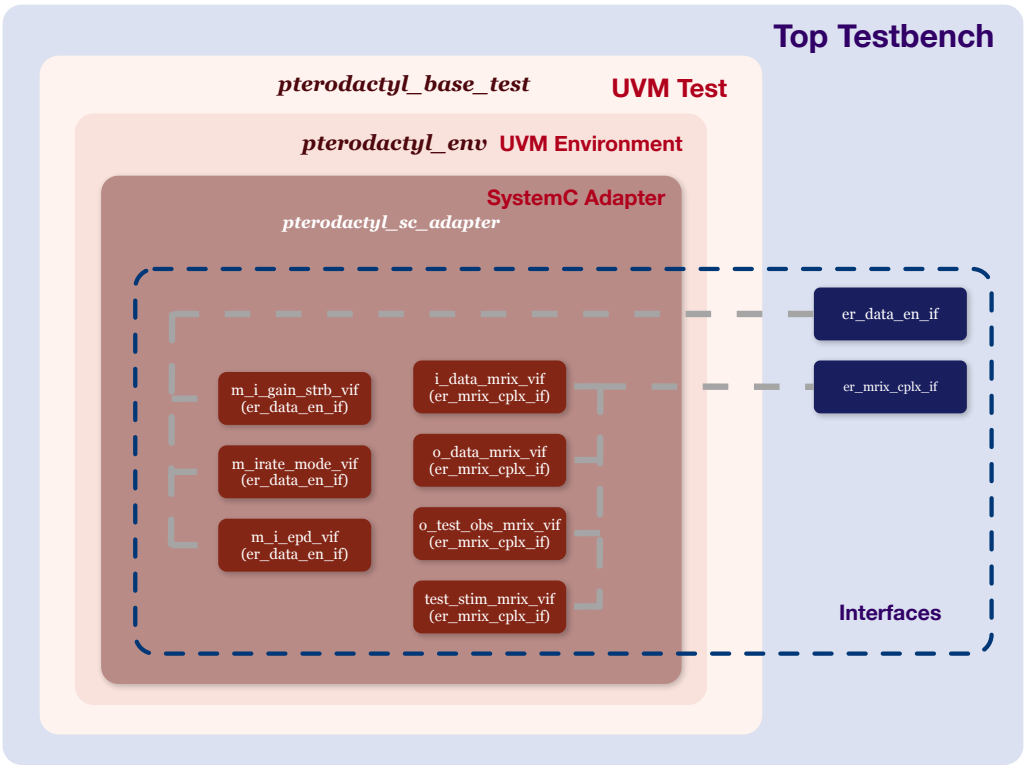

Figure 4.3: Overview of the interfaces in the top testbench.

4.3 Register model

A register model in a UVM verification environment consists of a hierarchy of blocks to simulate the reading and writing DUT registers in a high-level and abstract way. Register models are implemented by a set of standard base class libraries that enable users to implement a object-oriented model to access the DUT registers and memories. Every register in the model corresponds to an actual hardware register in the design.

4.4 Virtual sequencer

A virtual sequencer contains handles to other real sequencers and it is harnessed to control all other sequencers. Generally, an instance of the virtual sequencer will be created and real sequencers will be connected with their respective handles in the virtual sequencer. Commonly a virtual sequence will be operated on the virtual sequencer. Just like a virtual sequencer, a virtual sequence also contains various sequences to be executed on multiple sequencers.

4.5 UVM scoreboard

A UVM scoreboard is a commonly used verification component which is built to verify the functional correctness of the DUT by comparing the DUT output with golden references. It generally receives transaction level objects captured from the interfaces of a DUT through TLM Analysis Ports. After receiving data transactions, it can either implement calculations and produce the expected values or send data objects to a reference model to get expected values. The reference model is applied to emulate the functionality of the design.

Our DUT has a high complexity because it contains many filters so it is difficult to create a reference model using traditional SystemVerilog style. Meanwhile, Matlab is a quite powerful tool to handle this problem due to its extensive toolboxes and built-in functions. Therefore, a Matlab model is designed as the reference model and it will give the expected values. The input data will be written into some files and a Matlab script will read input data from these files and calculate results accordingly. Then the expected results will be written into some other files and the golden references will be extracted from those files and will be compared with the actual values from the interfaces to check the functionality of the DUT. The entire process is shown in Figure 4.4.

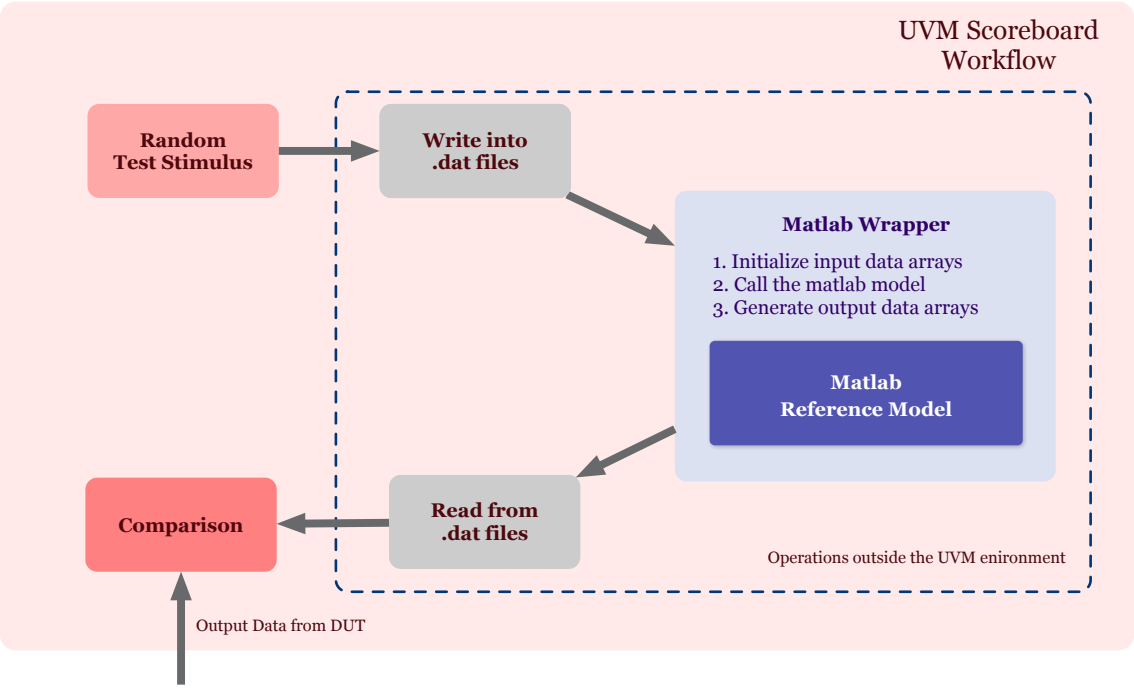

Figure 4.4: Overview of the scoreboard in the "Pterodactyl" verification environment.

4.6 UVM coverage

The coverage collector in the verification environment is utilized to collect the functional coverage. Coverage is a fundamental metric in verification that is used to measure how much of the design taht has been exercised. There are two coverage types, code coverage and functional coverage. Code coverage measures the percentage of code that has been exercised in a simulation. It includes the execution of design blocks, number of lines, conditions, FSM, toggle, etc. Nowadays the simulator tools are clever enough and the code coverage will be extracted automatically. Functional coverage is a user-defined metric that measures how much of the design specification has been exercised in verification. This can be beneficial in a constrained random based simulation to understand what features have been covered by a set of tests in a regression.

4.7 Agents

An agent simply contains a sequencer, a driver and a monitor. It can be configured to operate in active or passive mode. In the active mode, it drives required data to the input ports of DUT and also monitors the output ports of the DUT. In the passive mode, it only monitors the output ports of DUT.

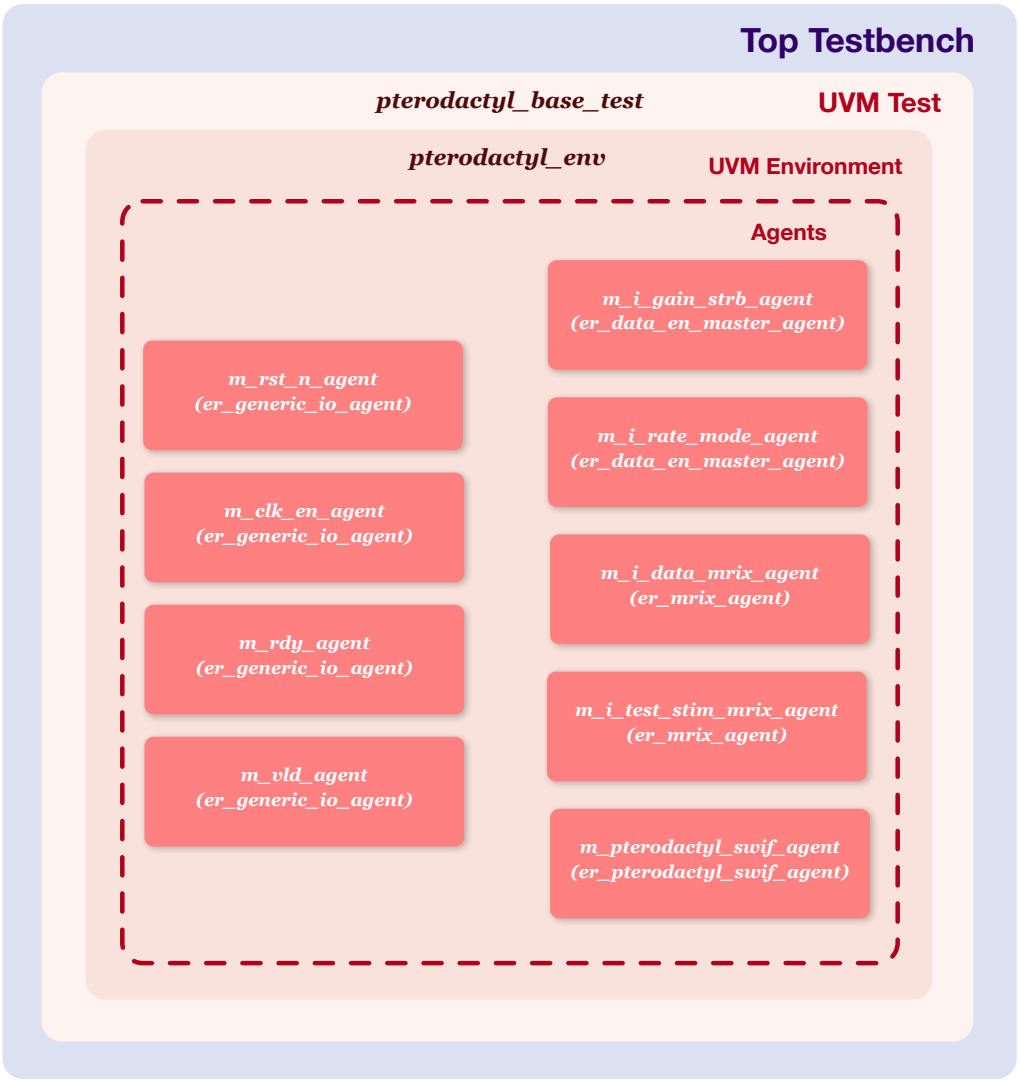

Figure 4.5: Overview of the agents in the UVM environment.

As shown in Figure 4.5, there are four kinds of agents for all the pins of the DUT, which are listed below.

- *er_ptrdct_generic_io_agent:* This kind of agent is created for pins like ready or valid signals, i.e., general and common input and output signals.
- *er_data_en_master_agent:* This kind of agent is also created for pins.
- *er_mrix_agent:* This agent is used to drive major data stimuli to the DUT.
- *er_ptrdct_swif_agent:* This kind of agent is created to configure the DUT. It can drive configuration data to the DUT so the DUT can be configured accordingly.

For example, the structure of one of the agents called `m_i_data_mrix_agent` whose type is `er_mrix_agent` is shown in Figure 4.6.

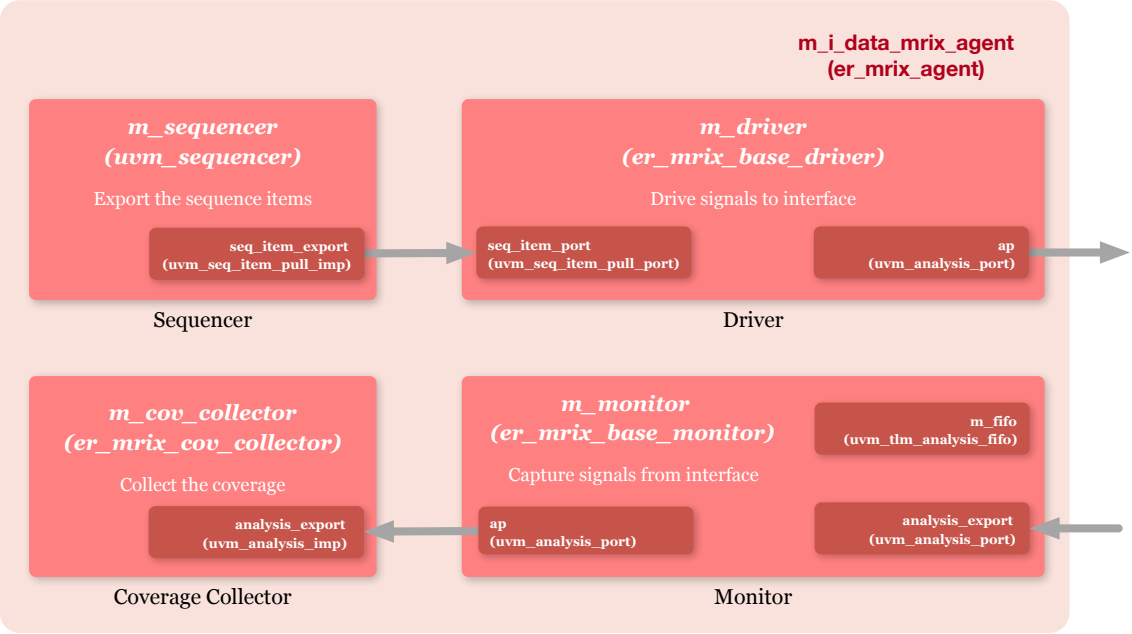

Figure 4.6: Structure of `m_i_data_mrix_agent`.

4.8 SystemC adapter

An additional component adapter will instantiated in the verification environment when the C++ DUT is compiled and simulated. The overview of the SystemC adapter is presented in Figure 4.7.

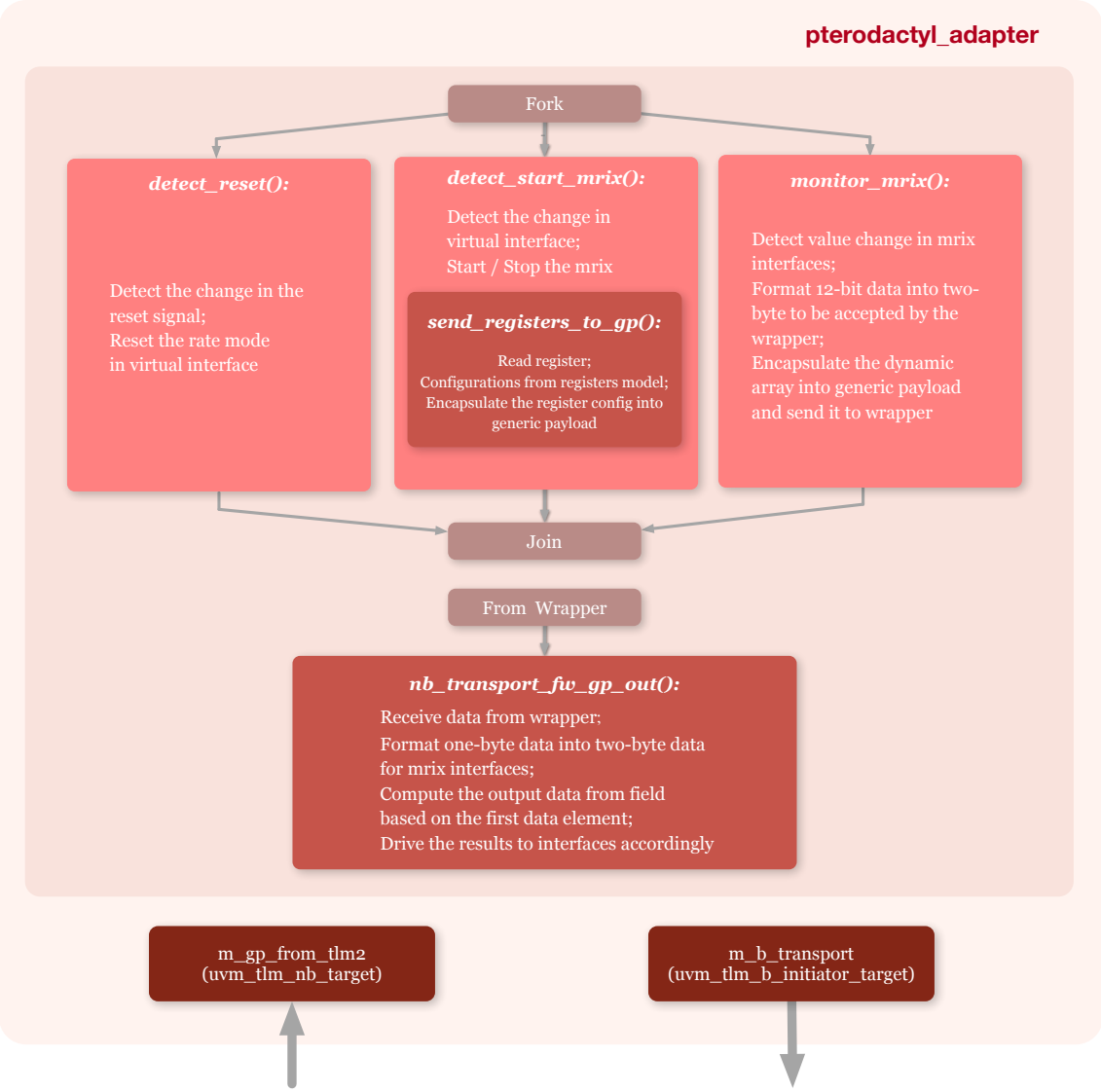

Figure 4.7: Overview of the adapter.

The SystemC adapter functions as a bridge between the UVM-based verification environment and the C++ DUT which is wrapped up in a wrapper. An overview of the wrapper is shown in Figure 4.8.

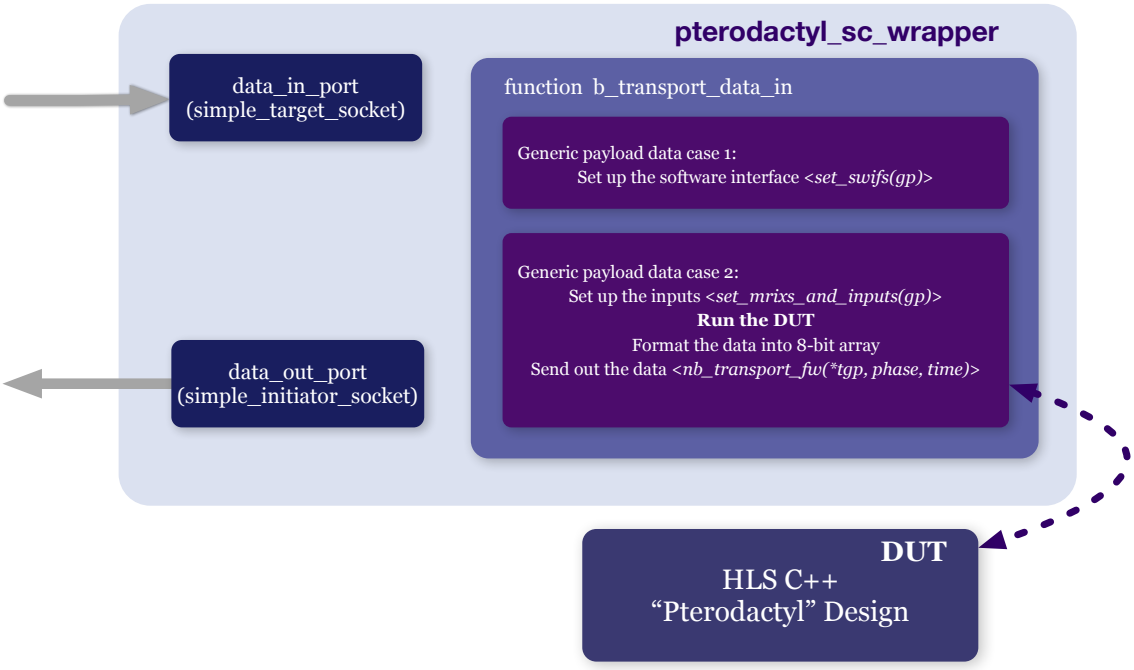

Figure 4.8: Overview of the wrapper.

Unlike the traditional connection style between the testbench and DUT, C++ DUT and UVM-based testbench cannot be connected through input and output pins. In our case, there is no actual connection between testbench and DUT.

Nevertheless, the TLM-2.0 protocol will be adopted for the communication between the adapter and the wrapper. The wrapper is not included in the verification environment but is at the same hierarchical level as the verification environment because it will call the DUT when needed. To be more specific, the transaction between the adapter in the verification environment and the wrapper that is responsible for calling the DUT is presented in Figure 4.9.

The fundamental data unit for the communication process between the C++ DUT and the verification environment is called a generic payload, which is a dedicated data type for TLM-2.0 communication. The adapter extracts input data from input interfaces and the extracted data will be reformatted into generic payload to be sent to the wrapper via the TLM-2.0 protocol. The wrapper will receive the input data from a generic payload and call the C++ DUT to get the output data from the DUT. Then the output data will be formatted into generic payloads and sent back to the adapter. Then the output data will be decoded into arrays and assigned to the output interfaces so the actual result from the DUT could be compared to the expected result in the scoreboard.

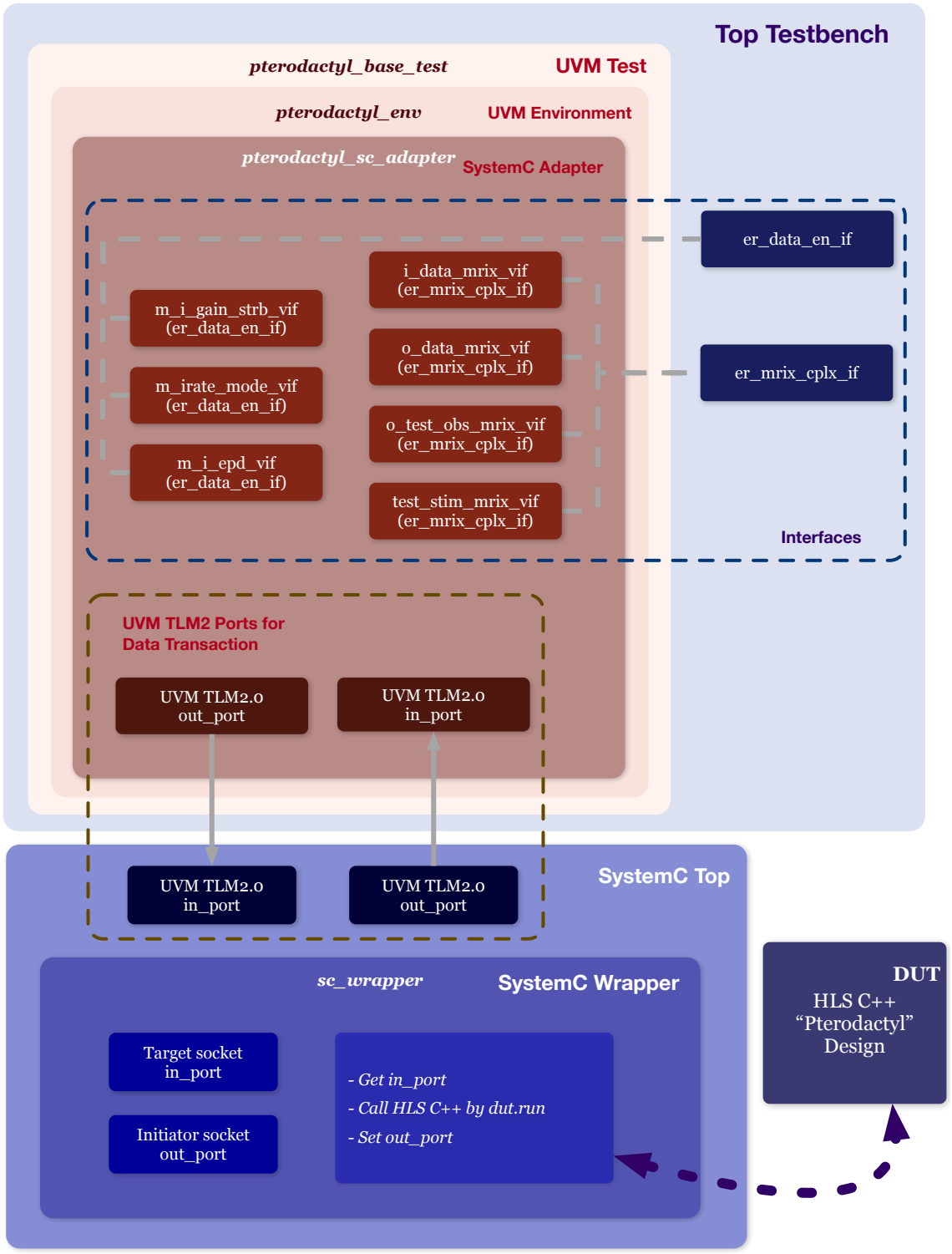

Figure 4.9: Communication of the adapter and the wrapper.

5

HLS verification strategy in Cadence environment

This chapter describes how the HLS verification flow is modified based on the existing flow in Cadence environment. An existing Cadence-provided shell script can do the compilation, simulation and regression test. We modified and debugged the framework of the cross-language data transaction so that the flow is specialized for HLS verification.

5.1 Compilation in Cadence Incisive Simulator

The main compilation bash script named "`incisive.sh`" for Cadence Incisive Simulator is provided by software support engineers from Cadence.

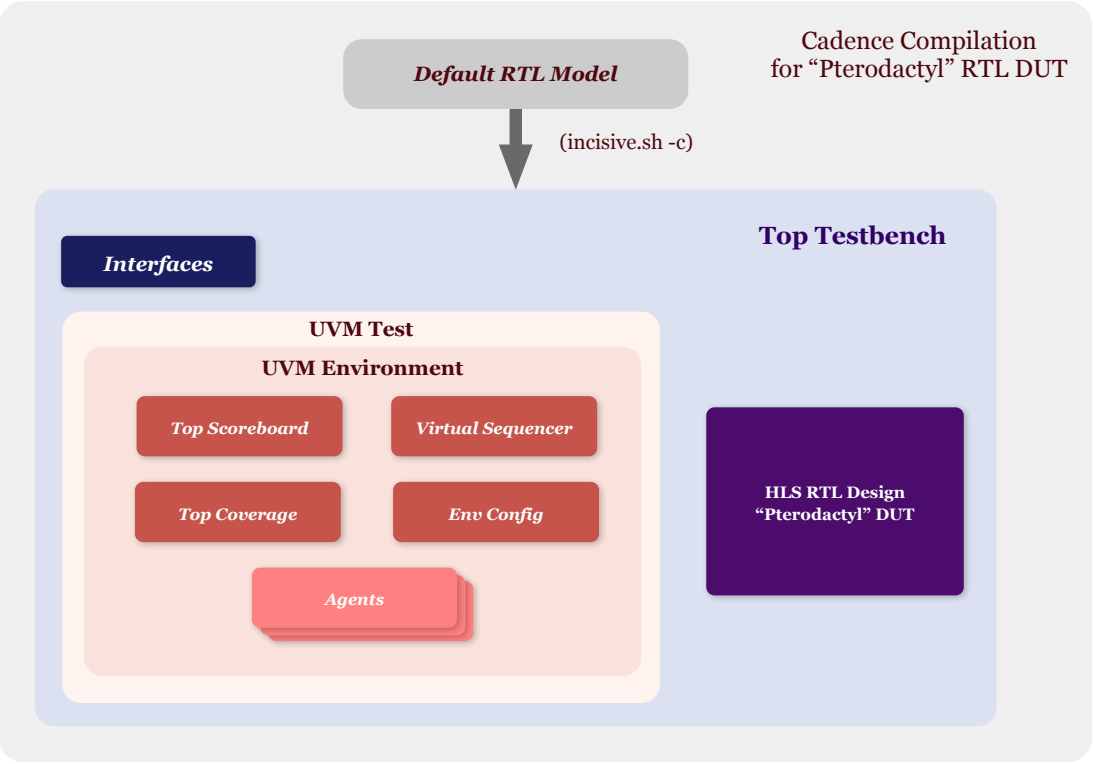

Figure 5.1: Compilation in Cadence Incisive Simulator, for "Pterodactyl" RTL DUT.

To reduce the amount of redundant code, the compilation script is able to parse different arguments in the command line to compile different environments with the same testbench code.

When using "`incisive.sh -c`", the RTL design is compiled. As shown in Figure 5.1, a top testbench is created for "Pterodactyl" RTL DUT. Another argument "`--COVERAGE`" can be added into the command for activating coverage collection.

When using "`incisive.sh -c HLS_CPP`", the HLS C++ design is compiled. As shown in Figure 5.2, a SystemC adapter is created in the SystemVerilog UVM environment and a SystemC top which includes a SystemC wrapper will be generated. The SystemC wrapper will call the "Pterodactyl" C++ DUT.

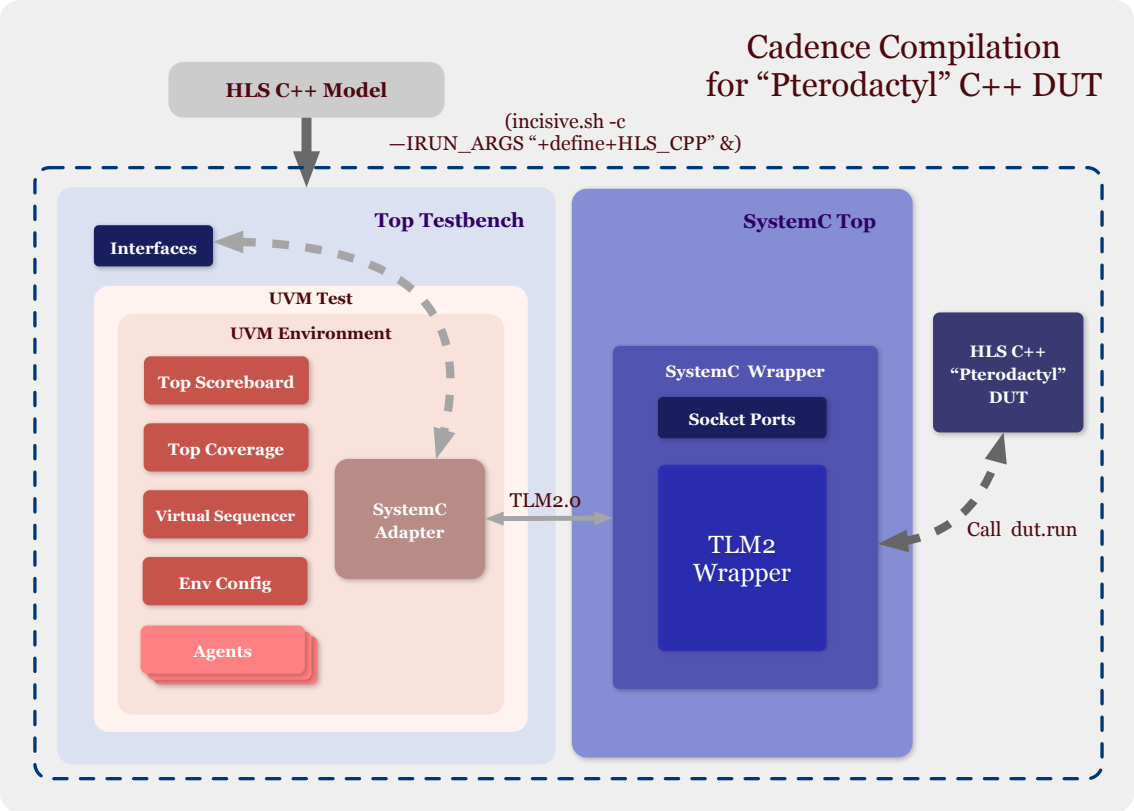

Figure 5.2: Compilation in Cadence Incisive Simulator, for "Pterodactyl" C++ DUT.

5.2 Application of UVM-ML library

Verification projects usually involve a number of verification components which are implemented in different languages such as SystemVerilog and SystemC and based on different verification methodologies like UVM. The multi language (ML) library provides a method to reuse these components with minimal modifications. The

UVM-ML library is based on UVM methodology.

There are two frameworks, UVM-SystemVerilog and UVM-SystemC that are required to enable UVM-ML functionality. In addition, UVM-ML supports data communication based on the TLM-1 and TLM-2.0 interfaces [23]. In our case, the UVM-ML library is utilized to enable the TLM-2.0 communication in a multi-language environment.

A function called `uvm_ml::ml_tlm2#()::register()` is used to register ports so that a TLM port in SystemVerilog framework could be connected to a port in SystemC framework. The usage of the function is demonstrated in Source Code 5.1.

Source Code 5.1: UVM-ML library top-level application, registration process in SystemVerilog side.

SystemVerilog adapter registration process with UVM-ML.

```
1  // register the ML ports and sockets
2  function void phase_ended(uvm_phase phase);
3    if (phase.get_name() == "build") begin
4      uvm_ml::ml_tlm2#()::register(tlm_initiator_socket);
5      uvm_ml::ml_tlm2#()::register(tlm_target_socket);
6    end
7  endfunction
```

Moreover, the function `uvm_ml::connect()` is used to connect TLM ports in different frameworks, which is shown in Source Code 5.2.

Source Code 5.2: UVM-ML library top-level application, connection process in SystemVerilog side.

SystemVerilog adapter connection process with UVM-ML.

```
1  function void connect_phase(uvm_phase phase);
2      super.connect_phase(phase);
3      ...
4      uvm_ml::connect(tlm_initiator_socket.get_full_name(),
       ↪  "sctb.sc_wrapper_inst.data_in_port");
5      uvm_ml::connect("sctb.sc_wrapper_inst.data_out_port",
       ↪  tlm_target_socket.get_full_name())
6      ...
7  end function
```

Besides, function `uvm_ml::synchronize()` is applied in the `reset_phase(uvm_phase phase)` and `main_phase(uvm_phase phase)` of UVM base test to provide an explicit synchronization mechanism between the SystemVerilog master framework and

the SystemC slave framework.

In the SystemC wrapper file, UVM-ML is also exploited. Function `ML_TLM2_REGISTER_TARGET()` and `ML_TLM2_REGISTER_INITIATOR()` is used to register SystemC TLM-2.0 sockets for ML connections. The initiator and target sockets for a TLM-2.0 transaction must be registered to enable a multi-language connection [23]. Finally, function `NCSC_MODULE_EXPORT()` will export SystemC top to SystemVerilog UVM test top using DPI features.

Source Code 5.3: UVM-ML library top-level application, SystemC side.

SystemC wrapper top-level structure in Incisive.

```
#include "uvm_ml.h"
using namespace uvm_ml;
#include "sc_wrapper.h"

SC_MODULE(sctb) {
public :
  sc_wrapper sc_wrapper_inst;
  SC_CTOR(sctb) : sc_wrapper_inst("sc_wrapper_inst")
  {
      ML_TLM2_REGISTER_TARGET(sc_wrapper_inst,
       ↪ tlm_generic_payload, data_in_port, 32);
      ML_TLM2_REGISTER_INITIATOR(sc_wrapper_inst,
       ↪ tlm_generic_payload, data_out_port , 32);
  }
};
NCSC_MODULE_EXPORT(sctb)
```

To realize the conditional compilation as mentioned in the previous section, if `HLS_CPP` is specified in the command line, then an SystemC adapter will be instantiate in the UVM environment, which is shown in the Source Code 5.4. Otherwise, by default, there will no SystemC adapter in the UVM environment because the HLS RTL design is used as DUT.

Source Code 5.4: SystemC adapter instantiation in Cadence UVM environment.

SystemC adapter instantiation by ifdef.

```
`ifdef HLS_CPP
  // Build SC adapter
  sc_adapter =
   ↪ ptrdct_sc_adapter#(.gp_type(uvm_tlm_generic_payload))::\
  type_id::create("sc_adapter",this);
`endif
```

Further more, in the top test, if the `HLS_CPP` is not specified, the HLS RTL DUT

will be instantiated by default as shown in the Source Code 5.5.

Source Code 5.5: SystemC adapter instantiation in Cadence UVM environment.

SystemC adapter instantiation by "ifndef HLS_CPP".

```
`ifndef HLS_CPP
pterodactyl ptrdct_str_hls_inst(
    .clk(clk),
    .clk_en(clk_en),
    .arst_n(rst_n),
    ...
);
`endif
```

A function from UVM-ML package is used to call both SystemC top and SystemVerilog top at the same time when `HLS_CPP` is specified in conditional compilation. The usage of the run test function is shown in the Source Code 5.6.

Source Code 5.6: The uvm run test function in top testbench, in Cadence.

SystemC top conditional compilation by "ifdef HLS_CPP".

```
// Start UVM test environment
initial begin
    string tops[1];
    $timeformat(-9, 0, " ns", 5);
    `ifdef HLS_CPP
      tops[0] = "SC:sctb";
    `endif
    uvm_ml_run_test(tops, "");
end
```

5.3 Simulation in Cadence Xcelium Simulator and regression in Cadence VManager

After the simulation is done, we can select some signals of interest to visually check the functional correctness in Xcelium waveform window as shown in Figure 5.3. If error occurs, then we can localize the corresponding source code and add some display statements to scrutinize if the displayed results are expected. Furthermore, there is a more straightforward and convenient way to accomplish the verification. In our UVM-based verification environment, we can take full advantage of the built-in UVM reporting macros. For example, in the scoreboard, if there is a mismatch between the actual output and the expected output then we can utilize macro `uvm_error` to report an error in the transcript. To able code and functional coverage collection, an additional argument `-COVERAGE` should be provided when compiling. To

simulate and enable the functional coverage collection at the same time, two addidtional arguments should be provided as well such as "`incisive.sh -r --TEST test_stim_test --IRUN_ARGS '+ENABLE_FUNC_COV' --SV_SEED random`". In this example, argument `--IRUN_ARGS '+ENABLE_FUNC_COV'` tells the simulator to collect the functional coverage and `--SV_SEED random` provides a seed for randomization. The simulation result is presented in Figure 5.3.

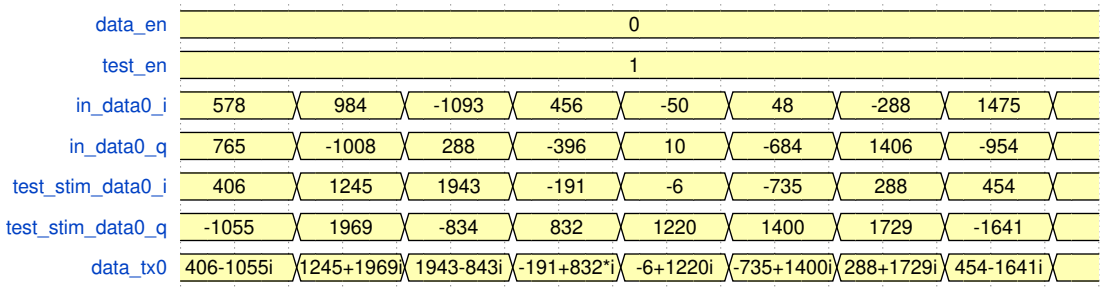

Figure 5.3: Simulation result in Cadence Simulator.

6

HLS verification strategy in Siemens Mentor Graphics environment

This chapter describes the entire HLS verification flow we perform to migrate from Cadence environment to Siemens Mentor Graphics environment. Based on a unified testbench for both HLS C++ design and HLS RTL design, the flow consists of the compilation of source code and testbench, the simulation for the selected test, and the coverage collection.

A complex shell script, named `mentor.sh`, including a series of filelists is developed to compile either HLS C++ design or HLS RTL design by parsing arguments in command line. Both the SystemVerilog adapter in UVM environment and the SystemC wrapper in SystemC top are re-developed according to the multi-language features provided by Siemens Mentor Graphics QuestaSim. The optimization and simulation scripts are merged into `mentor.sh`, so that the `mentor.sh` script can run the specified UVM test. Two Siemens Mentor Graphics EDA tools are used in this flow, one is Siemens Mentor Graphics QuestaSim 2021.1 and the other is Siemens Mentor Graphics Visualizer 2021.1. The compilation and optimization are done by QuestaSim, while the simulation and coverage collection are done in Visualizer.

6.1 Compilation in Siemens Mentor Graphics QuestaSim Simulator

Since there are no ready-to-use Siemens Mentor Graphics simulation scripts for project "Pterodactyl" in Ericsson, all the bash and tool command line (TCL) scripts from compilation to simulation should be developed.

In the QuestaSim verification, the conventional way is to use the three-step flow, which consists of three commands for compilation, optimization and simulation. In the first step, the design and the testbench is compiled by using `vlog`, `vcom` or `sccom`. Depending on which hardware description language is used, the corresponding command is selected. `vlog` is used for compiling VHDL modules, `vlog` is used for compiling Verilog modules and `sccom` is used for compiling SystemC modules. In the second step, the compiled design is optimized by using `vopt`. The optimized output of `vopt` can be reused for multiple simulation runs. In the third step, the

simulation is carried out by the `vsim` command.

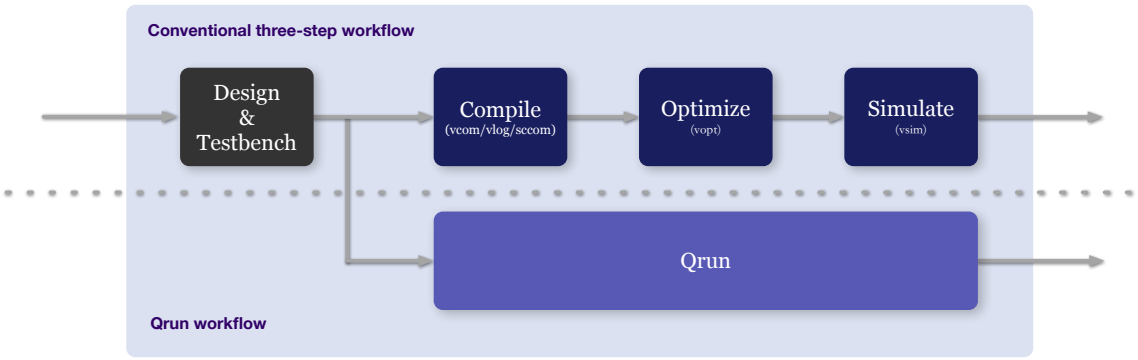

Figure 6.1: New Siemens Mentor Graphics QuestaSim simulation flow with `qrun` command.

After the release of QuestaSim 2019.4, a new command called `qrun` is introduced to TCL script to reduce the amount of code and merge the compile, optimize, and simulate functions into a single step, which is shown as the Figure 6.1. Although `qrun` reduces the amount of codes, it is inevitable to use conventional QuestaSim commands because `qrun` is still under development and cannot cover all the features.

Siemens Mentor Graphics QuestaSim currently does not support compiling the SystemVerilog files in an automatically generated order. Since the hierarchical architecture of the "Pterodactyl" verification environment is complex, a series of filelists are created from bottom to top. Eight filelists are created in a folder named `filelists` under the parent folder for the UVM verification environment.

The following code in Source Code 6.1 is an example of creating a library in the `basic.qrun.files`. The library is formatted in `-reflib/-makelib/-end` framework. The package file is listed first, then the related SystemVerilog headers and files in the folder are added by `+incdir+`.

Source Code 6.1: Register model library.

Register model library compilation commands in QuestaSim.

```
1  -reflib  pterodactyl_reg_model_lib
2  -makelib pterodactyl_reg_model_lib
3  /<PROJECT_PATH>/tb_uvm/lib/env/ \
4  pterodactyl_reg_model/pterodactyl_reg_model_pkg.sv
5  +incdir+/<PROJECT_PATH>/tb_uvm/src
6  +incdir+/<PROJECT_PATH>/tb_uvm/lib/env/pterodactyl_reg_model
7  -end
```

To realize the multi-language compilation between SystemC and SystemVerilog, the C++ compiler provided by QuestaSim is used to compile the SystemC wrap-

per related code. The following code in Source Code 6.2 is created for SystemC wrapper compilation. `-DSC_` related arguments specify the SystemVerilog libraries for SystemC DPI. `-DSC_INCLUDE_MTI_AC` enables debug support for Algorithm-C datatypes. `-DSC_INCLUDE_DYNAMIC_PROCESSES` enables dynamic processes. `-DMTI_BIND_SC_MEMBER_FUNCTION` enables registration of module member functions as DPI-SC imports. `-uvmc` clarifies using UVM-Connect library. `-sctop sc_main` claims the `sc_main` top is the SystemC top and will be exported to QuestaSim working library. After being exported, the SystemC top can be recognized from SystemVerilog side.

Source Code 6.2: SystemC wrapper filelist.

SystemC wrapper compilation commands in QuestaSim.

```
-sysc
-DSC_INCLUDE_MTI_AC -DSC_INCLUDE_DYNAMIC_PROCESSES -DQUESTA
  ↪  -DMTI_BIND_SC_MEMBER_FUNCTION -g
-DCATAPULT
-I/<COMMEN_IP_DIRECTORIES>
-I/<PROJECT_PATH>/tb_uvm/src/pterodactyl_sc_wrapper/
-I/<MENTOR_TOOLS_PATH>/ \
mentor/catapult/10.6/Mgc_home/shared/include/
-I/<MENTOR_TOOLS_PATH>/ \
mentor/questasim/2021.1/questasim/verilog_src/uvmc-2.3.2/  \
src/connect/sc
/<PROJECT_PATH>/tb_uvm/src/ \
pterodactyl_sc_wrapper/pterodactyl_sc_wrapper.cpp
-sclink
-uvmc
-sctop sc_main
-nodebug
-end
```

Table 6.1: Compilation filelists for C++ and RTL DUTs.

	HLS RTL DUT Compilation	**HLS C++ DUT Compilation**
Command	mentor.sh -c HLS_RTL	mentor.sh -c
Required Filelists	basics.qrun.files pterodactyl_env.qrun.files pterodactyl_generics.qrun.files dut.qrun.files pterodactyl_seq.qrun.files pterodactyl_tc.qrun.files pterodactyl_tb.qrun.files	basics.qrun.files pterodactyl_env.qrun.files pterodactyl_generics.qrun.files pterodactyl_sc_wrapper.qrun.files pterodactyl_seq.qrun.files pterodactyl_tc.qrun.files pterodactyl_tb.qrun.files

`mentor.sh` calls the filelists and compile them in the specified order. The left col-

umn of Table 6.1 lists the required filelists for compiling HLS RTL DUT, the corresponding command is `mentor.sh -c HLS_RTL`. The right column of Table 6.1 lists the required filelists for compiling HLS C++ DUT, the corresponding command is `mentor.sh -c`. All the filelists for "Pterodactyl" are created as shown in Table 6.2.

Table 6.2: Names and contents of the `qrun` filelists.

Names of the filelists	**Files included in the filelist**
basics.qrun.files	· *Universal verification components for clock reset, data enable, mrix, generic_io and software interfaces.* · *Common library for global variables.* · *Register model library for emulating AXI behaviour.*
pterodactyl_env.qrun.files	· *Siemens Mentor Graphics "UVM Connect" library: uvmc-2.3.2 for SystemVerilog side.* · *Environment package, source code and libraries.*
pterodactyl_generics.qrun.files	· *Generic libraries: common package, pre-defined variables and data types.* · *HDL model wrapper.*
pterodactyl_sc_wrapper.qrun.files	· *SystemC wrapper source code. Siemens Mentor Graphics Catapult 10.6 related directories.* · *Siemens Mentor Graphics "UVM Connect" library: uvmc-2.3.2 for SystemC side.*
dut.qrun.files	· *RTL VHDL code for "Pterodactyl".*
pterodactyl_seq.qrun.files	· *Register model pre-defined variables.* · *Sequence pre-defined variables.* · *Sequence package.* · *Sequence library and environment library related directories.*
pterodactyl_tc.qrun.files	· *Test case interfaces configuration.* · *Test case package and source code.* · *Sequence library and environment library related directories.*
pterodactyl_tb.qrun.files	· *Testbench package.* · *Testbench top.* · *Sequence library and environment library related directories.*

The hierarchical structures for "Pterodactyl" C++ and RTL DUTs are shown in Figure 6.2 and 6.3. Similar to the structures described in previous chapter, the main difference is that the verification environment for C++ DUT has an additional SystemVerilog adapter and a SystemC top to include a SystemC wrapper.

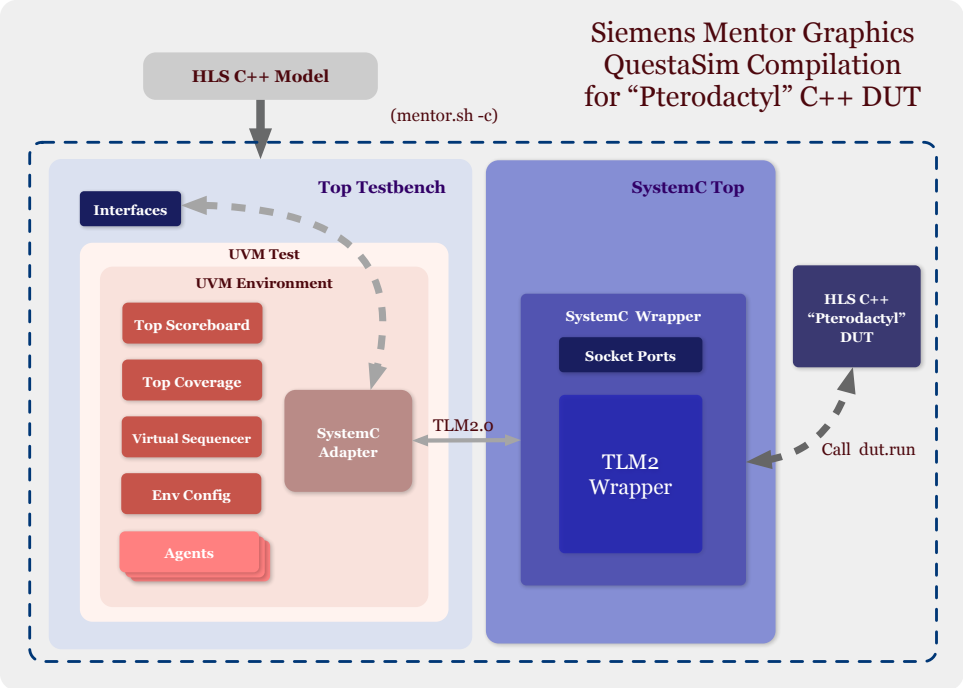

Figure 6.2: Compilation in Siemens Mentor Graphics Simulator, for "Pterodactyl" C++ DUT.

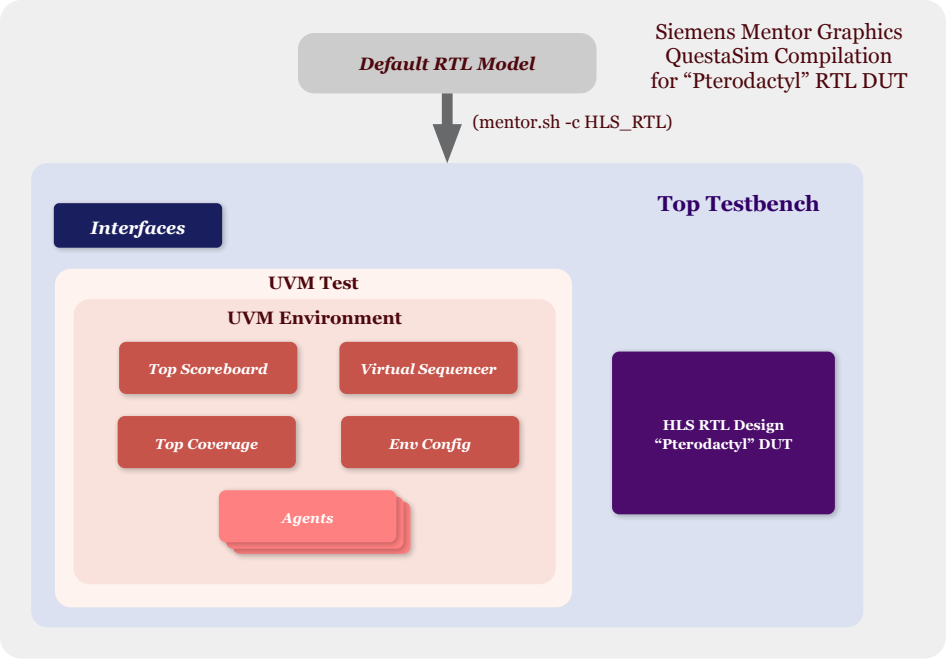

Figure 6.3: Compilation in Siemens Mentor Graphics Simulator, for "Pterodactyl" RTL DUT.

6.2 Re-use of UVM-ML library

In order to reuse the multi-language connection code as much as possible, the first trial is made to test the migration possibility of UVM-ML library in QuestaSim environment. The plan is to test the feasibility of UVM-ML library step by step in hierarchical level. First step is to compile all the SystemC files and export the SystemC top level design to work library. Second step is to compile the SystemC adapter in UVM environment, which is a component written in SystemVerilog.

SystemC wrapper written in SystemC can be compiled with UVM-ML library in QuestaSim, but SystemC adapter written in SystemVerilog cannot be compiled with UVM-ML library in QuestaSim. In the first attempt, UVM-ML library is included directly in the compilation libraries and the directory path of the UVM-ML pre-compiled files. In this method, `uvm-1.1d Built-in` package in default QuestaSim is imported. However, there exists name types difference between `uvm_sv/uvm_ml_phase.svh` in UVM-ML library and `uvm_phase.svh` in default QuestaSim, and between `uvm_sv/uvm_ml_resource.svh` in UVM-ML library and `uvm_resource.svh` in default QuestaSim. Therefore, the missing name types lead to compilation errors.

Source Code 6.3: UVM-ML library re-compilation in QuestaSim.

UVM-ML extra compilation commands in QuestaSim.

```
-dpicppinstall g++ -64 -suppress 2218,2181 \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/tlm2 \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/base \
+incdir. \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/macros \
+incdir$UVM_ML_HOME/ml/adapters/uvm_sv \
-sv +acc -timescale 1ns/1ns \
+define+UVM_ML_PORTABLE +define+UVM_ML_PORTABLE_QUESTA \
$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src/uvm.sv \
$UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv
```

In the second attempt, a package included all the UVM-ML SystemVerilog files and SystemVerilog headers is created for QuestaSim compilation. Therefore, the approach is to compile the UVM-ML library into a portable version in QuestaSim. However, the compiler complains about errors in parsing the types from UVM-ML library files to QuestaSim UVM. The types `macros/uvm_tlm_defines.svh` for TLM transaction in UVM-ML library are not fully supported in QuestaSim TLM implementation file `uvm_tlm_imp.svh`. Therefore, UVM-ML library is has compatibility problem in default UVM environment in QuestaSim 2021.1.

6.3 Application of UVM-Connect library

Since the attempt to reuse UVM-ML library failed, an alternative library, UVM Connect, designed by Siemens Mentor Graphics is adopted because it has better compatibility with EDA tools such as QuestaSim or Visualizer from Siemens Mentor Graphics. The UVM-Connect library provides TLM-1 and TLM-2.0 connectivity and data object passing between SystemC and SystemVerilog models and components. It also provides a UVM Command application programming interface (API) for accessing and controlling UVM simulation from SystemC (or C or C++) [22]. So in order to substitute UVM-ML library with UVM-Connect library, the functions that belong to UVM-ML library should be replaced by the counterparts in UVM-Connect library, and an overview of the connection can be seen on Figure 6.4.

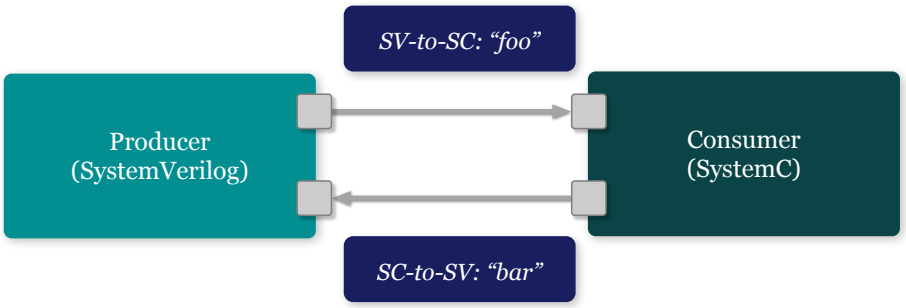

Figure 6.4: The application of UVMC library connection between SystemVerilog and SystemC.

For instance, the port connections in SystemC side are made by registering any TLM port or interface using the `uvmc_connect()` function, which is demonstrated in the Source Code 6.4. It provides a really easy and intuitive way to bind SystemC TLM-2.0 ports and the `uvmc_connect()` function is able to automatically figure out whether the provided port is an initiator port or a target export. Eventually, function `sc_start(-1)` is called to start SystemC [22].

Source Code 6.4: UVMC library top-level application, SystemC side.

The structural application of UVMC functions for SystemC side.

```
1  #include <uvmc.h>
2  using namespace uvmc;
3  #include "sc_wrapper.h"
4
5  int sc_main(int argc, char *argv[]) {
6     sc_wrapper wrapper_INST("wrapper_INST");
7     uvmc_connect<>(wrapper_INST.data_in_port,"data_in_port");
8     uvmc_connect<>(wrapper_INST.data_out_port,"data_out_port");
9     sc_start(-1);
10    return 0;
11 };
```

Furthermore, for the SystemVerilog side shown in the Source Code 6.5, the function in UVM-Connect called `uvmc_tlm#()::connect()` is used to replace functions `uvm_ml::ml_tlm2#()::register()` and `uvm_ml::connect()`. Because the function is able to register and connect any type of TLM-2.0 port or interface for connection across the language boundary at the same time. In addition, UVM-Connect has inherent synchronization mechanisms so such functions like `uvm_ml::synchronize()` are no longer needed.

Source Code 6.5: UVMC library top-level application, SystemVerilog side.

The structural application of UVMC functions for SystemVerilog side.

```
1  import uvm_pkg::*;
2  import uvmc_pkg::*;
3
4  function void connect_phase(uvm_phase phase);
5      super.connect_phase(phase);
6      ...
7      uvmc_tlm #(uvm_tlm_generic_payload,uvm_tlm_phase_e)::
8      connect(this.data_out_port,"data_out_port");
9      uvmc_tlm #(uvm_tlm_generic_payload,uvm_tlm_phase_e)::
10     connect(this.data_in_port,"data_in_port");
11     ...
12 endfunction: connect_phase
```

In the current implementation, the base data type generic payload is used for data transaction, which is `uvm_tlm_generic_payload`.The default data transaction function mechanism is shown in Figure 6.5, in which a typical producer and consumer is connected across the language boundary by UVM Connect library.

The ports and exports are connected in native language environments (SystemVerilog and SystemC) prior to their communication across the language boundary. The target socket proxy is acting as the target for the producer's initiator socket. The initiator socket proxy, used on the SystemC side, serves as the source of transaction data to the consumer's target socket.

When the SystemC side is set up, the initiator socket will wait for bits to arrive on, from SystemVerilog side. Therefore, the SystemC side keeps idle before the SystemVerilog side starts.

When the SystemVerilog side starts, the start of UVM lets the producer's run start to produce transactions. The transactions are going to be emitted out the producer's initiator socket, forwarding to the target proxy socket. The proxy socket first converts the incoming transaction using the `do_pack()` method of the converter. The converter converts the transaction object into bits, after which the proxy socket will send those bits across the language boundary by using standard DPI. The act of sending bits across the language boundary will cause the wait of the bits. After

the bits arrive, the thread in SystemC side will wake up and be scheduled for resumption on the first blocking call on the SystemVerilog side. Therefore, once the SystemVerilog side has sent the bits across in a non-blocking way, it will be waiting for an event that indicates the return bits.

Moving back to the SystemC side, the transaction bits have been received at this moment. First, a new object will be created which represents the data transaction. Then the converter is used to unpack the bits that have been received into the new object via `do_unpack()` method. When the transaction object is reconstituted, the object will call blocking transport through the initiator socket, so that the object can be forwarded to the target socket for execution. Once the consumer returns from `b_transport()` the transaction is completed.

Finally, the potentially modified data transaction are converted into bits by using `do_pack()` method. Once the bits are converted, `send(bits,id)` is called to send the bits to free the waiting SystemVerilog side. In the SystemC side, the process will be looped back to wait for new bits for the next transaction. In the SystemVerilog side, the converter will unpack and send the bits, returning from the blocking transport call back to the original producer.

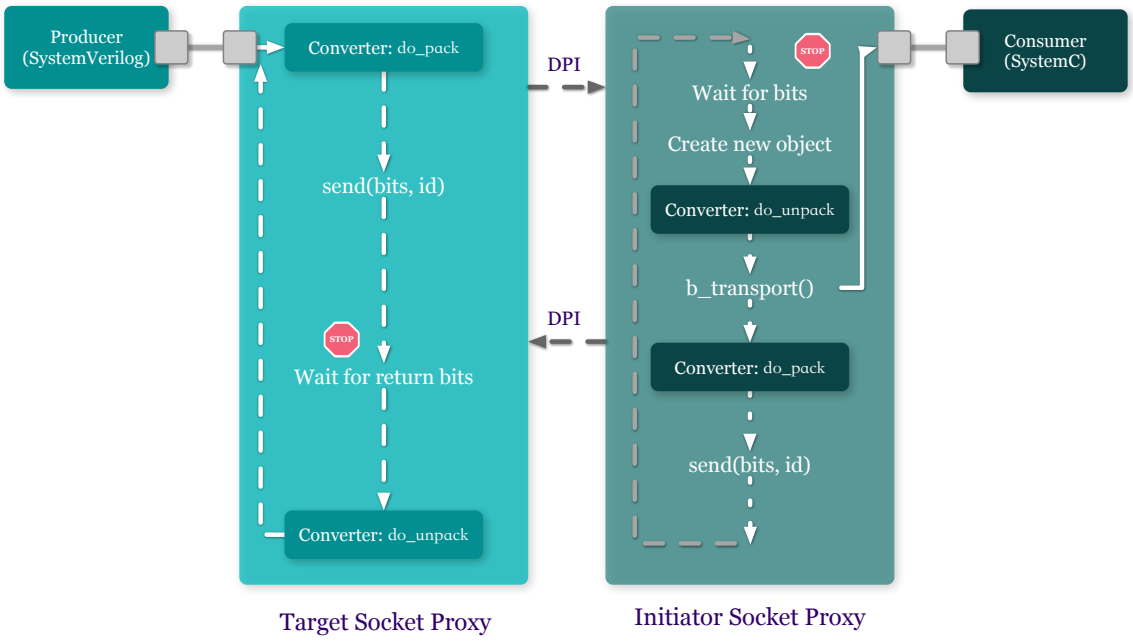

Figure 6.5: The default data transaction function in UVMC library.

If in further implementation a specified data transaction type is required, a customized UVMC transaction converter should be developed for both SystemVerilog side and SystemC side. The approach is similar to the default implementation with `do_pack()` and `do_unpack()` processes for bitwise conversion.

Similarly, to realize the conditional compilation, if `HLS_RTL` is not specified in the command line, then an SystemC adapter will be instantiate in the UVM environment, which is shown in the Source Code 6.6. Otherwise, by adding `HLS_RTL` in the command line, there will no SystemC adapter in the UVM environment because the HLS RTL design is used as DUT.

Source Code 6.6: SystemC adapter instantiation in Mentor UVM environment.

SystemC adapter instantiation by "ifndef HLS_RTL".

```
1  `ifndef HLS_RTL
2    // Build SC adapter
3    sc_adapter =
      ↪  ptrdct_sc_adapter#(.gp_type(uvm_tlm_generic_payload))::\
4    type_id::create("sc_adapter",this);
5  `endif
```

Further more, in the top test, if the `HLS_RTL` is specified, the HLS RTL DUT will be instantiated by default as shown in the Source Code 6.7.

Source Code 6.7: SystemC adapter instantiation in Mentor UVM environment.

SystemC adapter instantiation by "ifdef HLS_RTL".

```
1  `ifdef HLS_RTL
2  pterodactyl ptrdct_str_hls_inst(
3      .clk(clk),
4      .clk_en(clk_en),
5      .arst_n(rst_n),
6      ...
7  );
8  `endif
```

The ordinary function `run_test()` can be used to call both SystemC top and SystemVerilog top at the same time in UVMC library, since `sc_main` is exported to the work library automatically. The usage of the run test function is shown in the Source Code 6.8.

Source Code 6.8: The uvm run test function in top testbench, in Mentor.

SystemC top run test start.

```
1  // Start UVM test environment
2  initial begin
3      $timeformat(-9, 0, " ns", 5);
4      run_test("");
5  end
```

6.4 Simulation and Regression in Siemens Mentor Graphics Visualizer Debug Environment

As a matter of fact, the simulation process in Visualizer is similar to the counterpart in Xcelium except for some minor changes. It is worthwhile to mention that the `real` data type does not perform the same way in Cadence Xcelium Simulator and in Siemens Mentor Graphics QuestaSim Simulator. In Cadence, a value of real type can be negative while in QuestaSim real type is interpreted as unsigned, which incurred a subtle error when the verification environment is transplanted to QuestaSim from Xcelium. In addition, the expected output from the MATLAB reference model is represented in a format like $I \pm Qi$. The + sign could be recognized in Xcelium but it is not able to be identified in QuestaSim so the imaginary part of some output signals is 0 due to this issue.

The optimization commands and simulation commands are added into the script. For elaboration and optimization, the `qrun` command cannot support the elaboration between HLS C++ DUT and UVM environment. Since only one top can be specified in `qrun` command, a `vopt` command with arguments `-top pterodactyl_top sc_main` is used instead, which specifies the top testbench and SystemC top at the same time. An example usage of the script is demonstrated in Table 6.3. By default, the HLS C++ DUT will be used in the verification. For the first parameter, `-c` stands for compilation, `-s` stands for simulation, and `-r` stands for regression. For the second parameter in simulation and regression, the name of the UVM test should be specified in command line. For the third option parameter, if an extra parameter `HLS_RTL` is added in the ending of the command, the HLS RTL DUT will be used instead.

Table 6.3: mentor.sh commands usage.

	HLS C++ DUT Commands	**HLS RTL DUT Commands**
Compilation	mentor.sh -c	mentor.sh -c HLS_RTL
Simulation	mentor.sh -s test	mentor.sh -s test HLS_RTL
Regression	mentor.sh -r test	mentor.sh -r test HLS_RTL

In Figure 6.6, an ideal HLS code coverage flow in Siemens Mentor Graphics environment is described. The tool for collecting higher-level language code coverage is Siemens Mentor Graphics Catapult Coverage, which includes statement, branch, toggle and array access coverage for C++/SystemC HLS designs. The stimulus are sent to C++ DUT. Multiple tests are run for multiple times. The coverage results are recorded and merged into a unified coverage database (UCDB) file. If the C++ code coverage cannot reach 100%, more tests will be added to cover the missing coverage points. Once 100% C++ code coverage is reached, the Catapult Synthesis will generate the RTL code for "Pterodactyl" DUT. To collect the code coverage for "Pterodactyl" RTL DUT, the tests are run in QuestaSim simulator. The UCDB files are merged in the same way. The unreachables should be excluded by Questa CoverCheck, which is an automatic formal solution for achieving code coverage clo-

sure faster. If the code coverage of RTL DUT cannot teach 100% after automatic unreachable exclusion, more stall tests and reset tests need to be developed and run. Once the code coverage of RTL DUT reaches 100%, the flow concludes with final code coverage closure.

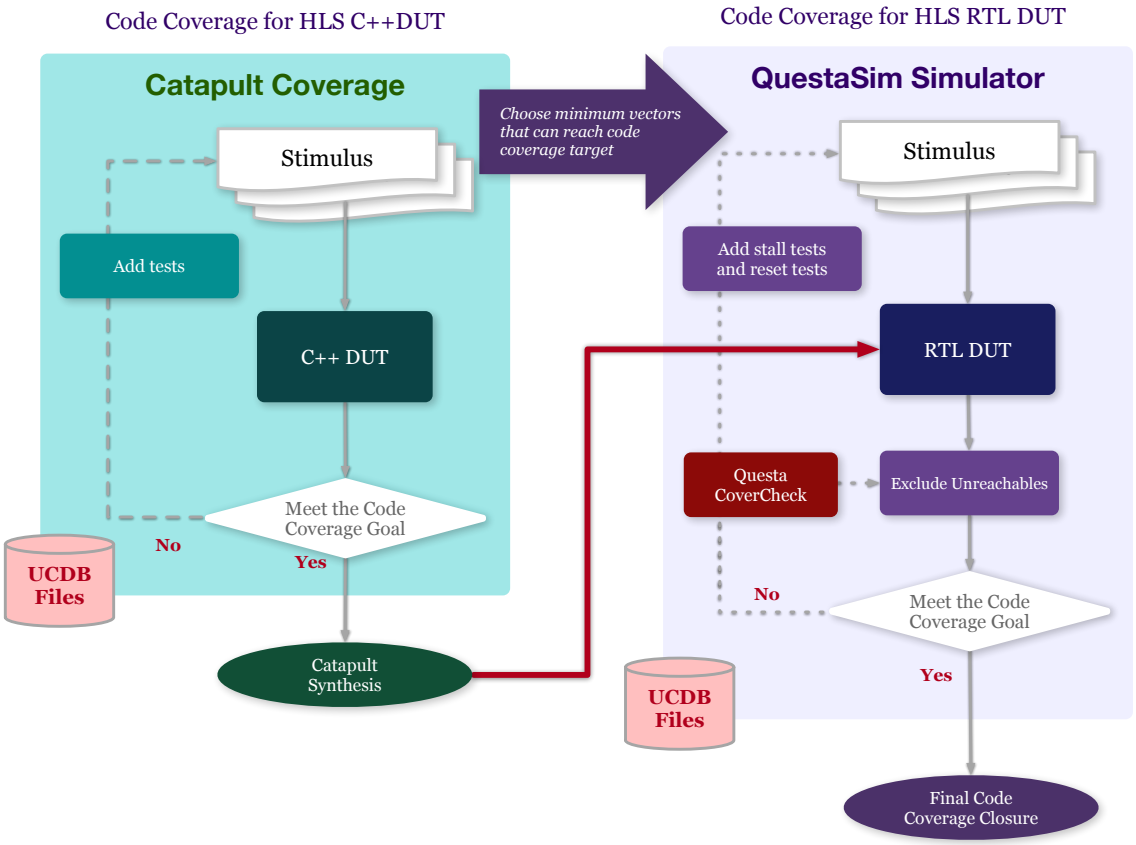

Figure 6.6: The code coverage closure flow in Siemens Mentor Graphics environment.

The C++ code coverage can only be collected Catapult Coverage, because QuestaSim does not support C++ code coverage feature. Before the compilation, the SystemC wrapper files are pre-compiled into a Catapult Coverage shared object. In the simulation, the shared object is specified in command line so that the C++ code coverage can be saved into a UCDB file by Catapult Coverage.

Source Code 6.9: The pre-compilation process of creating a shared object.

The creation of shared object by Catapult Coverage.

```
$CCOV_HOME/bin/ccov -c -fPIC -std=c++11
...
-I/<PROJECT_PATH>/sc_wrapper/
-I/<TOOLS_PATH>/mentor/catapult/10.6/ \
Mgc_home/shared/include/
/<PROJECT_PATH>/sc_wrapper/scWrapper.cpp

$CCOV_HOME/bin/g++ -Bsymbolic -shared \
scWrapper.o -o scWrapper.so -lm \
-L$CCOV_HOME/shared/lib \
-L$CCOV_HOME/lib -lccov_64
```

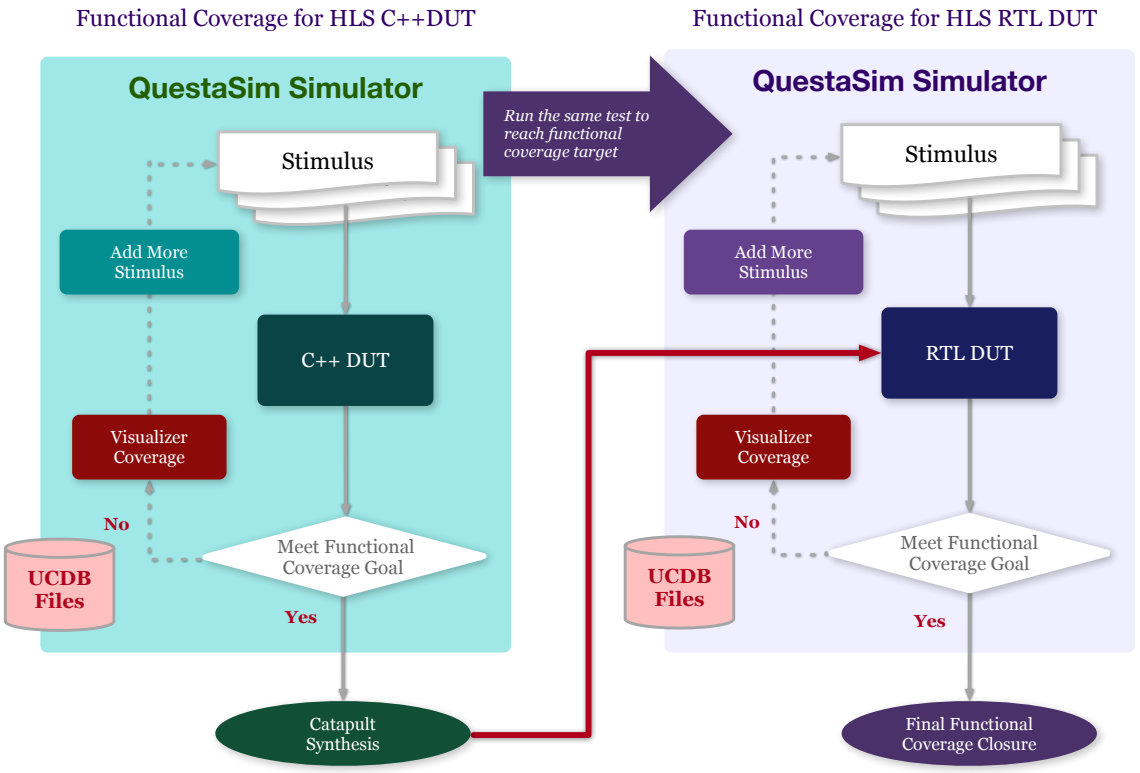

Figure 6.7: The functional coverage closure flow in Siemens Mentor Graphics environment.

In Figure 6.7, similarly, an HLS functional coverage flow in Siemens Mentor Graphics environment is described. The implementation to achieve functional coverage closure is basically the same with that for code coverage closure. The EDA tool for collecting functional coverage in Siemens Mentor Graphics environment is Visualizer. The functional coverage will be collected automatically based on the covergroups which are specified by the user. The coverage data captured from regression tests is also logged in a UCDB file. If the functional coverage cannot reach 100%, the reasons should be figuerd out and directed tests will be created to cover the missing coverage

points. If 100% functional coverage is reached, it means that all the functional points are exercised. Then we can conclude that the functional coverage closure is achieved.

7
Verification results

In this chapter, the functional and code coverage of the Pterodactyl adder block are collected and analysed to demonstrate the feasibility of the entire HLS verification flow. To be precise, we did not test the entire Pterodactyl design consisting multiple blocks. Since our goal is to investigate the feasibility of a high level verification flow, we selected a simple but important block called Pterodactyl adder for our trial to save time for creating more specific tests for other blocks. Therefore, the results mentioned below are collected from our tests targeting only at the Pterodactyl adder.

7.1 Functional coverage

The coverage model is defined using covergroup construct, which is a user-defined type. The idea is to sample interesting variables in the testbench and analyze if they have reached a certain set of values. For example, if there is a 4-bit variable in a covergroup, it can take 16 values. Therefore, if there is something to monitor the variable in a simulation and report what values of the variable have been exercised, we will know if the test covers a particular feature or not. The variable mentioned above is called a coverpoint. A covergroup can contain one or more coverpoints. The *bins* construct allows the creation of a separate bin for each value in the given range of possible values of a coverpoint. *Bins* are said to be hit when the variable reaches the specified values.

There are options in a simulator to dump such coverage details into a file so that it can be reviewed after the simulation is finished. However, to dump coverage details into a file, the tool-specific command should be provided in the command line when simulating. Then a coverage viewer tool like Cadence IMC can be used to open the generated file to examine the coverage details. There is one covergroup that is needed to be checked in our coverage model and four coverpoints are contained in the covergroup. The coverpoints are mainly used to check if the each bit of the 12-bit I or Q signal is both hit by 0 or 1 and if *data_en* and *test_en* have every combination scenario. In our coverage model, the covergroup has been instantiated four times for the real part and imaginary part of TX0 and TX1 signal. After the simulation and coverage collection are done, Cadence IMC/Siemens Mentor Graphics Visualizer which are coverage viewer tools are utilized to investigate the coverage data.

7.1.1 Test scenario

There are multiple tests in the testbench which are used to achieve 100% code coverage because it is not really possible to achieve full code coverage by means of just one test case. We will only focus on one primary test that is used to test the functionality of the adder block. In this test case, test sequences are created and assigned to the corresponding sequencers. Then the configuration bits for the adder mode are randomized and written into the corresponding registers. Also in this test case, the number of transaction items is specified as 1024. The operations are repeated four times for two transmission channels *TX0, TX1* and two *test_stim* signals.

7.1.2 Functional coverage result

The UVM test *test_stim_test* is run and the UVM test report in the transcript of Visualizer is shown in Source Code 7.1. If `TEST_FAILED` is printed out, then the simulation result should be investigated, either checking the waveform of each polyphases or checking the Matlab generated text files can be helpful for debugging.

Source Code 7.1: The UVM test report after `test_stim_test` is run for one simulation.

UVM test report (Part I).

```
# UVM_INFO /<TESTCASE_PATH>/pterodactyl_base_test.svh(293)
↪  @ 11945 ns: uvm_test_top [REPORT PHASE: ]
# ------------------------------------------------
# --------------- TEST PASSED-------------------
# ------------------------------------------------
# --- UVM Report Summary ---
# ** Report counts by severity
# UVM_INFO :   48
# UVM_WARNING :    1
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [PTERODACTYL_SWIF_DRIVER]    12
# [MATLAB_SCBD]     1
# [Questa UVM]     2
# [REPORT PHASE: ]     1
# [RNTST]     1
# [SC_ADAPTER]     1
# [TEST]     4
# [TEST_SEQ]     3
```

UVM test report (Part II).

```
# [TPRGED]     1
# [mrix_cplx_agent:connect]     4
# [main_seq]     1
# [reset_seq]     1
# [uvm_sequence_item]     8
# [uvm_test_top.uvm_env.epd_agent.driver]     2
# [uvm_test_top.uvm_env.epd_agent.sequencer.2b_seq]     1
# [uvm_test_top.uvm_env.gain_strb_agent.driver]     2
# [uvm_test_top.uvm_env.gain_strb_agent.sequencer.2b_seq]
  ↪  1
# [uvm_test_top.uvm_env.rate_mode_agent.driver]     2
# [uvm_test_top.uvm_env.rate_mode_agent.sequencer.1b_seq]
  ↪  1
#    Time: 11945 ns  Iteration: 103  Instance:
  ↪  .pterodactyl_tb_top
```

Table 7.1 shows the overall functional coverage of `data_mrix_cg` in a single simulation run which is 80%. In the Coverage column, the overall coverage is the average value of all the cover points.

Table 7.1: Overall functional coverage for the Pterodactyl adder block, based on single simulation run of `test_stim_test`.

Name	Missing Bins	Total Bins	% Hit	Coverage
data_mrix_cg	3	48	93.75%	80.00%
– data_en_cp	1	2	50.00%	50.00%
– test_en_cp	0	2	100.00%	100.00%
– data_en_test_en_cross	2	4	50.00%	50.00%
– mrix_range_cp	0	16	100.00%	100.00%
– mrix_val_cp	0	24	100.00%	100.00%

Table 7.2: The control signal cover groups and the cross cover group, based on single simulation run of `test_stim_test`.

Name	Coverage	Goal
data_en_cp	50.00%	100.00%
– auto[0]	Count: 0	1
– auto[1]	Count: 16368	1
test_en_cp	100.00%	100.00%
– auto[0]	Count: 8184	1
– auto[1]	Count: 8184	1
data_en_test_en_cross	50.00%	100.00%
– <auto[1],auto[0]>	Count: 8184	1
– <auto[1],auto[1]>	Count: 8184	1
– <auto[0],auto[0]>	Count: 0	1
– <auto[0],auto[1]>	Count: 0	1

Table 7.2 shows the functional coverage result of several coverpoints for the `data_en` and `test_en` control signals. It can be observed that the missing bin in `data_en_cp` is caused by insufficient randomized test stimulus.

There are separated cover groups for each real or imaginary part for the `TX0` and `TX1` channels. The cover points are the same as the aforementioned assignment.

Table 7.3: Functional coverage of `TX0_I`/`TX0_Q`/`TX1_I`/`TX1_Q`, based on single simulation run of `test_stim_test`.

Name	Missing Bins	Total Bins	Hit Ratio	Coverage
Vdata_mrix_cg	5	48	89.58%	65.00%
Vdata_mrix_cg#2	5	48	89.58%	65.00%
Vdata_mrix_cg#3	5	48	89.58%	65.00%
Vdata_mrix_cg#4	5	48	89.58%	65.00%

7.2 Code coverage

Unlike the functional coverage collection, the code coverage is collected by the simulators automatically if the code coverage option is enabled during the optimization and simulation stage. For example, to enable the code coverage collection for the Visualizer Debug Environment, an argument *+cover=sbceft* should be provided at the optimization phase. *sbceft* represents various types of code coverage, which is explained as following:

- b - include branch statistics.
- c - include condition statistics.
- e - include expression statistics.
- s - include statement statistics.

- f - include finite state machine statistics.
- t - include toggle statistics.

At the simulation stage, we also need to provide an additional argument `-coverage` to indicate the simulator to collect code coverage. Table 7.4 demonstrates the code coverage collection for a single test simulation. Reset tests and stall tests should be added in the further regression.

Table 7.4: The result of code coverage, based on single simulation run of `test_stim_test`.

Design Hierarchy	HLS_Pterodactyl	add_tx0	add_tx1
Overall Coverage%	74.81%	79.38%	79.38%
Statement%	96.26%	100.00%	100.00%
Branch%	86.16%	92.18%	92.18%
Toggle%	73.47%	92.51%	92.51%
FSM State%	100.00%	100.00%	100.00%
FSM Transition%	50.00%	50.00%	50.00%
Condition%	61.90%	/	/
Expression%	55.87%	41.60%	41.60%

7.3 Regression test

Because it is not realistic to achieve coverage closure using just one test so we need other additional directed tests to make sure every part of our design is exercised. This is where the regression test comes from. The technique behind the idea is that coverage data from different tests can be merged together into a single database. In other words, if test one covers feature one and test two covers feature two, the merged database will show that both feature one and feature two are covered.

Since the design is not completed, the regression server has not been set up. So the regression test is carried out by the script, the test name and iterations are listed in Table 7.5.

Table 7.5: Plan for regression test.

Test name	Iterations
clk_en_test	10
reg_test	5
reset_test	10
stall_test	1
test_stim_test	15

7.3.1 Code coverage closure

For phase I, the result of the C++ code coverage is stored in a merged UCDB file, which is generated by Catapult Coverage. The only thing that needs to be checked is

the statement coverage of `adder`, and it reaches 100%. For phase II, the final result of the RTL code coverage is merged and shown in Table 7.6. It can be observed that the overall coverage for `add_tx0` and `add_tx1` do reach the goal of 100%. The problem is that, in `add_run_fsm_inst` the toggle coverage is 87.50%. After the source code is investigated, we confirm that the missing part is an unreachable case. The normal practice is to use Questa CoverCheck for automatic exclusions. However, Questa CoverCheck is not available on Ericsson's server, so we have to exclude the uncover part manually.

Table 7.6: Regression results of code coverage.

Design Hierarchy	HLS_Pterodactyl	add_tx0	add_tx1
Overall Coverage%	89.60%	99.53%	99.53%
Statement%	96.67%	100.00%	100.00%
Branch%	92.72%	100.00%	100.00%
Toggle%	76.99%	97.23%	97.23%
FSM State%	100.00%	100.00%	100.00%
FSM Transition%	100.00%	100.00%	100.00%
Condition%	66.66%	/	/
Expression%	94.18%	100.00%	100.00%

7.3.2 Functional coverage closure

Table 7.7 shows the overall functional coverage of `data_mrix_cg` in the regression test has reached 100%.

Table 7.7: Overall functional coverage of the Pterodactyl adder block, based on regression test.

Name	Missing Bins	Total Bins	Hit Ratio	Coverage
data_mrix_cg	0	48	100.00%	100.00%
– data_en_cp	0	2	100.00%	100.00%
– test_en_cp	0	2	100.00%	100.00%
– data_en_test_en_cross	0	4	100.00%	100.00%
– mrix_range_cp	0	16	100.00%	100.00%
– mrix_val_cp	0	24	100.00%	100.00%

Table 7.8: The control signal cover groups and the cross cover group, based on regression test.

Name	Coverage	Goal
data_en_cp	100.00%	100.00%
– auto[0]	Count: 2987160	1
– auto[1]	Count: 3199944	1
test_en_cp	100.00%	100.00%
– auto[0]	Count: 3183576	1
– auto[1]	Count: 3003528	1
data_en_test_en_cross	100.00%	100.00%
– <auto[1],auto[0]>	Count: 1726824	1
– <auto[1],auto[1]>	Count: 1923240	1
– <auto[0],auto[0]>	Count: 1276704	1
– <auto[0],auto[1]>	Count: 1260336	1

As shown in Table 7.9, all the cover groups for each real or imaginary part for `TX0` and `TX1` reach 100% in the regression test. In conclusion, the functional code closure has been reached.

Table 7.9: Functional coverage of `TX0_I`/`TX0_Q`/`TX1_I`/`TX1_Q`, based on regression test.

Name	Missing Bins	Total Bins	Hit Ratio	Coverage
Vdata_mrix_cg	0	48	100.00%	100.00%
Vdata_mrix_cg#2	0	48	100.00%	100.00%
Vdata_mrix_cg#3	0	48	100.00%	100.00%
Vdata_mrix_cg#4	0	48	100.00%	100.00%

Once both code coverage and functional coverage reaches 100% for the HLS RTL DUT, the verification process closure is achieved.

8

Discussion

In this project we investigated an efficient strategy to verify the automatically generated HLS RTL design using the UVM verification standard under Cadence environment and Siemens Mentor Graphics environment respectively. We found that the devised verification flow is universal and can be applied under both scenarios with minor modifications to the UVM-based verification environment and the associated libraries as well as corresponding EDA tools. The functional and code coverage targets can be achieved in both Cadence and Siemens Mentor Graphics tools, which indicates that the DUT works properly and that the UVM-based testbench is able to verify the functionality of the DUT while ensuring that each block of the DUT is exercised.

The critical part of the high level verification flow is the communication and data transaction between the C++ based DUT and the UVM based testbench because the two objects are implemented using two disparate languages. However, two essential libraries, UVM-ML and UVM-Connect, provided by Cadence and Siemens Mentor Graphics respectively are exploited to enable the data communication between the C++ based DUT and the UVM based testbench. The whole point of spending efforts to implement this is to enhance the reusability of our testbench, which means the same testbench can be utilized in both scenarios where the C++ DUT and RTL DUT are being verified. Furthermore, our high level verification flow is quite time-saving because it is strictly structured where the next stage cannot proceed if errors occur at the current stage. Meanwhile, the flow allows the design and verification to be performed at the same time to reduce the time cost.

The exploration of the thesis work provides an alternative verification vendor choice for Ericsson's future plans. Through our investigation, a clear high level verification flow is proposed and assessed as well as the required EDA tools in Cadence and Siemens Mentor Graphics environments. Besides, from Ericsson's perspective, it is important that our verification flow is capable of adapting to various EDA verification environments so that Ericsson could significantly reduce the vendor-dependence.

8.1 Limitations

During the entire investigation, we found out that the most prominent limitation is the insufficient support provided by the EDA tools. For example, we tried not to use the graphical user interface (GUI) of QuestaSim to run the regression test. However, somehow some of the resulting UCDB files containing the coverage data are missing. As a result, we have to use Visualizer tool and keep clicking some icons when a test is finished. Besides, although we can confirm the feasibility of the Siemens Mentor Graphics approach, we are unable to test the efficiency by measuring the total simulation time as listed in our initial goal. The Ericsson's internal task management system, IBM Spectrum load sharing facility (LSF) does not support QuestaSim 2021.1 with `qrun` command which could be used in the local QuestaSim version. The problem is that the regression test that has been run in Cadence environment is assigned to LSF system as a job to be carried out remotely. Therefore, we cannot compare the simulation time elapsed in the Cadence and Siemens Mentor Graphics environments, which is a crucial benchmark that could be utilized to measure the performance. Moreover, we found out that the code coverage for the RTL DUT cannot reach one hundred percent. After careful assessment, it is discovered that some code would not be executed no matter how many tests are performed. And such block of code is called an unreachable item. However, unreachable items should be detected by EDA tools instead of manually. Questa CoverCheck is a powerful tool provided by Siemens Mentor Graphics which is capable of checking for coverage exclusions. By means of the tool, unreachable items are automatically excluded from the coverage model. However, the license for using this tool is missing. Another thing that needs to be mentioned is that the our design does not support downward compatibility when it comes to the versions of QuestaSim, meaning that for example if default QuestaSim 10.6c is used, then the compilation would fail because the `qrun` command and some compilation flags are not supported in previous versions.

8.2 Further improvements

As we mentioned in previous section, the Questa CoverCheck can be a helpful additional tool to do the unreachable coverage exclusion automatically. The following Figure 8.1 shows the flow of using Questa CoverCheck. After running a UVM test, the coverage results can be saved into UCDB files. The UCDB files will be merged and analyzed by Questa CoverCheck automatically. The missing part of the coverage can be categorized into coverables and uncoverables. For example, for branch coverage, there may exist some unreachable if/else and case branches. If the missing part is coverable, then new tests should be written to reach the uncovered part. If the missing part is uncoverable, then the results will be reviewed by both the design team and verification team. After the review, the waivers will be generated and the uncoverable will be excluded.

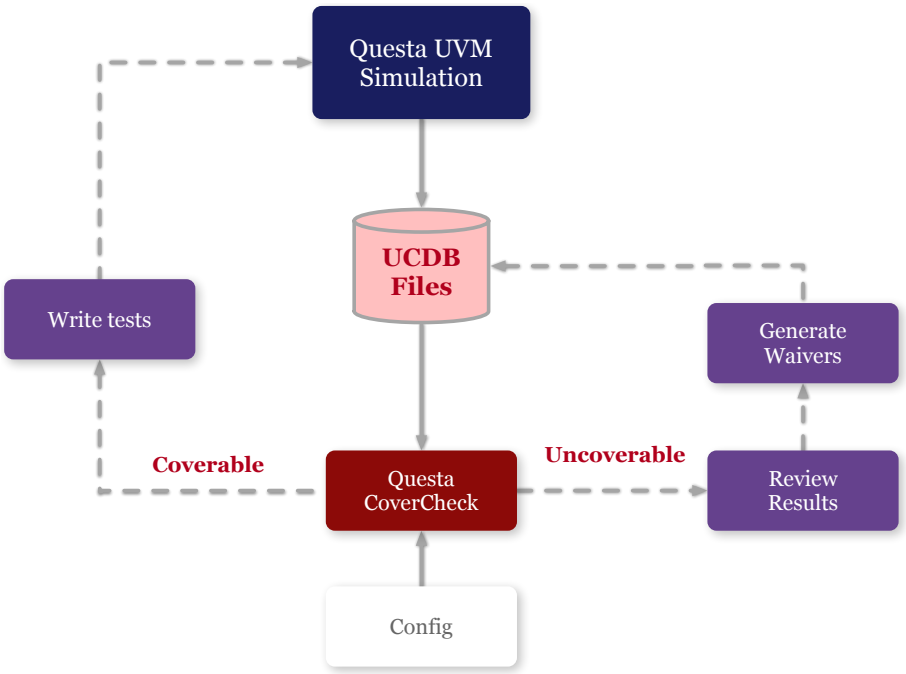

Figure 8.1: The Siemens Mentor Graphics Questa CoverCheck flow.

Based on the discussion with Siemens Mentor Graphics hardware engineers, we found out that there is a better alternative to carry out the coverage driven verification. They can provide a fully automated and latest working demo environment called verification makefile environment (VME) 2022.4, which is an advanced verification environment that uses the Questa Verification tools from Siemens EDA for running simulations and regressions. As shown in Figure 8.2, the VME uses a series of makefiles to simplify the process of compilation, simulation and regressions.

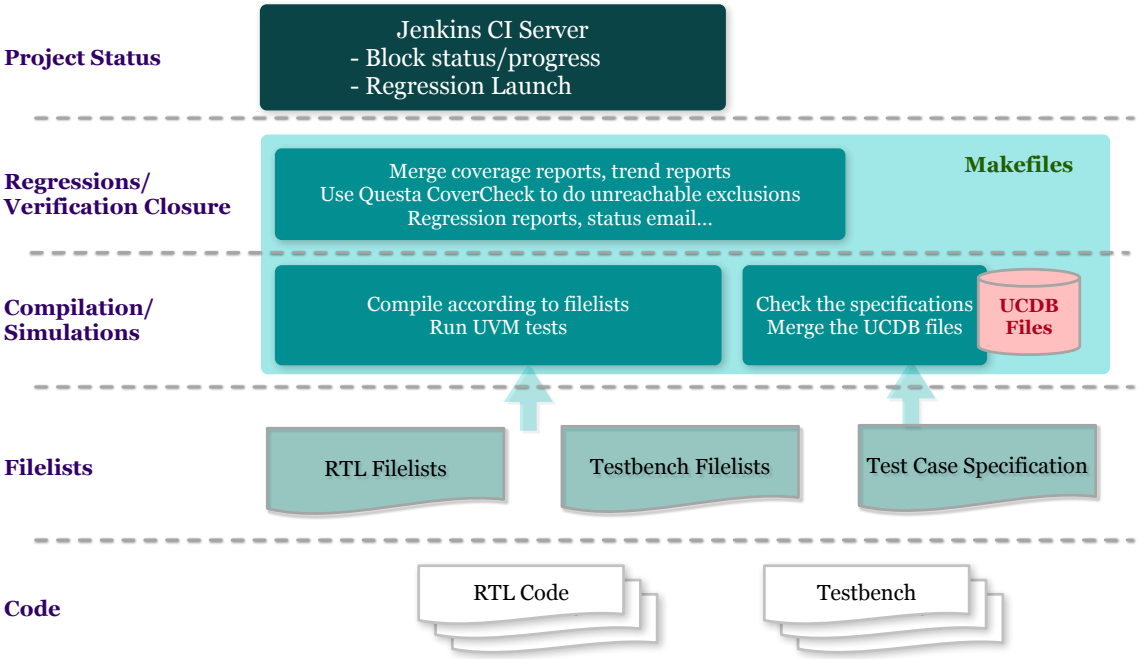

Figure 8.2: The VME verification hierarchy.

It aims to promote the efficiency of the verification flow which takes advantage of latest stuff like the `qrun` command. In the meantime, VME is characterized as flexible, deployable and easy to adopt and improve. It has multiple important features. For instance, the compilation performed in VME is done using `qrun` command that can speed up the compilation and only re-compile the code that is changed. More importantly, it supports lots of advanced techniques such as regression result analysis, test ranking and automatic rerun mechanism if tests fail. Due to the time limit and license issue, we are not able to explore this powerful tool. Nevertheless, it is a quite promising and useful tool that could be adopted in the future to expedite the coverage driven verification flow.

9
Conclusion

In this thesis report we investigated and developed high-level verification flows, as well as explored the reusability and scalability of the UVM testbenches. The HLS verification flow we developed enables verification engineers to verify both HLS C++ and RTL DUTs with the same conditionally-compiled UVM testbench, which saves the time for building a separate C++ testbench particularly for the HLS C++ DUT. Moreover, we elaborated on the working mechanism of the UVM-based verification environment and how our testbench is constructed by means of UVM standard. Besides, we implemented the cross-language framework that enables the communication between the UVM-based verification environment and the C++ based DUT. The framework is able to fit in Cadence or Siemens Mentor Graphics environments with necessary adjustments in code. The automation script we implemented improves the verification process efficiently, which allows compilation, simulation and regression tests to be operated automatically.

However, there are some challenges that demand continuing research and investment in HLS and its verification flow. At present the high-level verification flow suffers from the inadequate support by EDA vendors under the circumstances where the verification environment needs to be migrated from one EDA environment to another. Once the high-level code coverage collection features are integrated to EDA tools, it will lead to a further improvement in the automation process. Nevertheless, the progress on HLS as well as the corresponding verification flow is substantially accelerating, thanks to the increasing support of technology companies, making the goal of designing sophisticated SoC systems in a high-level behavioral description more likely to achieve.

Bibliography

[1] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[2] A. Takach, "Design and verification using high-level synthesis," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 198–203.

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[4] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal verification of high-level synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485494

[5] Y. Herklotz, Z. Du, N. Ramanathan, and J. Wickerson, "An empirical study of the reliability of high-level synthesis tools," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 219–223.

[6] Mentor Graphics, *UVM Cookbook*. Siemens Mentor Graphics, 2022.

[7] ——. (2019, Mar) Closing functional and structural coverage on RTL generated by high-level synthesis. [Online]. Available: https://semiengineering.com/closing-functional-and-structural-coverage-on-rtl-generated-by-high-level-synthesis/

[8] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 88–95, 2009.

[9] E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Elsevier Inc., 2015.

[10] H. Foster. (2021, Jun) Beyond the water cooler: 2020 report on IC/ASIC design and verification trends? [Online]. Available: https://semiengineering.com/beyond-the-water-cooler-2020-report-on-ic-asic-design-and-verification-trends/

[11] B. Murphy, M. Pandey, and S. Safarpour, *Finding Your Way Through Formal Verification*. SemiWiki LLC, 2018.

[12] H. Foster. (2021, Jul) ASIC/IC verification trends with a focus on factors of silicon success. [Online]. Available: https://semiengineering.com/asic-ic-verification-trends-with-a-focus-on-factors-of-silicon-success/

[13] D. Chen. (2020, Aug) The evolution of high-level synthesis. [Online]. Available: https://semiengineering.com/the-evolution-of-high-level-synthesis/

[14] High-level synthesis (HLS): status, trends and future directions. [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-high-level-synthesis-hls-status-trends-and-future-directions#

[15] Mentor Graphics, *Coverage Cookbook*. Siemens Mentor Graphics, 2022.

[16] IEEE, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[17] Y.-N. Yun, J.-B. Kim, N.-D. Kim, and B. Min, "Beyond UVM for practical SoC verification," in *2011 International SoC Design Conference*, 2011, pp. 158–162.

[18] J. Francesconi, J. Agustin Rodriguez, and P. M. Julián, "UVM based testbench architecture for unit verification," in *2014 Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, 2014, pp. 89–94.

[19] About SystemC: The Language for System-Level Modeling, Design and Verification. [Online]. Available: https://www.accellera.org/community/systemc/about-systemc

[20] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*. IEEE, 2003, pp. 19–24.

[21] Accellera, *Universal Verification Methodology (UVM) 1.2 User's Guide*. Accellera Systems Initiative, 2015.

[22] Mentor Graphics, *UVM-Connect and TLM-2.0 Primer*. Siemens Mentor Graphics, 2019.

[23] Cadence, *UVM-ML OA Open Source Reference*. Cadence Design Systems, Inc., 2004.