Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Eivind Strand Erichsen

Interfacing SystemC inside UVM using TLM 2.0

Electronics Systems Design and Innovation

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Trond Ytterdal

June 2020

NTNU
Norwegian University of Science and Technology

Eivind Strand Erichsen

Interfacing SystemC inside UVM using TLM 2.0

Electronics Systems Design and Innovation

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Trond Ytterdal
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

The possibilities and challenges with simulation-based verification with Universal Verification Methodology(UVM), SystemC and TLM2.0 are investigated in this thesis. The methodology presented deviates from most UVM environments with the reuse of a SystemC model reference model. This model generates the expected response from the Design Under Test(DUT) and enables automatic checking. The methodology use constrained random stimulus that excels at finding unexpected bugs and reduce implementation time. The DUT is a complex memory controller previously verified with directed test vectors. It is a confidential Nordic Semiconductor IP recently implemented in their nRF52 System-on-Chip(SoC).

Traditional directed tests specify each test vector to simulate, and the list can grow huge to cover all functionality. These lists are very time-consuming to implement and maintain. The DUT has a 3500 lines long file with test vectors and is updated over several years. Constrained random stimuli can replace the time-consuming lists. Test3 presented in the implementation chapter shows how to generate configuration and memory transactions concurrently in 19 lines of code in a completed environment. In addition to fast stimuli generation, enhances this methodology reuse to decrease implementation time. Protocol components were reused to communicate with the DUT. Only 42% code coverage was reached due to an inadequate reference model. However, the implementation show that the technology works and can be used in verification.

Sammendrag

Mulighetene og utfordringene i simulasjons basert verifikasjon med Universal Verification Methodology(UVM), SystemC og TLM2.0 er analysert i denne masteroppgaven. Denne metodikken avviker fra de fleste UVM verifikasjon miljø ved gjenbruket av en SystemC referanse model. Denne modellen genererer forventet respons fra Designet Under Test(DUT) og muliggjør automatisk sjekking av responsen. Metodikken presentert bruker begrenset randomisert stimuli til å finne uventede feil og redusere implementasjons tiden. DUTen er en minnekontroller som er tidligere verifisert med direkte tester. Det er en konfidensiell enhet fra Nordic Semiconductor som er nylig brukt i deres nRF52 SoC.

Tradisjonelle tester med bestemt stimuli spesifiserer hver eneste test vektor og listen med bestemt stimuli kan vokse til å bli ekstremt stor for å dekke all funksjonalitet. Listene med stimuli er veldig tidkrevende å utvikle og vedlikeholde. DUTen har en fil med test vektorer som er 3500 linjer lang og oppdatert over flere år. Begrenset randomisert stimuli kan erstatte de tidkrevende listene. Test3 presentert i implementasjons kapittelet viser hvordan 19 linjer genererer minne og konfigurasjons transaksjoner samtidig. I tillegg til hurtig stimuli generasjon, muliggjør metodikken gjenbruk av kode som også reduserer implementasjons tiden. Protokoll komponenter er gjenbrukt for å kommunisere med DUTen. Bare 42% kode dekning var oppnådd på grunn av ufullstendig referansemodell. Derimot, viser implementasjonen at teknologien fungerer og kan bli brukt til verifikasjon.

Preface and acknowledgements

This thesis builds on the project "Interfacing SystemC inside UVM using TLM 2.0" written by me in the fall of 2019. It was the proof-of-concept for the methodology presented in this thesis. The background chapter in this thesis builds on the background chapter in the project mentioned.

I want to thank Nordic Semiconductor for providing me with the tools and models needed for this thesis. I also appreciate the weekly meetings that helped to solve problems and encourage me to keep working.

Another asset that was important to finish this thesis was Mentor Graphics. They helped in initial compilation and simulation of the complex SystemC model, a lot of time was saved and ensured I finished in time.

Contents

List of Figures

List of Tables

Listings

Abbreviations

AHB AMBA High-performance Bus.

APB Advanced Peripheral Bus.

CRV Constrained Random Verification.

DUT Design Under Test.

FIFO First In, First Out.

IP Intellectual Property.

OOP Object-Oriented Programming.

RTL Register-Transfer Level.

SoC System-on-Chip.

TLM Transaction-Level Modelling.

UVC UVM Verification Component.

1 | Introduction

The complexity of System-on-Chip grows every year as the costumers are expecting better performance and more features. These chips need to be designed and verified with increasingly demanding time-to-market as competition hardens. A significant market share is lost if the time-to-market goal is lost, and a competitor releases similar products first. According to the 2018 Wilson Research Group Functional Verification Study, the average total project time spent in verification in 2018 was 53 percent[1]. Adopting new verification technology can shorten the verification time, but can also have mandatory expensive tools and training to be efficient.

Verification is an essential part of digital design development, and UVM is a promising verification methodology in simulation-based verification. Mentor's Verification Horizons blog states 71% of all ASIC projects have adopted UVM for testbench development in 2018, and the 2020 projection is 74%[2]. UVM is a new methodology from 2012 and was quickly adopted by companies. UVM is an IEEE standard with guidelines on how to use UVM. This thesis deviates from most UVM testbenches by the use of a SystemC reference model. The UVM scoreboard compares transactions from the DUT and the reference model to check the DUT's functionality. A standard technique seen in the UVM guidelines and typical UVM testbenches, where applicable, is to use a predictor and a memory model as a reference model.

Nordic Semiconductor is in the process of developing virtual platforms in SystemC. The virtual platform is software that can mimic an SoC and enables early software development, architecture exploration, architecture validation and can be used as a reference model in functional verification. A large part of SoCs consists of already verified blocks, and the connection between blocks are essential in verification today. TLM 2.0 with generic payload transaction object enhances interoperability, meaning it is easy to connect other blocks. The memory controller is a part of an SoC, and more blocks can be connected to expand the verification possibilities. Nordic Semiconductor can use their own developed SystemC models and buy SystemC models from vendors. ARM processors is typical in SoC today, and a SystemC model can be bought from them. Developing and buying SystemC models can be expensive, but can be very useful in various phases in digital design development.

Traditional testbenches specify each test vector and expected outcome, resulting in thousands of lines of code to test the functionality of complex DUTs. Constrained Random Verification(CRV) eliminates or drastically reduce the number of manual written test vectors and is a significant advantage for fast to implement and easy to maintain tests. This shift in simulation-based verification is unstoppable as designs grow, and the solutions for the new challenges of simulation-based verification is essential for verification engineers. There are significant advantages using CRV methodology, but the disadvantages in the methodology is the increase of testbench complexity and require resources spent on training

and tools. This thesis has components from different technologies, and the challenges of implementing, simulating and debugging are present. However, these obstacles will decay as the methodology mature in a company. Unlike traditional testbenches with various structures, UVM testbenches can have very similar structure and compilation method, so engineers will quickly become comfortable and efficient with the methodology.

2 | Background

A detailed verification plan can shorten verification time and is important for releasing complex System-on-Chips before time-to-marked. The verification plan should have details on which systems and sub-systems are tested, test plan, coverage plan, response methodology and much more. The response methodology is methods used to check the response from the DUT. The response could be checked with pre-defined expected result, manually checked or automatically checked with a reference model. Verification is often performed hierarchically, sub-components are first verified individually, then a system of these components are verified. Many details should be included for a detailed plan and needs to be made by a person with a good understanding of the systems. If only the input and output interfaces are investigated in the test plan, corner cases may be missed. It is important to have the internal components in mind, so the sub-systems that are not verified have their corner cases tested. Time invested in the verification plan will reduce the time consumed by the later stages of verification. Simulation-based verification is investigated in this thesis, but other methods as formal verification are also possible.

2.1 Simulation-based verification

Simulation-based verification is a common verification technique where test vectors are generated and simulated to verify the DUT. Low-level assertions can verify specific logic and high-level response checking can verify the functionality. In figure 2.1 below, test vectors are manually created together with expected results, simulated and the response are checked with the pre-defined expected results.

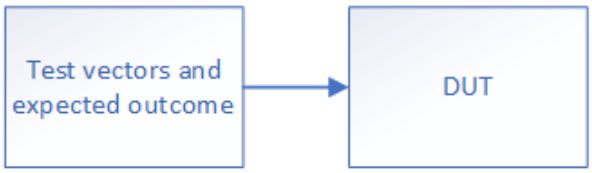


Figure 2.1: Traditional directed testing

When the DUT's response does not match the expected value, the hunt for the bug begins, and it can be challenging to find the bug with no information from the simulation. It is here assertions are very useful as they excite an error in the simulation when it detects something has gone wrong. The designers should make these assertions while they implement the system. The assertions are vital for debugging, and if a test fails while no assertions excites an error, too few assertions are likely implemented. While assertions tell where and when an error has occurred, coverage is signalling that something has been

exercised. When the coverage reaches a certain point, the verification can be documented as complete. Code coverage shows which branches, variables, states and statements has been exercised and how many times. It is a good start to show that all individual statements work, but it does not check cross coverage. Functional coverage can cross-check transitions and succeed in describing complex corner cases where code coverage fails. Code coverage provide an easy metric of exercised code because simulation tools often offer this feature. It is a useful and efficient early verification metric, but functional coverage should be added to complete the coverage of more complex scenarios.

Generating test vectors with expected results can be efficient in verifying small designs, but labour-intensive and error-prone to unexpected bugs with complex designs. This traditional directed testing can be insufficient in verifying today's complex designs and improved tests have automation and reusability in mind. CRV has automatic stimuli generation and shown in figure 2.2 below.

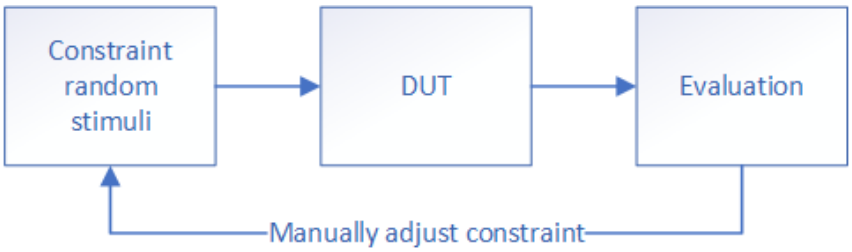


Figure 2.2: Constrained random stimuli

The random stimuli is constrained to target certain coverage and prevent illegal stimuli. The constrained random stimuli are simulated on the DUT who generates a response. A predictor is a component that predicts the expected result. The actual and expected responses are then automatically checked. The predictor can use different methods to get the expected response. The most common methods are using a UVM register model or a high-level model. A UVM register model is simple to implement and are used to model registers. A high-level reference model is a model that can describe complex functionality and has the same functionality as the DUT. The reference model will receive the same data as the DUT and produce the expected response. The evaluation step is where the system is verified or if more testing is needed and if the constraints needs adjustment to target certain coverage. These constraint adjustments are done manually, but a coverage model could automate this process. The coverage plan should be the main driver of the stimuli to achieve completeness.

The simulation can stop after a certain number of stimuli has been simulated or when a certain coverage level has been reached. However, a certain coverage level can require high simulation time with large designs and few constraints. Complex RTL models can use hours or days to simulate with directed testing, and the simulation time could increase with CRV. This increase in simulation time is mainly due to more test vectors may be required in CRV. The directed tests are manually made efficient, while the randomness in CRV can generate less efficient test vectors. CRV would use longer simulation time to achieve the same coverage. This increase in necessary test vectors can contribute to a very high simulation time and results in only small sections of designs can be verified with CRV. Some advanced methods can be implemented to reduce simulation time. High-quality test vectors that excites bugs or hit hard to hit coverage can be reused, a filter can only let through novel test vectors and constraints

can be adjusted. The figure 2.3 below shows an advanced ways to reduce simulation time.

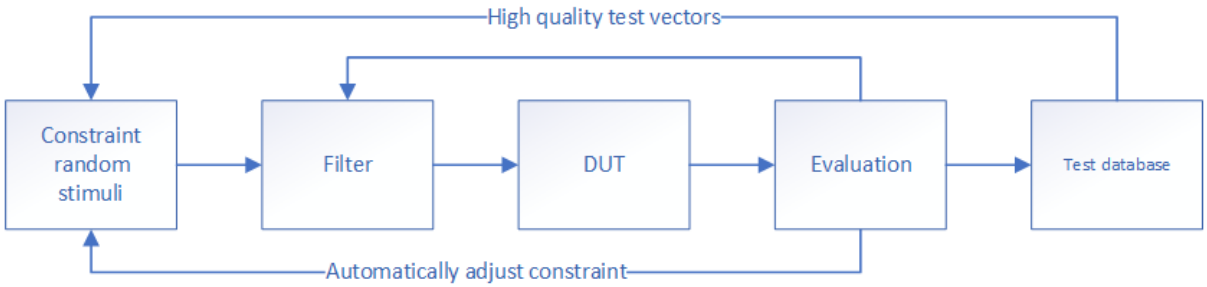


Figure 2.3: Advanced simulation-based test

Simulation time can be huge when the automated stimuli generator is shooting test vectors from the hip, and each test vector is resource-heavy to simulate. It is common to run tests several times and reuse IPs. With this in mind, essential test scenarios can be saved in a test database and reused, as shown above in figure 2.3. Test vectors can be reused from previous simulations or previous projects. A scenario where disposal tests from a database are first run, followed by constrained random tests can harvest the strengths from both traditional directed testing and CRV. Expensive expertise is needed in developing high-quality scenarios for SoC that trigger edge cases. An initiative (Portable Stimulus Working Group) defines a standardise description language of scenarios to maximise reusability across abstraction levels [3]. Lowering verification time is important to meet deadlines, and supplying high-quality tests are becoming important as simulation times increases with large systems. Applying these directed tests should not remove the advantages we are trying to achieve with CRV, but instead add the efficiency of some high-quality directed tests.

The use of filter and test database shown in figure 2.3 is not essential in CRV, but it can solve the problem of high simulation time. CRV can often only be applied to a small section of a design as the simulation time would be too high to test an entire SoC. A paper from 2012 claims a four days simulation was reduced to below six hours by applying novel test detection to a constrained random test generation and simulation environment for a power architecture-compliant processor core[4]. The novel test detector will filter out the tests that are of little or no value, and CRV can be used in more scenarios by ensuring high-quality test vectors.

2.2 Languages and technology

Different languages and technology has different strengths and companies are often combining technology to obtain all the best features. Combining several languages and technology can be beneficial, but can also contribute to complex implementation and debugging.

2.2.1 SystemVerilog

SystemVerilog is called the industry's first Hardware Description and Verification Language (HDVL) because it combines the features of Hardware Description Languages such as Verilog and VHDL with features from specialised Hardware Verification Languages [5]. It is used in RTL development,

implementing SystemVerilog Assertions and in building coverage-driven verification environments using constrained random techniques. SystemVerilog has object-oriented features from c and c++ which makes the language richer that is especially useful in testbenches.

2.2.2 UVM

The UVM standard is developed by the UVM Working Group and improves interoperability and reduces the cost of repurchasing and rewriting IP for each new project or electronic design automation tool[6][7]. Mentor Graphics, a technology leader in electronic design automation, claims UVM represents the latest advancements in verification technology and is designed to enable the creation of robust, reusable, interoperable verification IP and testbench components[8]. UVM is an open-source SystemVerilog library developed independently by the simulator vendors with support from Aldec, Cadence, Mentor Graphics, and Synopsys. This thesis uses Mentor Graphics' Questasim verification tool to compile and simulate UVM, SystemVerilog, SystemC and TLM 2.0 together.

UVM communicates on transaction-level and is used for testbench development. Two important features is the object-oriented programming(OOP), and polymorphism that enables reuse and configuration of classes. A standard library of UVM classes is recommended to be used in the testbench. This means the designer does not need to be an expert in OOP since the recommended classes is already developed, and just extending these classes can be the best practise. Pure SystemVerilog testbenches are written specifically for a single purpose use and often with little guidelines. These testbenches have low or none reusability, high maintenance cost and hard to understand. All UVM testbenches should use the same base classes and are easily understandable for engineers worked with UVM before.

UVM was designed with reuse in mind and encourage the use of the standard classes that can encapsulate behaviour. While UVM enables reuse, it is not guaranteed that something can be reused between environments, hierarchies or projects. It is the verification engineers responsibility to design components that can be reused. The most common reuse is verification components between environments, but entire verification environments can also be used between different projects or in the same tests. A common practice is to instantiate `umv_environment` multiple times in the same test, but with a different configuration.

A reusable UVM Verification Component(UVC) for a particular interface can be used in many testbenches. A protocol UVC can, for example, generate an AHB transaction item and drive it on the DUT's interface. A specific UVC is created for each of the DUT's interfaces and usually with a `uvm_sequencer`, `uvm_driver` and `uvm_monitor` components inside as figure 2.4 shows. The dashed line is marking what is included in the UVC. The role of a UVM agent is to encapsulate the sequencer, driver and monitor into a single container to enable reuse. Other components as protocol checker, coverage monitoring or other components can also be added to the agent.

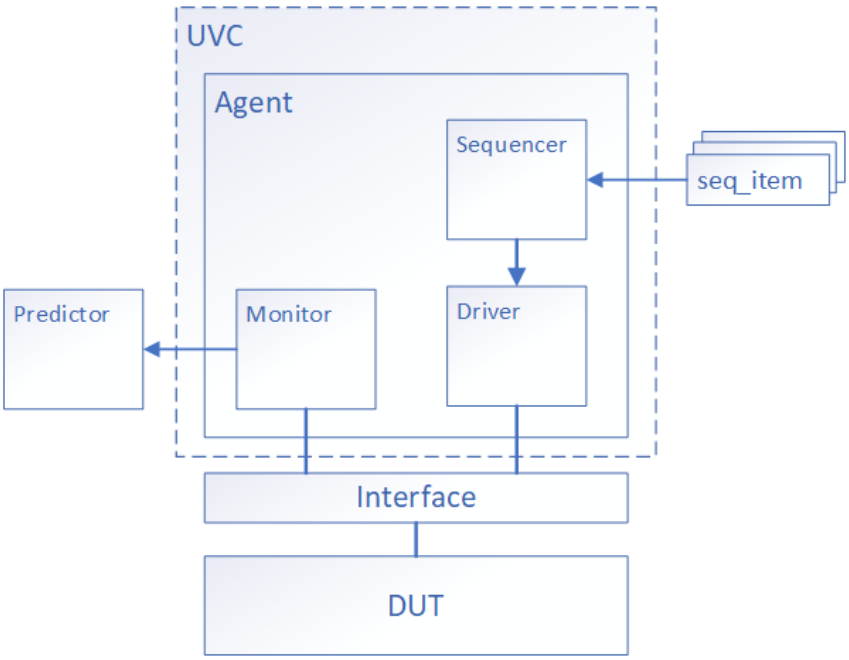


Figure 2.4: Block scheme of a typical UVC and connections

A sequence consists of sequence items that are passed by the test to the sequencer who communicate with the driver. The driver drives a particular the interface like AHB or APB. The DUT and the `uvm_monitor` is connected to the interface and receives these transactions. The monitor should not set any signal on the interface, but only listen and broadcast completed transactions.

The high abstraction enables UVM components to be compact with a lot of functionality, but a consequence can be a steep learning curve. A single line of code connects a sequencer with a driver and establish a handshaking protocol. This is an example of UVM that abstracts away details and force engineers to implement best practices. This abstraction can contribute to confusion in what is happening during the simulation, but enables engineers to implement tests quick and reliable when they are comfortable with UVM.

UVM Connect is an open-source UVM-based library from Mentor Graphics that enable TLM 1.0 and TLM 2.0 communication between UVM and SystemC. It is used by SystemC designers who want to leverage SystemVerilog and UVM functionality and opposite where verification engineers wants to leverage SystemC[9].

2.2.3 UVM code generation

UVM has its upsides, but one of the downside are large base structure needed to start simulating. While there can be a lot of code, a similar structure is used for every testbench with repeating patterns. UVM code generators are made to enhance productivity and remove the tedious process of writing the base structure. Script generated files remove pitfalls done by new and experienced designers and give a "flying start" to the verification. Using a code generator can further enable cooperation as it is much easier to understand a structure a designer has previously work with. Many code generators exist, and Nordic Semiconductor may make their own to suit their needs. Two well-known examples of code generators is Doulous Easier UVM and Mentor UVM Framework.

Doulos Easier UVM[10] is a UVM code generator written in Perl that converts files describing the components and the DUT into basic UVM structure with components and connections. Usually, only the necessary functions are implemented, and it is up to the designer to implement further functionality. www.edaplayground.com is an online generator with many examples that simplifying the learning process.

Mentor's UVM Framework (UVMF) is a code generator written in Python that promise best-practices UVM on the code generated. The designer write a YAML file describing the testbench and the generator create UVM infrastructure that is ready to run and guarantees good model reuse possibilities.

2.2.4 Transaction-level modelling

Transaction-level modelling(TLM) is a communication abstraction where transactions are sent as a communication object. UVM uses TLM communication between components and a driver to convert the transactions to signal toggling for the RTL DUT interfaces. The UVM monitor translate signal toggling on the DUT's interfaces back to transactions. SystemC is also communicating on a TLM abstraction, but not compatible with UVM without any bridge between them.

TLM 2.0 is an Open SystemC Initiative transaction-level modelling standard and can together with UVM Connect connect UVM with SystemC. TLM 2.0 is focused towards the modelling of on-chip memory-mapped busses and ensures high speed by sending the reference of the object. The generic payload is the standard transaction object, but an extension can be attached if some attributes are lacking or it can be replaced by another transaction object. The SystemC reference model used in this thesis attach an extension on the generic payload to distinguish transactions sent from different sockets. A bus model can be implemented with an initiator socket at the master and target socket at the slave, and the generic payload as a transaction object enable maximum interoperability.

High abstraction TLM models can simulate many order of magnitudes times faster than the RTL models and can be an important factor for reach deadlines. When RTL models are simulated, the simulator is examining every event or clock cycle and results in slow simulation speed. TLM use function calls for inter-module communication which ensures high simulation speed.

2.2.5 SystemC

SystemC[11] is an extension of C++ and provides an event-driven simulation interface that was released in 2000 and had waves with popularity and new releases. It is an IEEE standard that only requires a C++ compiler to simulate, and this is usually an advantage over proprietary technology. SystemC deliberately mimics the hardware description languages, but at a system-level which makes it an excellent reference model that is faster to implement and simulate than RTL. SystemC is applied in system-level modelling, abstract analogue/mixed-signal modelling, architectural exploration, architectural validation, performance modelling, software development, functional verification and high-level synthesis[12].

SystemC is used in software-based systems that can fully mirror the functionality of a complex SoC, and this is called virtual platforming. System architect engineers design the virtual platform and can be used by several groups in several design phases. Software engineers can use it for application software and software development, system architect engineers can use it in architecture exploration, and hardware

engineers can use it as a golden reference model in testbenches. This exchange in executable code between engineers can also replace or be added to traditional paper specification between teams. Cadence claims their Virtual System Platform can shave months off system development schedule by facilitating early software development, higher software developer productivity, and continuous hardware/software integration validation[13]. The fact that Cadence, a respected semiconductor company, build virtual prototypes with SystemC and TLM 2.0 technology, backs up the importance of this technology today[14].

SystemC TLM models can be untimed, loosely timed or approximately timed, meaning different level of correct performance representation of an implementation. The real transaction times can be estimated and specified in the TLM model resulting in an approximately timed model, and the performance of the system can be analysed. Many SoCs are targeted to embedded system applications which have real-time constraints. It is often necessary to verify real-time performance goals by simulating complex scenarios with hardware and software components. Verifying such scenarios can be as important as verifying the logic design. High-speed transaction-level models with the appropriate level of timing accuracy can be an excellent way of verifying that performance requirements are met in certain scenarios[15]. Timing accurate SystemC models enable simulation of performance and architecture exploration, but this thesis uses only the functionality of the model. Therefore, the timing is not essential in this thesis. A system-level model can be used as a golden reference model regardless of the timing, and only the functionality has to be correct. By sending the test vectors to the DUT and the system-level model, functional equivalence check can be performed to verify the functionality of the DUT.

It is possible to make a reference model in RTL, but it is a complex task and will increase time-to-market versus implementing the reference model with a high abstraction level. Some language options for designing a high-level reference model is Matlab, Python, C or SystemC, all with different pros and cons. Python makes it very easy to implement the model, but has a relatively slow simulation speed and is difficult to interface to SystemVerilog. Matlab makes it easy to implement the model, has a fast simulation speed and has better interface opportunities than Python, but is not open source and makes it difficult with reusability. C is fast and have a good interface, but lacks the mimic opportunities that SystemC has. SystemC is fast, moderate interface difficulty, moderate implementation difficulty and suitable hardware mimicking. It requires time and effort to implement a reference model in SystemC, but it has fast simulation speed and mimics the real implementation well [16].

A company can implement TLM models for new IPs to gradually building up a library of TLM models or implement TLM methodology for all major existing IP blocks. Architectural exploration can be done early and efficiently when a pool of relevant TLM models is available.

2.3 Coverage-Driven CRV

The two cornerstones of CRV is the automatic check of the DUT's response and coverage which is the metric-based completeness. Both features can be challenging to implement and use. The automatic check of the response can be implemented in different ways. A UVM register model is a common way to predict values in registers and can be used to produce the expected response from the DUT. The register model can not predict all types of responses from a DUT that a reference model can. The register model

is commonly used in simpler DUT's with registers, and a reference model is implemented to mimic a more complex DUT. The reference model can be implemented in different languages, as discussed in chapter 2.2.5. A SystemC reference model is used in this thesis to enable automatic checking. A challenge arises when the reference model and DUT does not have the same functionality. This scenario is present in this thesis because the reference model has only the basic functionality of the DUT. The result is a restriction in what features that can be tested and limitation on the stimulus generation. This reference model can not be used in complete verification before more functionality is added. This thesis focuses on the possibilities with this methodology and not the verification of the DUT.

Another challenge is to decide when we are done simulating. A coverage plan monitors exercised functionality and determine the completeness. Ensuring the completeness with the tests can be difficult with a complex DUT. Guiding tests with constraints can be mandatory to hit specific functionality, and achieving boundary scenarios can be non-trivial in complex systems.

3 | Implementation

The DUT is a verified complex memory controller with an existing testbench implemented with directed test vectors. It is a confidential Nordic Semiconductor IP used in their nRF52 SoC and other products. The main purpose of the IP is to translate AHB bus-accesses into flash accesses. Internal registers in the memory controller enables write, read or erase of the flash memory. The internal registers are accessed with the APB bus-accesses. The flash memory simulated in this thesis is 1MB large and is a detailed RTL model.

UVM uses transaction-level communication between all UVM components, but the UVM scoreboard predictors convert the transactions to be able to communicate with the SystemC reference model. The transactions are converted to TLM2.0 generic payloads and sent over TLM2.0 sockets to the reference model. Figure 3.1 shows an overview of the test structure.

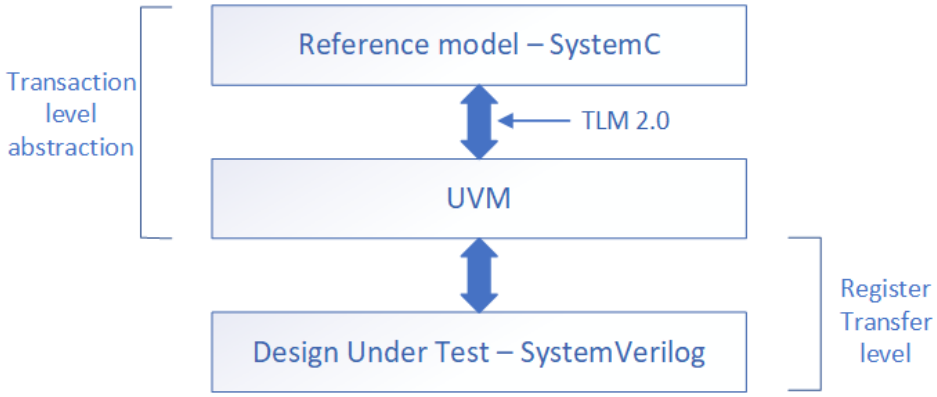


Figure 3.1: Test overview

3.1 SystemC Reference model

The SystemC reference model is intended to have similar functionality as the DUT, but at a high abstraction level. It uses function calls as communication method and simulate must faster than RTL. The model uses TLM 2.0 blocking and non-blocking interface that enables communication to UVM or other SystemC blocks. The model was designed with the purpose to test the possibilities with a SystemC reference model in UVM. It has only the basic functionality of the DUT. The functionality is listed in table 3.1.

Table 3.1: List of the reference model's features

Configuration registers (APB)	Flash memory (AHB)
Write enable	Write to memory
Read enable	Read from memory
Erase enable	
Erase all	

A SystemC model can be approximately timed when the functionality is fully developed with non-blocking communication and approximate timing inserted. The non-blocking can enhance the timing of the model to approximately timed because it does not block the simulation and has more phases to model a transaction accurately. The non-blocking has more phases that make it more resource-heavy and thus slower simulation speed than blocking, but much faster than RTL simulation. Approximate timing is important for performance modelling and architectural exploration. The reference model has non-blocking communication implemented, but not the needed details to approximately model the performance of the memory controller. Both blocking and non-blocking interface is implemented in UVM, but blocking interface is preferred because it has faster simulation speeds and is easier to implement. SystemC reference model is shown below in figure 3.2.

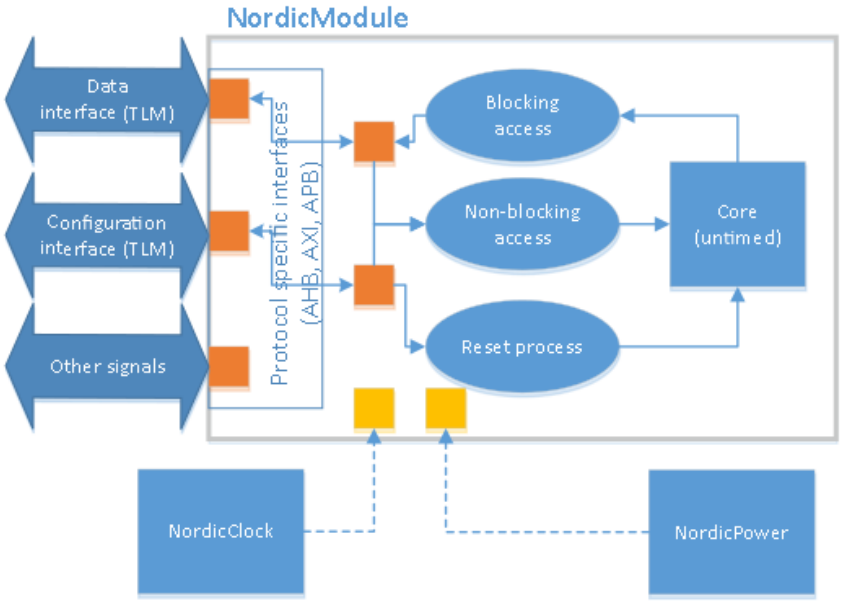


Figure 3.2: SystemC reference model block scheme. Figure made by Nordic Semiconductor.

The SystemC model was developed with SystemC and compiler versions not available in the Mentor Graphics' Questasim verification tool and never tested for compatibility. It was a challenge to compile, and different tool versions had different problems. Some functionality as accessing the generic payload extension exits the simulation without any warning or error from the simulator. Specific obstacles show up in new methodologies, but can be evaded or fixed as the technology matures in the company.

3.2 UVM

3.2.1 UVM Verification Component

A Verification IP is commonly known as a component used in testbenches. Similarly, a UVM Verification Component(UVC) is a component used in UVM testbenches. This thesis uses two protocol UVCs that have the main purpose of driving transactions to the DUT's interfaces. The main components inside the UVCs are `uvm_sequencer`, `uvm_monitor` and `uvm_driver` and shown below in figure 3.3.

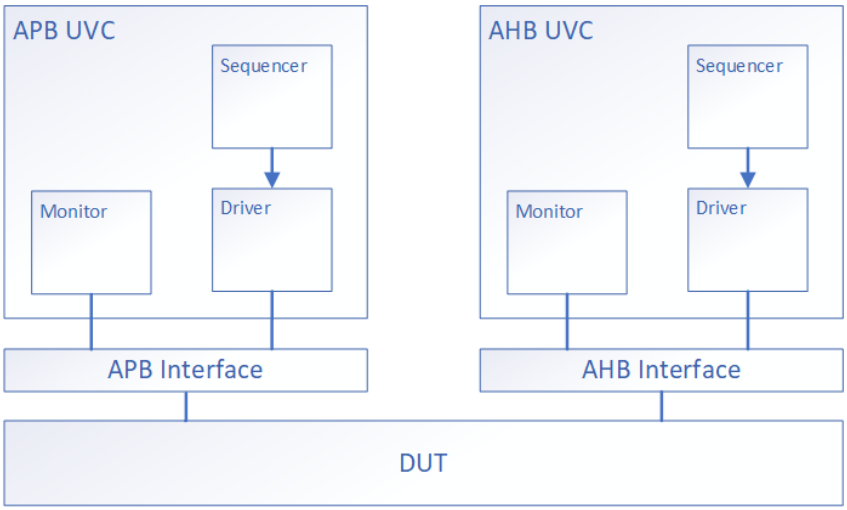


Figure 3.3: Block scheme of the UVCs

The sequencer controls the flow of request and response sequence items between sequences and the driver. The driver has knowledge how to drive signals on a particular interface. The AHB UVC has a driver for the AHB interface, and the APB UVC has a driver for the APB interface. A monitor is a component that observes DUT pin level activity and converts its observations into transactions. It also sends these transactions to analysis components through an analysis port. The UVCs used in this thesis are made by Nordic Semiconductors and have more features as the choice between master and slave, protocol checker, coverage monitor and library of sequences. It is a component that can be used as it is, but it is common to write sequences specially made for the DUT. The sequence library has basic sequences as random base transaction, burst transfer and pipeline transfers. Custom sequences are easily made when they are extended from one of these sequences in the library as these have a lot of the functionality needed.

3.2.2 Scoreboard predictor

The scoreboard predictors are the main components implemented that enables the unique flow presented in this thesis, the use of a SystemC reference model. The UVCs broadcast their completed transactions through a `uvm_analysis_port`. A predictor is implemented for each UVC to receive these transactions through a `uvm_analysis_imp`. The APB predictor receives the completed APB transactions, and the AHB predictor receives the completed AHB transactions. These transactions are transformed into TLM 2.0 `uvm_tlm_generic_payload` to be compatible with the reference model. Using the generic payload as a transaction object improves interoperability between blocks because many blocks are compatible.

When a transaction reaches a predictor, the transaction has already been executed by the DUT meaning the read transactions have the DUT's response. The read transactions are converted to two separate generic payloads objects. The first object has the address and the data; this object represents the actual response from the DUT and is sent to the scoreboard. The second, has only the address and is sent to the reference model to retrieve the reference data. This object represents the predicted expected response and is also sent to the scoreboard. A write transaction is only transformed to one generic payload since these are only sent to the reference model, not to the scoreboard. Two methods of transaction-level communication with the reference model are implemented. A simple and fast blocking socket is implemented with `uvm_tlm_b_initiator_socket` and a more complex non-blocking socket is implemented with `uvm_tlm_nb_initiator_socket`. Only one socket is needed to communicate with the SystemC reference model, but they demonstrate the possibilities. A compilation flag determine which socket type is compiled and simulated. Figure 3.4 shows a block scheme of the predictors.

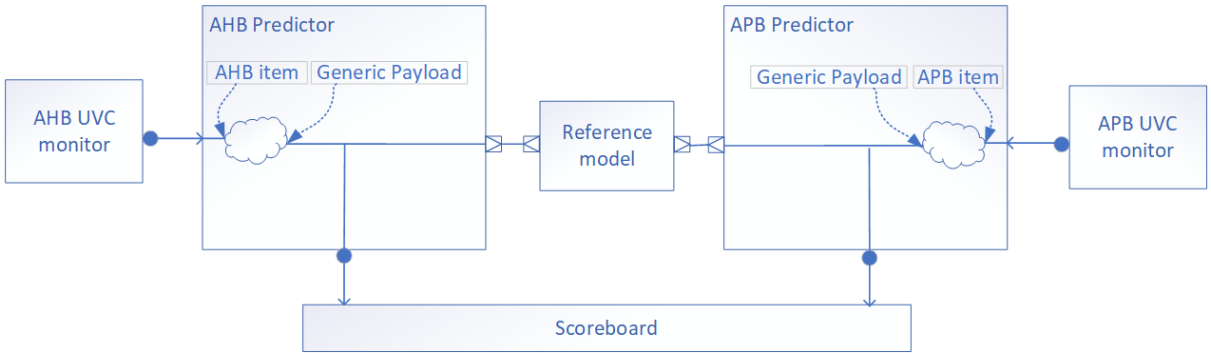


Figure 3.4: Block scheme of the predictors and their connections

The predictors were implemented as separate components to keep the behaviour simple and encapsulated to enable reuse. An implementation where a predictor received transactions from both UVCs can be efficient for this testbench, but can be too complex for easy reuse. The predictors can easily be reused in a test that uses the respective UVC and a reference model with TLM 2.0 sockets with generic payload.

The non-blocking interface has a unique opportunity to model a transaction accurately and thus, model performance. The DUT could have a performance demand that arguable can be just as important as logic bugs. The SystemC model has an opportunity to model these performance demands and can be used to validate the performance of the DUT. The reference model can provide the information about the expected response, but could, in theory, give information about expected performance. The scoreboard could compare the performance and is a reason for the demonstration of non-blocking communication and why it is included in the predictor despite blocking is almost always best in functional verification. The non-blocking have advanced features as the possibility to send a new transaction before the previous is completed. There is a conflict between having as many features as possible versus a simple and easy to understand component. Every code line written increases complexity, the possibility of bugs and extra code that decrease simulation speed. The AHB predictor with both blocking and non-blocking is presented in appendix A.1. This version of the predictor has all the functionality, but is complex. A version of the AHB predictor with only blocking can be found in appendix A.2. This version has the functionality needed to use the reference model in functional verification and is simple and easy to

understand. The predictors was written with reusability in mind, and a simple and easy to comprehend predictor would decrease implementation time, debugging time and reduce maintenance cost. The APB predictor is implemented similarly as the AHB predictor.

3.2.3 TLM 2.0 sockets

The generic payload transaction object represents a generic bus read/write access. It is used as the default transaction in TLM 2.0 blocking and non-blocking transport interfaces. It has common bus attributes listed in the table 3.2 below.

Table 3.2: List of generic payload attributes

Attribute	Explanation
`bit [63:0] m_address`	Address for bus operation.
`uvm_tlm_command_e m_command`	Bus operation type.
`byte unsigned m_data[]`	Data read or to be written.
`int m_length`	Number of data bytes.
`int m_dmi`	Not yet supported.
`int m_byte_enable`	Indicates valid m_data array elements.
`int m_byte_enable_length`	m_byte_enable length.
`int m_streaming_width`	Number of bytes transferred on each beat.
`uvm_tlm_response_status_e m_response_status`	Status of the bus operation.

The data attribute in the generic payload is a byte array, and the data in a testbench is rarely this format for practical reasons. A transformation is needed to match the testbench data to the `m_data[]` generic payload attribute. The data in this testbench is a 32-bit int and the first transformation performed is dividing the int into 4 bytes with the SystemVerilog streaming operator "»". The bytes is then switched to reverse order since SystemC reads the bytes from left to right. The transformation is done in two steps and shown in the figure 3.5 below. The data returned from SystemC has its byte order switch back to be comparable.

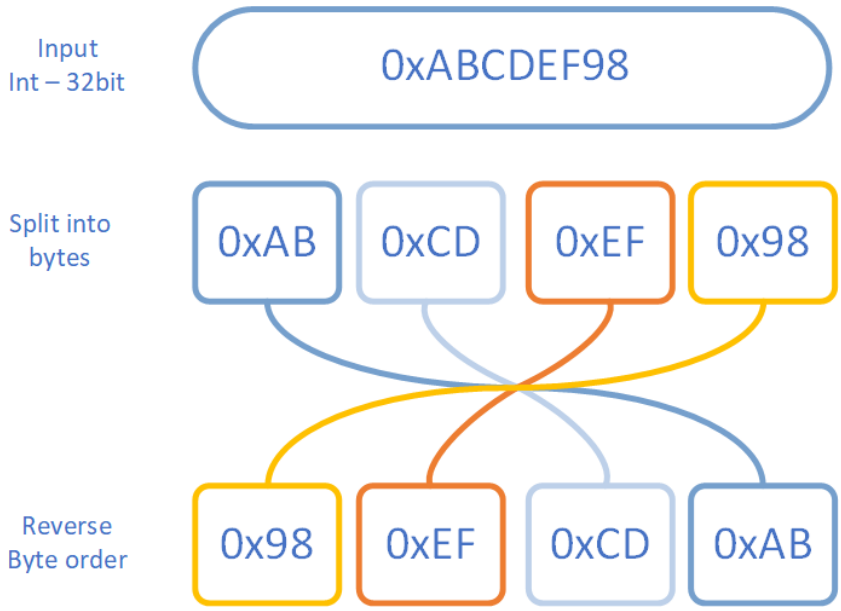


Figure 3.5: Packing generic payload data

TLM 2.0 offers many methods of communication, and the reference model has blocking and non-blocking sockets implemented. One socket type is sufficient for communication, but these sockets have pros and cons and used in different applications. The blocking socket is simple and fast while the non-blocking can be complex, timing approximate and more resource-heavy. The blocking socket completes the transaction in one function call and is therefore not resource-heavy on the simulator. In contrast, the non-blocking can use several phases and be more resource-heavy. The fast and simple blocking implementation is preferred in this application, but both blocking and non-blocking are implemented in the predictor to demonstrate the possibilities. The chart of a typical successful non-blocking transaction is shown in figure 3.6.

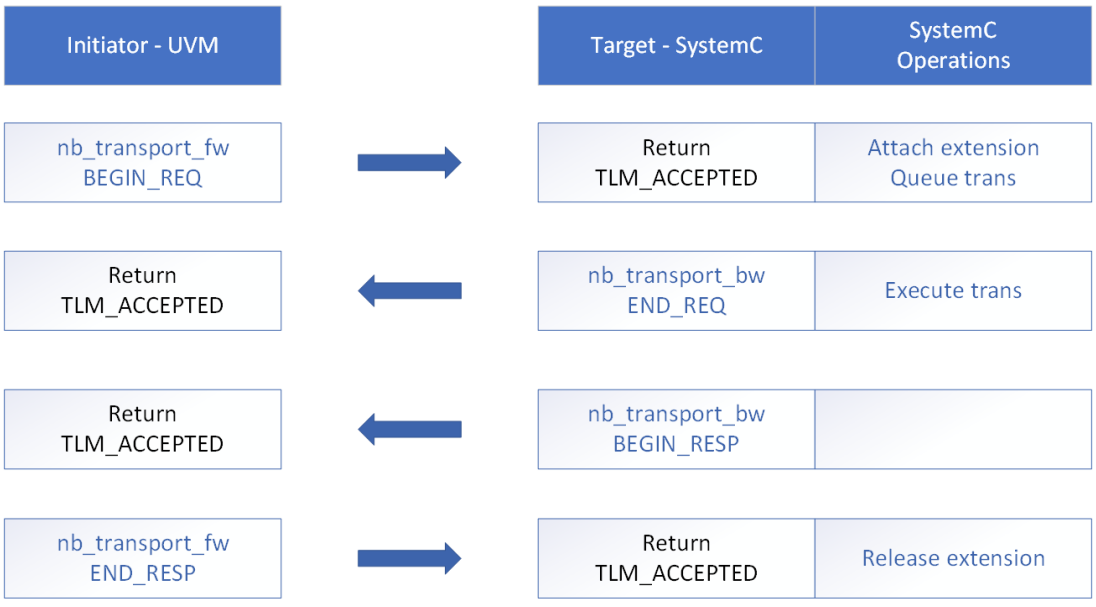


Figure 3.6: Non-blocking transaction chart

The blue arrow represents a function call and can either be in the forward direction or the backwards direction. In this protocol, TLM_ACCEPTED is returned to proceed the transaction and TLM_COMPLETED is returned to terminate the transaction early. If TLM_ACCEPTED is always returned, the transaction stops after the fourth phase, which is the end response. This means TLM_COMPLETED is not needed to complete a transaction, but can terminate a completed transaction. The reference model has several sockets for different access types and uses an extension on the generic payload to specify which socket the transaction was sent through. An APB and AHB have both address and data, but it is important to differentiate that one sends data to registers and one to memory.

More detailed information about the transaction can be found in the generic payload transaction object. The generic payload m_response_status attribute shown in the attributes table above 3.2 has several states that can be used to determine what to do next. The possible states are shown in table 3.3.

Table 3.3: List of possible response status

UVM TLM generic payload response status	
OK_RESPONSE	Bus operation completed successfully
INCOMPLETE_RESPONSE	Transaction was not delivered to target
GENERIC_ERROR_RESPONSE	Bus operation had an error
ADDRESS_ERROR_RESPONSE	Invalid address specified
COMMAND_ERROR_RESPONSE	Invalid command specified
BURST_ERROR_RESPONSE	Invalid burst specified
BYTE_ENABLE_ERROR_RESPONSE	Invalid byte enabling specified

The default state is INCOMPLETE_RESPONSE, and the target should only change the state. If the transaction has status OK_RESPONSE, the target has completed the transaction and the transaction can in some cases end early with TLM_COMPLETED. An error will in many scenarios stop the simulation, but for example, the
ADDRESS_ERROR_RESPONSE can be solved by terminating the transaction and send a transaction with different address instead. Implementing different solutions to different errors can make a robust simulation.

3.2.4 UVM Connect

UVM Connect connects the SystemC model with UVM in a relatively straightforward process. The sockets are compiled separately, but with a look up string to later connect them during the link phase. The AHB transactions are sent through the socket called data_socket, and APB transactions are sent through the socket called config_socket. SystemC implement target sockets using the tlm.h library with the following code:

```
tlm_utils::simple_target_socket<memctrl, DATA_BUSWIDTH> data_socket;
```

This target socket in SystemC can be used as blocking or non-blocking by registering local call-back functions for data interface:

```
data_socket.register_b_transport(this, &memctrl::data_b_transport);
data_socket.register_nb_transport_fw(this, &memctrl::data_nb_transport_fw);
```

The SystemC data socket is registered in UVM Connect using the uvmc.h library. The last argument is a lookup string used for connecting the socket to the UVM socket.

```
uvmc_connect(mem.data_socket,"data_trans");
```

The UVM predictors implement blocking and non-blocking initiator sockets with:

```
uvm_tlm_b_initiator_socket #() data_socket;
uvm_tlm_nb_initiator_socket #(ahb_predictor) nb_data_socket;
```

The sockets in the UVM predictors are registered in UVM Connect with identical lookup strings to the corresponding SystemC socket. Only one of the sockets should be instantiated, either blocking or non-blocking.

```
uvmc_tlm #()::connect(m_ahb_pred.nb_data_socket, "data_trans");
uvmc_tlm #()::connect(m_ahb_pred.data_socket, "data_trans");
```

Above is the description of connecting the data socket for AHB and similar is implemented for the APB config socket.

3.2.5 Scoreboard

Each predictor broadcast two generic payloads through two `uvm_analysis_port` and the scoreboard receive these through four `uvm_analysis_imp`. Four queues implemented as FIFOs are used to store the incoming transactions. The APB transactions are compared when the queues with the actual response from the DUT and expected response from the reference model are not empty, and similar with AHB.

The scoreboard ensures correct functionality on a high-level and have no information about low-level specific protocol execution or RTL behavior. Assertions check the RTL behaviour and a protocol checker checks the protocol. The UVCs includes protocol checkers that monitors the interface and signals an error if an error occurs, similar to assertions. The generic payload is only a summary of the real transaction, but it is enough to say if the functionality is correct.

The desired number of AHB and APB transactions generated and compared is stored in the UVM configuration database and retrieved by the scoreboard. If the number of transactions is above zero, the scoreboard raises an objection that prevents the simulation from ending before all transactions intended is compared. The scoreboard is almost always tailored towards the test and the DUT, and thus not entirely reusable. The implementation of the scoreboard is presented in appendix A.3.

3.2.6 Code generator

The UVM structure is made with Easier UVM code generator by Doulus and includes generation of files, components, the connection between components and an initial script to run the simulation. Easier UVM gave the project a flying start, and a code generator should always be considered when starting a UVM project. The result of a company encourage engineers to use this technology are many testbenches with similar structure. This will enable fast cooperation as every engineer is comfortable with the structure. The files and generated structure is shown below in figure 3.7.

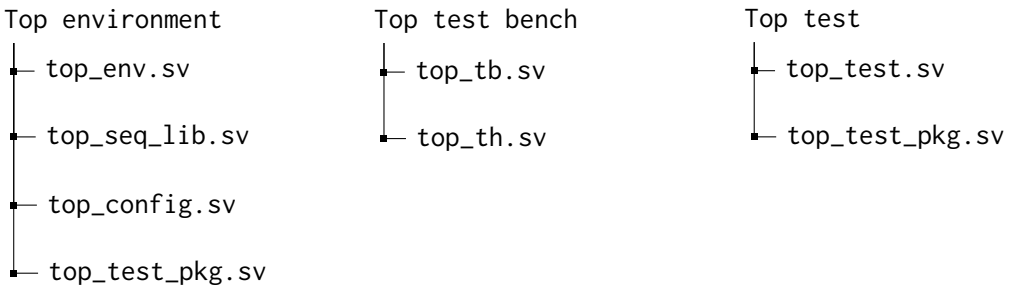


Figure 3.7: Initial structure

The initial structure has built a solid foundation were more components can be added. The "Top environment" is where the UVM components belong and the "Top testbench" is where the RTL models belong. The "Top test" wraps "Top environment" and can instantiate different configurations of the environment and start different tests. The package files is used to decide what files are compiled and marked with "pkg.sv" in the file name. "top_env.sv" is the only UVM environment in this test and all the UVM components are instantiate here. "top_seq_lib.sv is a virtual sequence that generates other sequences, AHB and APB sequences in this case. Details about the virtual sequencer is described later in section 3.3. "top_config" is a UVM object that has interfaces and variables that can be accessed globally through the UVM config database.

The UVCs are instantiated in "top_env.sv", the RTL models is instantiated in "top_th.sv". Stimuli can now be generated in the virtual sequencer to exercise the DUT, but we need additional components to complete the test environment. The predictors are developed as described in section 3.2.2 and instantiated in "top_env.sv". Similar with the scoreboard, it is developed as described in section 3.2.5 and instantiated in "top_env.sv". The DUT is a memory controller, and the memory it is controlling is an RTL flash memory model from one of Nordic Semiconductors vendors. It is 1MB in size and is a complex memory model with many features. A power and clock module from Nordic Semiconductor is instantiated to control the supply of power and clock. It will ensure low power consumption by turning off the supply when not needed. `top_seq_lib.sv` is a virtual UVM sequence and is used to generate AHB and APB sequences. The virtual sequencer is described in the next section. The overview of the complete structure is shown in figure 3.8.

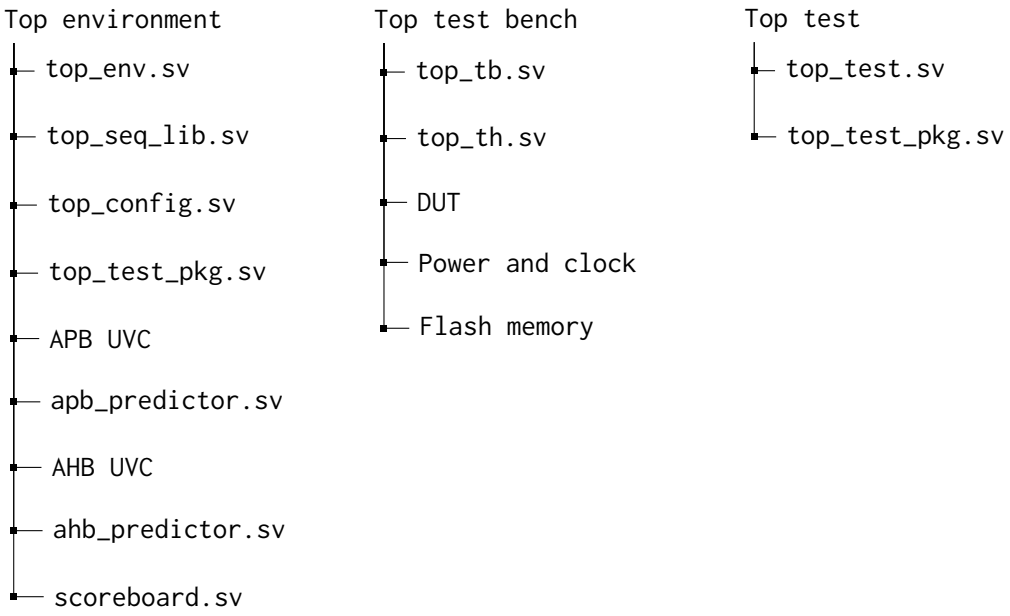


Figure 3.8: Complete structure

The code generator accelerated the development and implemented best practices. Writing the structure manually would be time-consuming and error-prone. The SystemC model is in another folder and is compiled separately, but connects through UVM Connect in the link stage. The block scheme of the structure with the SytemC model, UVM and the DUT is shown in figure 3.9

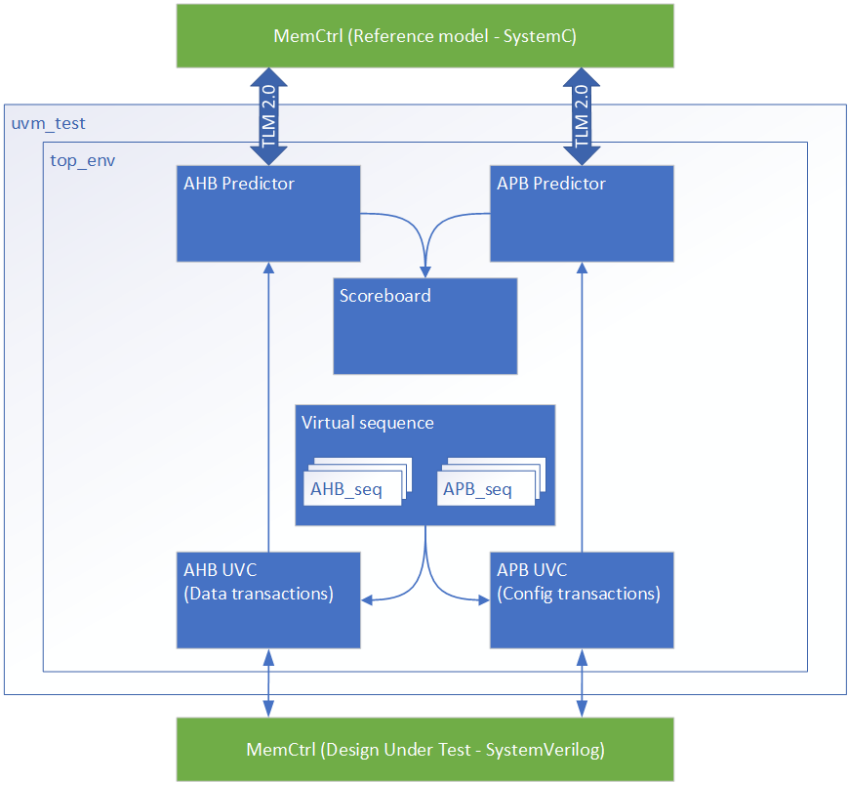


Figure 3.9: Complete block scheme

The constrained random stimuli are generated in the virtual sequencer and simulated on the DUT. It

generates two types of stimuli, APB transactions to the configuration registers and AHB transaction to the flash memory. The test is complete when the DUT's features have been exercised thoroughly, and this is measured by the coverage. The predictors receive the executed transactions from the UVCs and simulate them on the reference model and uses the response to produce the expected response. The responses in this test are the APB and AHB read transactions. Both the expected response and the actual response are sent to the scoreboard for comparison.

3.3 Tests

The CONFIG register in the memory controller can be read and written to, and it has three states that either enable read(state 0), write(state 1) or erase(state 2) of flash memory. This means a random stimuli must be constrained only to write 0, 1 or 2 to the register. The ERASE_ALL register in the memory controller is a write-only register. The flash memory is erased when 1 is written to the ERASE_ALL register, and erase is enabled in the CONFIG register. This means random stimuli must be constrained only to write if the address is the ERASE_ALL register. The constraints on the APB stimuli to prevent illegal stimuli are:

```
rand bit [DATA_BUS_WIDTH-1:0] data;
rand bit [ADDR_WIDTH-1:0] addr;
rand bit pwrite; // 1 -> WRITE, 0 -> READ

constraint c_addr {
  addr inside { CONFIG_REG, ERASE_ALL_REG};
}
constraint c_data {
  data inside { 'h0, 'h1, 'h2};
}
// Constraint to not read from ERASE_ALL_REG, only write
constraint c_pwrite {
  addr inside {ERASE_ALL_REG} -> pwrite == 1;
}
```

Listing 3.1: APB constraints to prevent illegal transactions

With the constraints mentioned above, an exhaustive constrained random test can be performed on the registers. Test1 creates a desired number of stimuli with the following code in the virtual sequencer:

```
begin   // APB - SEND RANDOM READ/WRITE TRANS //
  apb_rand_seq=apb_transfer#(32,32)::type_id::create("apb_rand_seq");
  for(int i = 0; i < NUM_APB_TRANS; i++) begin
    if ( !apb_rand_seq.randomize() )
     `uvm_error(get_type_name(), "Failed to randomize sequence")
    apb_rand_seq.start(m_apb_uvc_agent.sqr, this);
  end
end
```

Listing 3.2: Test1: Constrained random APB sequences

The range of possible input combinations to the ERASE_ALL and CONFIG registers are small and simulating some constrained random sequences could test all combinations, but the coverage should confirm this. Constraints can easily be added to target functionality with the following addition to the randomization:

```
1       if ( !apb_rand_seq.randomize()
2             with {apb_rand_seq.addr = CONFIG_REG;})
```

Listing 3.3: Adding a constraint to target functionality

Constraining the randomization can be important in large tests to target functionality. Test1 exercise a small part of the DUT and the Questa verification tool recorded only 24% code coverage. It is easy to apply efficient constrained random stimuli to the registers since they can be accessed at any time. The flash memory is more difficult to access because there are many illegal scenarios. The operations allowed are: Read from memory if the read is enabled, write to the flash memory if write is enabled and the memory location is erased, erase memory if erase is enabled. The reference model have not modelled illegal transactions, and performing these illegal transactions can result in unpredictable behaviour. The scoreboard could ignore illegal transactions if the reference model recognized the illegal transactions and flagged them. The specification sheet lists legal transactions, and it is up to the user to avoid illegal transactions that could lead to unpredictable behaviour. The restrictions of legal flash transactions results in a manual made test where only the AHB data is random. Test2 tests the memory controller and the flash memory with the following virtual sequence:

```
1   begin // APB - ERASE ALL AND ENABLE WRITE //
2     apb_write_seq=apb_write_transfer #(32,32)::type_id::create("apb_write_seq");
3     apb_write_seq.ta_enable_erase(m_apb_uvc_agent.sqr);
4     apb_write_seq.ta_erase_all(m_apb_uvc_agent.sqr);
5     apb_write_seq.ta_enable_write(m_apb_uvc_agent.sqr);
6   end
7
8   begin // AHB - SEND WRITE TRANS //
9     ahb_write_seq = ahb_write_sequental_reg#(32)::type_id::create("ahb_write_seq");
10    if ( !ahb_write_seq.randomize() )
11      `uvm_error(get_type_name(), "Failed to randomize sequence")
12    ahb_write_seq.start(m_ahb_agent.sqr, this);
13  end
14
15  begin // APB - ENABLE READ //
16    apb_write_seq.ta_enable_read(m_apb_uvc_agent.sqr);
17  end
18
19  begin // AHB - SEND READ TRANS //
20    ahb_read_seq = ahb_read_sequental_reg#(32)::type_id::create("ahb_read_seq");
21    if ( !ahb_read_seq.randomize() )
22      `uvm_error(get_type_name(), "Failed to randomize sequence")
23    ahb_read_seq.start(m_ahb_agent.sqr, this);
24  end
```

Listing 3.4: Test2: AHB and APB sequences with only random AHB data

The first sequence created is `apb_write_seq`, which has pre-defined tasks for specific transactions. The

three first APB transactions are started by tasks and enable erase, erase all flash-memory and enable write. The next sequence created is `ahb_write_sequental_reg`, and this sequence writes random data to a pre-defined number of addresses. The address is incremented to prevent two writes to an address. Writing multiple times to a memory address without erasing results in corrupted data not modelled by the reference model. Simulating such transactions will provoke error that is not caused by a bug, but user error. The next sequence simulated is an APB sequence started by a task and enable read from flash memory. The following sequence created is `ahb_read_sequental_reg`, and this sequence reads data from the addresses previously written to. The stimuli from Test2 resulted in 42% code coverage, meaning a lot of the DUT is still not exercised, but this is expected as the reference model has only the basic functionality of the DUT.

Test2 consist mostly of determined tests testing specific functionality and is similar to the existing traditional testbench with directed tests. Adding sequences to reach specific coverage will inherit some of the same weaknesses of a directed test. Instead of developing manual tests, the reference model should be developed to have the same functionality as the DUT and flag illegal transactions. The virtual sequencer could then be reduced to constrained random APB and AHB sequences:

```
fork
  begin
    APB_SEQ=apb_transfer #(32,32)::type_id::create("APB_SEQ");
    for(int i = 0; i < NUM_APB_TRANS; i++) begin
      if ( !APB_SEQ.randomize() )
       `uvm_error(get_type_name(), "Failed to randomize sequence")
      APB_SEQ.start(m_apb_uvc_agent.sqr, this);
    end
  end

  begin
    AHB_SEQ = ahb_transger#(32)::type_id::create("seq");
    for(int i = 0; i < NUM_AHB_TRANS; i++) begin
      if ( !AHB_SEQ.randomize() )
        `uvm_error(get_type_name(), "Failed to randomize sequence")
      AHB_SEQ.start(m_ahb_agent.sqr, this);
    end
  end
join
```

Listing 3.5: Test3: Constrained random APB and AHB sequences

Test3 is not simulated, but is an example of a test that can be performed with an adequate reference model and test environment. Test3 create constraint random APB and AHB sequences in two concurrent processes.

3.4 Makefile

The makefile of a multi-language environment can be complex and an obstacle for engineers new to the technology and methodology. The engineer will eventually become comfortable with the makefile as test environments have a similar structure. Executing "make all" will delete old files, compile, link, optimize and simulate. A summary and description of the makefile is shown below in listing 3.6.

```
all: clean comp sccom vopt qsim

# comp sccom vopt qsim are operations from Mentor's Questasim verification tool

comp:
    # create work folder
    vlib work

  # vlog compiles SystemVerilog and UVM
  # files.f specify UVM and SystemVerilog files for compilation
  # +define+NON_BLOCKING_TRANSPORT or +define+BLOCKING_TRANSPORT compilation flag
    decide TLM socket technology
  vlog -f ../files.f -sv +define+BLOCKING_TRANSPORT

sccom:
    # sccom compiles SystemC
  sccom

  # include SystemC library and UVM Connect
  I../systemc/sys_lib/src/connect/sc

   # include SystemC source file s
  -I ref_model/src

  # compile cpp files
  all_cpp_files,cpp

   # link the registered SystemC sockets with registered UVM sockets
  sccom -link -uvmc -lib work

vopt:
    # vopt optimize design and enable coverage collection
  vopt sc_main top_tb -o DesignOpt +cover=sbecft

qsim:
    # vsim starts the simulation
  vsim -64 -gui DesignOpt $(UVMC_LIB) -l questa.log

clean:
  rm -rf work $(designfile) $(wavefile) UVM_MESSAGES
  mkdir UVM_MESSAGES
```

Listing 3.6: Summary and description of the makefile

Controlling tests from the makefile can be a useful abstraction. The choice between blocking and non-

blocking communication between SystemC and UVM is controlled with a compilation flag. Implementing variables in the makefile reduced knowledge acquired of the testbench to operate it. The engineer can control the variable from the makefile without the need of modifying the complex testbench.

4 | Results

Simulation with UVM and SystemC was successfully demonstrated. The functionality tested was limited by the reference model's lack of functionality, and only 42% code coverage was reached. Adding more functionality to the reference model would enable greater coverage and completeness.

The system developed enabled the SystemC reference model to provide the expected response from the DUT. One key component developed are the predictors that predicts the expected response with the use of the reference model. It is not the exact transaction that the DUT received, but a high-level representation of the transaction with data and address. The predictor was developed to be reusable in other tests and demonstrate the possibilities with both blocking and non-blocking TLM2.0 sockets. The scoreboard was explicitly designed for this test and succeeded in comparing the response from the DUT and the reference model. This thesis is the first large scale simulation of SystemC and UVM at Nordic Semiconductor, and provides the groundwork that this methodology can be used in verification.

The SystemC model was successfully compiled and simulated, but with some obstacles, because it was not compatible with the verification tool. It was designed with different gcc compiler and SystemC language version than what was available in the verification tool. Future SystemC models must be implemented with compatibility to all tools. The SystemC model was not specifically designed to be compatible with the verification tool, and a lot of time were used to modify it, so it compiled and simulated. UVM is a well-established technology and compiled without many obstacles. The verification tool was generous about giving UVM information in compilation and simulation. However, SystemC information were sometimes misguiding or non-existing. The use of the SystemC reference model can be time-consuming when not correctly adapted to tools. The methodology can be efficient in performing functional verification, but obstacles as SystemC adaptation can limit the efficiency.

A factor in deciding verification methodology is the implementation time. The DUT was previously verified with a traditional directed test that uses 3500 lines to implement test vectors. Removing the need to write enormous lists is a motivation to adopt CRV and reduce implementation time. In Test3[Listing: 3.5], 19 code lines is enough to create a test with constrained random APB and AHB sequence when the reference model is adequate and can replace many directed tests. However, this test requires a large effort made in the test environment. The reason this is still a promising methodology is the fact that a lot of the environment is reusable or automatically generated. Chapter 3.2.6 about code generation demonstrates how to give the any UVM development a flying start. The structure is standardized and enable effective cooperation. UVM and CRV can drastically decrease development time of test structure and test generation, and a SystemC model should give no extra implementation time as it should already exist.

A simple method of calculating implementation time is counting the number of code lines written. Table 4.1 shows the approximate number of code lines in the files. 100% percentage written means I wrote it myself, 50% means the code generator wrote some of it and 0% means it is reused.

Table 4.1: Code lines and implementation time

Percentage written	File(s)	Code lines
100%	apb_predictor.sv	220
100%	ahb_predictor.sv	220
100%	scoreboard.sv	160
50%	top_env.sv	140
50%	top_seq_lib.sv	115
50%	top_config.sv	30
50%	top_tb.sv	50
50%	top_th.sv	440
50%	top_test.sv	20
50%	pkg	45
0%	APB UVC	2700
0%	AHB UVC	2300
0%	SystemC	900

The number of code lines multiplied with the percentage gives approximately the code lines written by me. I have written 1000 of the total 7200 lines, or 86%. A project similar to this project can reuse the predictors and achieve even lower implementation time with 92% reuse. A low implementation time is possible with UVM, code generator and reusable components.

Another factor in deciding verification methodology is the simulation time. Simulation time is a limiting factor in simulation-based verification. The test environment in traditional directed test are usually simple, and the DUT's complexity is the main contributor for high simulation time. UVM has an infrastructure more significant than a traditional directed test and also a SystemC reference model to simulate, this means a slightly higher simulation time is expected. UVM and SystemC use transaction-level communication that are many times faster than RTL simulation. UVM and SystemC could use more simulation resources than a traditional directed test, but the DUT is the main contributor to the simulation time. A more significant reason for the rise in simulation time with this methodology is that CRV would increase the number of test vectors needed for completeness. Measures to decrease the number of test vectors required are presented in the background chapter, but would still not be as efficient as manual made test vectors. Figure 4.1 shows how many minutes to compile and simulate Test2 with different amounts of test vectors. Different transactions use different simulate time, and the APB erase all transaction has the highest simulation time. The x-axis shows how many AHB read and write are performed. Ten thousand on the x-axis include four APB transactions, 5000 AHB write transactions and 5000 AHB read transactions. The y-axis shows the time in minutes.

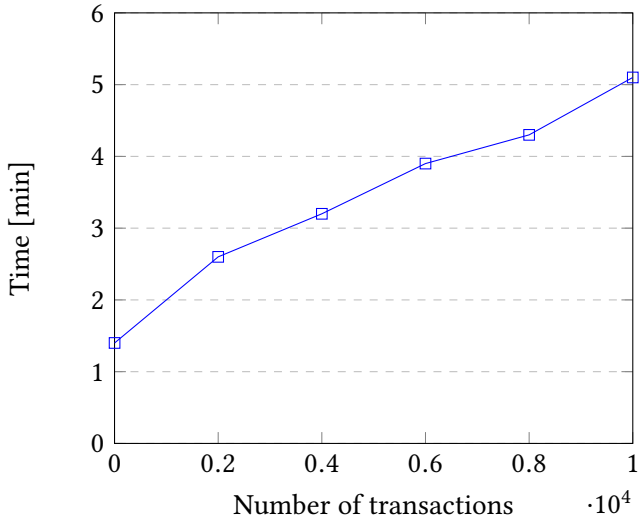


Figure 4.1: Time to complete Test2 with different number of transactions

Many thousands of transactions can be simulated in a couple of minutes on the Nordic Semiconductors servers and can quickly achieve exhausting testing of a small system. Simulating complex SoC can significantly increase the simulation time as each transaction takes longer to simulate, and the test needs more test vectors to test all sub-components. Features discussed in the background as high-quality reusable tests, novel test detection and constraints adjustment can reduce simulation time and achieve higher coverage.

5 | Discussion

Simulation-based tests can be implemented with different levels of complexity. The memory controller simulated in this thesis is a complex design with industry standards and already verified with a traditional directed testbench. The testbench has a long list of directed tests that covers all functionality and is updated over several years. Updating the large lists of fixed test vectors is non-trivial for engineers not familiar with the test. It is fair to say that the existing testbench has high development cost and high maintenance cost. This means it is possible to verify it with directed tests, but is costly and a more efficient methodology should be investigated.

5.1 CRV

CRV reduce or eliminate the need of directed tests and is usually implemented in SystemVerilog or UVM. It is easy and quick to build a single constrained random test that is equivalent to many directed tests, but the CRV environment is more complicated. Some DUTs are too complex for an engineer to comprehend fully, and the generation of directed test are error-prone to unexpected bugs. The randomization can find unexpected bugs, but can also reach unexpected illegal states. Designs today may not have logic to prevent all illegal scenarios, and it is the user's responsibility to prevent illegal states. The stimuli generation have powerful constraints to avoid stimuli that is always illegal, but sometimes the illegal stimuli are dependent on the DUT state and can be challenging to prevent. This is visible in Test2[Listing:3.4], because only the data is random to avoid any illegal scenarios. Individual challenges arrive with every DUT, and it can be challenging to create meaningful constrained random stimuli that have the freedom to find unexpected bugs, and does not stop the simulation by provoking illegal states. The technical difficulties of completing verification in CRV is more challenging than directed tests. Still, the need to tackle these challenges is about time to enhance verification today. The technical challenges are present, but a company will overcome these challenges by investing time and resources in these areas.

5.2 UVM

It can be challenging to complete verification with CRV. The challenges can appear in stimuli generation, response checking, coverage checking or other areas. Different methodologies have different challenges, and a company should choose to invest in a small number of verification methodologies to adopt the methodologies properly. The use of many different methodologies can result in many poorly adopted methodologies. A company with one or few methodologies solves problems when they appear and share

them so coworkers can use the solutions. It is up to the company to enable advanced methodologies since it can be too much for a few persons to solve all challenges in a reasonable amount of time. UVM is a new promising standard verification methodology that is built on SystemVerilog and implements best practises. Company guidelines enhances the benefits from the use of a standard methodology. UVM has a lot of freedom in implementation, but UVM with strict company guidelines enables efficient cooperation between coworkers because many test environments are similar. These test environments will arguably be improved after each project as engineers find smart solutions and share with coworkers.

Some downsides of UVM is complexity and expensive training. SystemVerilog can indeed provide all features an engineer need. The transition from SystemVerilog to UVM is arguably not worth the resources spent if only a small percentage of verification engineers do the transition. The result then is just more unique testbenches with unique problems. The goal should be to standardize the verification methodology used in the company. Also, the simulation time can be expected to grow since the test environments are more resource-heavy, and more test vectors are maybe required to complete verification. Both implementation time and simulation time are important in reaching the required verification time.

UVM did not invent reuse, but enhances it. The UVCs used in this thesis provoked no challenges, but that was after some trial and error with bad UVCs. At the start of this thesis, no standard AHB or APB were made at nordic and the first I tried lacked features, had no documentation and I had to read through all of the code to understand if I could use it or not. The difference between an excellent reusable UVC and a bad UVC are huge. One reduces the implementation time, and one increases implementation time. This is one more example of a whole company must make verification efficient together as a team, and remove time-consuming tasks.

Starting with UVM is a daunting process where object-oriented programming and polymorphism is essential, and a whole bunch of standard classes must be understood. However, arguably all the standard classes are intuitive, and the best-practise classes are already made, so the object-oriented part are feasible to learn. The tedious process of making the large infrastructure needed to simulate anything is made easy with code generators, and make UVM usable for everyone. A lot of investment is needed to be efficient in UVM.

Several techniques of enabling CRV with automatic checking are possible, a high-level model or register model are the most used. UVM register models are simple and can be the most efficient when modelling DUTs with registers. The DUT in this thesis could be modelled with a register model and predictor to produce expected result. This would be efficient in is this scenario, but this thesis is investigating the SystemC technology and not the DUT. A company that uses SystemC and UVM in verification can also use other setups as register models and directed testing where it is efficient. However, a lot must be invested in the technology presented in this thesis to make it efficient, and it would then make sense to use it widely across the company.

5.3 SystemC

Of the CRV challenges are the prediction of the response the most challenging. A high-level model can be implemented in various languages, as described in chapter 2.2.5. They have different challenges

and opportunities. SystemC is arguably the most challenging and most rewarding solution. It is most rewarding because it can be used in many design phases by different groups. It has unique hardware mimicking opportunities that can perform functional verification.

The SystemC model compiled easily with the gcc compiler it was designed for, but was a challenge to compile in the verification tool. The model was developed with different language and compiler versions than available in the verification tool, and it was developed without the intended use in mind. It is a complex task to compile the SystemC model that includes c++, SystemC, TLM2.0 and UVM Connect. A verification tool has approximately four major updates a year, and nearly every version gave different errors. UVM is the main use in this verification tool and has matured, but the SystemC is a long way from the same level of consistency. The SystemC compilation and simulation information was often misguiding or non-existent. It gave errors on internal compiler files inaccessible by the user and could end the simulation without error. Resources must be spent to investigate what are the possibilities on what versions. The SystemC technology must be matured and show consistency to be efficient. Just as the undocumented UVCs increase the implementation time, increase an undocumented SystemC model the implementation time. The SystemC model had documented gcc compatibility, but none of the verification tool versions. Again, a company must use considerable resources to make the model usable because a verification engineer does not have time to read and understand a complex SystemC model.

6 | Conclusion and Future work

Developing a test with SystemC and UVM from the ground up to verify the functionality of an RTL model is undoubtedly a resource-heavy task for verification engineers. SystemC modelling is time-consuming, UVM development is complex and compiling and simulating together introduce different challenges. However, this thesis describes implementing tests when the use of UVM and SystemC is the standard methodology in a company and resources are available. System architects develop SystemC models for multi-purpose, reusable UVCs are available, and a standard method of compiling, simulating and achieving completeness is developed. These prerequisites are necessary to achieve completeness quickly. Companies unable to create the essential infrastructure this methodology needs, should arguably not depend their verification on this methodology.

Nordic Semiconductor's UVCs ensured easy and quick connection between UVM and the DUT, and made it easy to generate stimuli in the desired protocol. UVM has matured in the company and resulted in easy use of the technology. The SystemC technology provided difficulties because it has not matured as a technology in verification. Firstly, the model was not designed for use in verification tools, and initial compilation was a big challenge. Secondly, the model had only the basic functionality of the DUT, making it impossible to complete the verification. The verification methodology presented in this thesis shows great promise in reducing verification time by eliminating the process of manually writing test vectors and enabling reuse. The SystemC model can be a limiting factor as a considerable amount of resources must be invested in SystemC development to enable this methodology.

6.1 Future work

- Further develop SystemC model to complete verification of the DUT.
- Investigate the challenges and solutions to illegal scenarios in the DUT.
- Investigate coverage methods to indicate completeness.
- Investigate TLM 2.0 interoperability and easy connection to additional blocks.
- Apply novel test detection to decrease the number of test vectors.
- Implement a test database for high-quality tests.

Bibliography

[1] H. Foster, "Part 8: The 2018 Wilson Research Group Functional Verification Study," https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verifi- cation-study/ (visited: 6.6.2020).

[2] ——, "How do you spell UVM? Opportunities in professional development.," https://blogs.mentor.com/verificationhorizons/blog/2020/03/31/how-do-you-spell-uvm-opportunities-in-professional-development/ (visited: 19.05.2020).

[3] A. S. Initiative, "Portable Stimulus Working Group," https://www.accellera.org/activities/working-groups/portable-stimulus (visited: 19.05.2020).

[4] "Novel test detection to improve simulation efficiency — a commercial experiment," *IEEE 6386595*, Dec. 2012.

[5] Doulos, "What is SystemVerilog?," https://www.doulos.com/knowhow/sysverilog/whatissv/ (visited: 6.6.2020).

[6] A. S. Initiative, "Universal Verification Methodology," https://www.mentor.com/products/fv/uvm (visited: 19.05.2020).

[7] A. standards, "UVM," https://www.accellera.org/downloads/standards/uvm (visited: 19.05.2020).

[8] M. Graphics, "Universal Verification Methodology (UVM)," https://www.mentor.com/products/fv/uvm (visited: 19.05.2020).

[9] ——, "UVM Connect," https://www.mentor.com/products/fv/multimedia/uvm-connect (visited: 19.05.2020).

[10] Doulus, "Easier UVM," https://www.doulos.com/content/events/easierUVM.php (visited: 19.05.2020).

[11] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[12] A. standards, “SystemC and TLM2.0,” www.accellera.org/downloads/standards/systemc (visited: 19.05.2020).

[13] Accellera, “Virtual System Platform,” https://www.cadence.com/en_US/home/tools/system-design-and-verification /software-driven-verification/virtual-system-platform.html (visited: 19.05.2020).

[14] Cadence, “Cadence Virtual System Platform virtual prototyping solution,” https://www.cadence.com/en_US/home/tools/system-design-and-verification /software-driven-verification/virtual-system-platform.html (visited: 19.05.2020).

[15] S. Swan, “Systemc transaction level models and rtl verification,” *IEEE 1688767*, Jul. 2006.

[16] A. Moursi, R. Samhoud, Y. Kamal, M. Magdy, S. El-Ashry, and A. Shalaby, “Different reference models for uvm environment to speed up the verification time,” *IEEE 8746056*, Dec. 2018.

A | Source code

```
import uvm_pkg::*;

class ahb_predictor extends uvm_component;
   `uvm_component_utils (ahb_predictor)

   // - output port with the actual DUT response - //
   uvm_analysis_port #(uvm_tlm_gp) broadcast_AHB_actual;

   // - output port with the expected refence response - //
   uvm_analysis_port #(uvm_tlm_gp) broadcast_AHB_expected;

   // - input port with UVC trans - //
   uvm_analysis_imp  #(ahb_item, ahb_predictor)   imp_trans;

   // - sockets to connect to SC - //
   `ifdef NON_BLOCKING_TRANSPORT
     uvm_tlm_nb_initiator_socket #(ahb_predictor) nb_data_socket;
   `endif
   `ifdef BLOCKING_TRANSPORT
      uvm_tlm_b_initiator_socket #() data_socket;
   `endif

   uvm_tlm_gp trans_ahb_queue[$];
   bit sc_is_ready = 1;
   bit end_resp_phase;
   virtual in_ahb vif;
   top_config m_config;

   function new (string name = "ahb_predictor", uvm_component parent= null);
      super.new (name, parent);
      broadcast_AHB_actual    = new("broadcast_AHB_actual", this);
      broadcast_AHB_expected  = new("broadcast_AHB_expected", this);
      imp_trans               = new("imp_trans", this);

      `ifdef BLOCKING_TRANSPORT
         data_socket      = new("data_socket", this);
      `endif
      `ifdef NON_BLOCKING_TRANSPORT
        nb_data_socket    = new("nb_data_socket", this);
      `endif
```

```
41
42   endfunction
43
44
45  virtual function void build_phase(uvm_phase phase);
46    super.build_phase(phase);
47    `uvm_info(get_type_name(), "..is alive!", UVM_DEBUG)
48
49    if (!uvm_config_db #(top_config)::get(this, "", "m_config", m_config))
50      `uvm_error(get_type_name(), "ahb config not found")
51
52    if(m_config != null) begin
53       if(m_config.ahb_vif != null) begin
54         $cast(vif, m_config.ahb_vif);
55       end
56    end
57
58  endfunction
59
60   `ifdef NON_BLOCKING_TRANSPORT
61      virtual function uvm_tlm_sync_e nb_transport_bw( uvm_tlm_gp data_trans, ref
    uvm_tlm_phase_e ph, uvm_tlm_time delay);
62
63         if(ph == END_REQ) begin
64            `uvm_info(get_type_name(), $sformatf("nb_transport_bw END_REQ phase,
    return UVM_TLM_ACCEPTED"), UVM_HIGH)
65            return UVM_TLM_ACCEPTED;
66         end else if(ph == BEGIN_RESP && data_trans.is_response_ok()) begin
67
68            if(data_trans.get_command==UVM_TLM_READ_COMMAND) begin
69               fu_reverse_byte_order(data_trans);
70               broadcast_AHB_expected.write(data_trans);
71            end
72
73            end_resp_phase = 1;
74            return UVM_TLM_ACCEPTED;
75         end
76
77         `uvm_error(get_type_name(), $sformatf("ERROR, incorrect state with phase: %
    s, and response %s.", ph, data_trans.get_response_string()))
78
79      endfunction
80   `endif
81
82  // - RECEIVE COMPLETED TRANS FROM APB UVC MONITOR - //
83   virtual function void write(ahb_item trans);
84      uvm_tlm_gp          gp_to_ref_model = new;
85      uvm_tlm_gp          gp_to_scrb = new;
86      uvm_tlm_gp          t_copy = new;
87
88      bit is_write;
89      is_write = trans.ahbHWrite;
```

```
90
91      // if WRITE transactions make generic payload for ref model
92      if(is_write == 1) begin
93        fu_set_gp_for_ref_model(trans, gp_to_ref_model);
94
95      // else if READ transactions make seperate generic payloads for ref model and
    scoreboard
96      end else if (is_write == 0) begin
97        fu_set_gp_for_ref_model(trans, gp_to_ref_model);
98        fu_set_gp_for_scrb(trans, gp_to_scrb);
99        broadcast_AHB_actual.write(gp_to_scrb);
100     end else begin
101       `uvm_error(get_type_name(), "apb trans not read or write")
102     end
103
104     // Systemc has different endian
105     fu_reverse_byte_order(gp_to_ref_model);
106     trans_ahb_queue.push_back(gp_to_ref_model);
107     `uvm_info(get_type_name(), $sformatf("TRANS QUEUED = %s ;", gp_to_ref_model.
    convert2string()), UVM_DEBUG)
108   endfunction
109
110   function void fu_reverse_byte_order(uvm_tlm_gp data_trans);
111     byte unsigned temp_data[];
112     int reverse;
113
114     data_trans.get_data(temp_data);
115     reverse = {<<8{temp_data}};
116     temp_data = {>>{reverse}};
117     data_trans.set_data(temp_data);
118   endfunction
119
120 // - set generic payload attributes to be sendt to ref model - //
121   function void fu_set_gp_for_ref_model(ahb_item ahb_trans, uvm_tlm_gp gp_trans);
122     bit [127:0]    bit_data;
123     int           int_data;
124     byte unsigned byte_data[];
125     int           address;
126     bit           is_write;
127
128     is_write   = ahb_trans.ahbHWrite;
129     address    = ahb_trans.ahbHAddr;
130
131     bit_data   = {>>{ahb_trans.ahbHWData}};
132     int_data   = bit_data;
133     byte_data = {>>{int_data}}; // stream data
134
135     gp_trans.set_data_length(4);
136     gp_trans.set_streaming_width(4); // = data_length to indicate no streaming
137     gp_trans.set_address(address);
138
139     if(is_write == 1) begin
```

```
140         gp_trans.set_data(byte_data);
141         gp_trans.set_command(UVM_TLM_WRITE_COMMAND);
142       end else if (is_write == 0) begin
143         gp_trans.set_command(UVM_TLM_READ_COMMAND);
144       end else begin
145         `uvm_error(get_type_name(), "apb trans not read or write")
146       end
147    endfunction
148
149   // - set generic payload attributes to be sendt to scoreboard - //
150   function void fu_set_gp_for_scrb(ahb_item ahb_trans, uvm_tlm_gp gp_trans);
151       bit [127:0]    bit_data;
152       int            int_data;
153       byte unsigned  byte_data[];
154       int            address;
155
156       address    = ahb_trans.ahbHAddr;
157
158       bit_data   = {>>{ahb_trans.ahbHRData}};
159       int_data   = bit_data;
160       byte_data  = {>>{int_data}}; // stream data
161
162       gp_trans.set_data_length(4);
163       gp_trans.set_streaming_width(4); // = data_length to indicate no streaming
164       gp_trans.set_address(address);
165       gp_trans.set_data(byte_data);
166       gp_trans.set_command(UVM_TLM_READ_COMMAND);
167       gp_trans.set_response_status(UVM_TLM_OK_RESPONSE);
168    endfunction
169
170    virtual task run_phase (uvm_phase phase);
171       uvm_tlm_time        delay  = new;
172       uvm_tlm_gp          trans_to_SC  = new;
173
174       uvm_tlm_sync_e status;
175       uvm_tlm_phase_e phase1;
176
177       byte unsigned temp_data[];
178       int reverse;
179
180       forever begin
181          @(posedge vif.ckAhb);
182
183             if(end_resp_phase==1) begin
184                `ifdef NON_BLOCKING_TRANSPORT
185                   end_resp_phase = 0;
186                   sc_is_ready = 1;
187                   phase1 = END_RESP;
188                   //`uvm_info(get_type_name(), $sformatf("APB before nb_transport_fw
     = %s.", trans_to_SC.convert2string()), UVM_HIGH)
189                   status = nb_data_socket.nb_transport_fw( trans_to_SC, phase1,
    delay);
```

```
190                 `uvm_info(get_type_name(), $sformatf("after nb_data_socket = %s,
    and status: %s, and data_trans.get_response_string() %s.", trans_to_SC.
    convert2string(), status, trans_to_SC.get_response_string()), UVM_HIGH)
191              `endif
192           end
193           else if(sc_is_ready == 1 && trans_ahb_queue.size() > 0) begin
194              `uvm_info(get_type_name(), $sformatf("AHB PREDICTOR STARTING TO SEND
    TRANS TO SC, queue size = %x", trans_ahb_queue.size()), UVM_HIGH)
195
196              trans_to_SC = trans_ahb_queue.pop_front();
197
198              `ifdef NON_BLOCKING_TRANSPORT
199                 sc_is_ready = 0;
200                 phase1 = BEGIN_REQ;
201                 status = nb_data_socket.nb_transport_fw( trans_to_SC, phase1,
    delay);
202                 `uvm_info(get_type_name(), $sformatf("after nb_data_socket = %s,
    and status: %s, and data_trans.get_response_string() %s.", trans_to_SC.
    convert2string(), status, trans_to_SC.get_response_string()), UVM_HIGH)
203              `endif
204
205              `ifdef BLOCKING_TRANSPORT
206                 // Send transaction to SC model
207                 data_socket.b_transport( trans_to_SC, delay);
208                 `uvm_info(get_type_name(), $sformatf("after b_transport = %s.",
    trans_to_SC.convert2string()), UVM_HIGH)
209
210                 // Brodcast READ transaction to SCOREBOARD
211                 if (trans_to_SC.is_read()) begin
212
213                    // Systemc has different endian
214                    fu_reverse_byte_order(trans_to_SC);
215
216                    broadcast_AHB_expected.write(trans_to_SC);
217                 end
218              `endif
219           end
220     end
221
222   endtask
223
224 endclass
```

Listing A.1: AHB predictor with both blocking and non-blocking interface implemented

```
1 import uvm_pkg::*;
2
3 class ahb_predictor extends uvm_component;
4    `uvm_component_utils (ahb_predictor)
5
6    // - output port with the actual DUT response - //
7    uvm_analysis_port #(uvm_tlm_gp) broadcast_AHB_actual;
8
```

```
  // - output port with the expected refence response - //
  uvm_analysis_port #(uvm_tlm_gp) broadcast_AHB_expected;

 // - input port with UVC trans - //
  uvm_analysis_imp  #(ahb_item, ahb_predictor)   imp_trans;

 // - socket to connect to SC - //
  uvm_tlm_b_initiator_socket #() data_socket;

  uvm_tlm_gp trans_ahb_queue[$];
  virtual in_ahb vif;
  top_config m_config;

  function new (string name = "ahb_predictor", uvm_component parent= null);
     super.new (name, parent);
     broadcast_AHB_actual    = new("broadcast_AHB_actual", this);
     broadcast_AHB_expected  = new("broadcast_AHB_expected", this);
     imp_trans               = new("imp_trans", this);
     data_socket             = new("data_socket", this);
  endfunction

 virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   `uvm_info(get_type_name(), "..is alive!", UVM_DEBUG)

   if (!uvm_config_db #(top_config)::get(this, "", "m_config", m_config))
     `uvm_error(get_type_name(), "ahb config not found")

 endfunction
 // - RECEIVE COMPLETED TRANS FROM APB UVC MONITOR - //
  virtual function void write(ahb_item trans);
     uvm_tlm_gp          gp_to_ref_model = new;
     uvm_tlm_gp          gp_to_scrb = new;
     uvm_tlm_gp          t_copy = new;

     bit is_write;
     is_write = trans.ahbHWrite;

     // if WRITE transactions make generic payload for ref model
     if(is_write == 1) begin
       fu_set_gp_for_ref_model(trans, gp_to_ref_model);

     // else if READ transactions make seperate generic payloads for ref model and
   scoreboard
     end else if (is_write == 0) begin
       fu_set_gp_for_ref_model(trans, gp_to_ref_model);
       fu_set_gp_for_scrb(trans, gp_to_scrb);
       broadcast_AHB_actual.write(gp_to_scrb);
     end else begin
       `uvm_error(get_type_name(), "apb trans not read or write")
     end

```

```
60      // Systemc has different endian
61      fu_reverse_byte_order(gp_to_ref_model);
62      trans_ahb_queue.push_back(gp_to_ref_model);
63      `uvm_info(get_type_name(), $sformatf("TRANS QUEUED = %s ;", gp_to_ref_model.
    convert2string()), UVM_DEBUG)
64    endfunction
65
66  // - set generic payload attributes to be sendt to ref model - //
67    function void fu_set_gp_for_ref_model(ahb_item ahb_trans, uvm_tlm_gp gp_trans);
68      bit [127:0]    bit_data;
69      int            int_data;
70      byte unsigned  byte_data[];
71      int            address;
72      bit            is_write;
73
74      is_write   = ahb_trans.ahbHWrite;
75      address    = ahb_trans.ahbHAddr;
76
77      bit_data   = {>>{ahb_trans.ahbHWData}};
78      int_data   = bit_data;
79      byte_data = {>>{int_data}}; // stream data
80
81      gp_trans.set_data_length(4);
82      gp_trans.set_streaming_width(4); // = data_length to indicate no streaming
83      gp_trans.set_address(address);
84
85      if(is_write == 1) begin
86        gp_trans.set_data(byte_data);
87        gp_trans.set_command(UVM_TLM_WRITE_COMMAND);
88      end else if (is_write == 0) begin
89        gp_trans.set_command(UVM_TLM_READ_COMMAND);
90      end else begin
91        `uvm_error(get_type_name(), "apb trans not read or write")
92      end
93    endfunction
94
95  // - set generic payload attributes to be sendt to scoreboard - //
96  function void fu_set_gp_for_scrb(ahb_item ahb_trans, uvm_tlm_gp gp_trans);
97      bit [127:0]    bit_data;
98      int            int_data;
99      byte unsigned  byte_data[];
100     int            address;
101
102     address    = ahb_trans.ahbHAddr;
103
104     bit_data   = {>>{ahb_trans.ahbHRData}};
105     int_data   = bit_data;
106     byte_data  = {>>{int_data}}; // stream data
107
108     gp_trans.set_data_length(4);
109     gp_trans.set_streaming_width(4); // = data_length to indicate no streaming
110     gp_trans.set_address(address);
```

```
111       gp_trans.set_data(byte_data);
112       gp_trans.set_command(UVM_TLM_READ_COMMAND);
113       gp_trans.set_response_status(UVM_TLM_OK_RESPONSE);
114    endfunction
115
116   // - Reverse byte order of data - //
117    function void fu_reverse_byte_order(uvm_tlm_gp data_trans);
118       byte unsigned temp_data[];
119       int reverse;
120
121       data_trans.get_data(temp_data);
122       reverse = {<<8{temp_data}};
123       temp_data = {>>{reverse}};
124       data_trans.set_data(temp_data);
125    endfunction
126
127    virtual task run_phase (uvm_phase phase);
128       uvm_tlm_time        delay  = new;
129       uvm_tlm_gp          trans_to_SC  = new;
130
131       uvm_tlm_sync_e status;
132       uvm_tlm_phase_e phase1;
133
134       byte unsigned temp_data[];
135       int reverse;
136
137       forever begin
138          @(posedge trans_ahb_queue.size());
139             if(trans_ahb_queue.size() > 0) begin
140                trans_to_SC = trans_ahb_queue.pop_front();
141
142               // Send trans_to_SC to the Reference model
143                data_socket.b_transport( trans_to_SC, delay);
144                //`uvm_info(get_type_name(), $sformatf("after b_transport = %s.",
    trans_to_SC.convert2string()), UVM_HIGH)
145
146                // Brodcast READ transaction to SCOREBOARD
147                if (trans_to_SC.is_read()) begin
148
149                   // Systemc has different endian
150                   fu_reverse_byte_order(trans_to_SC);
151                   broadcast_AHB_expected.write(trans_to_SC);
152                end
153             end
154       end
155    endtask
156 endclass
```

Listing A.2: AHB predictor with blocking interface implemented

```
1 `ifndef SCOREBOARD_SV
2 `define SCOREBOARD_SV
3
```

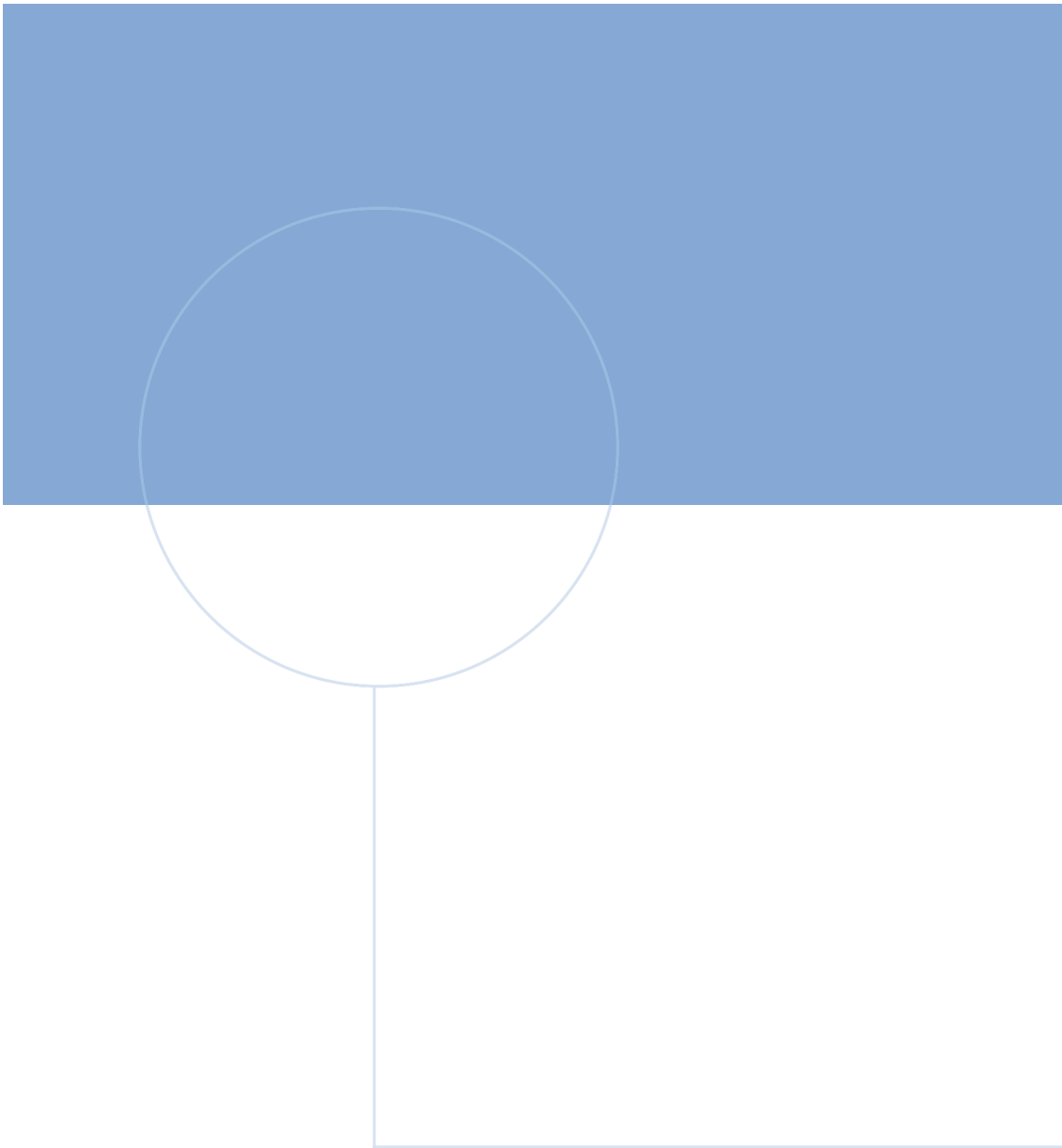
```
import uvm_pkg::*;
`include "uvm_macros.svh"

`uvm_analysis_imp_decl(_ahb_expected)
`uvm_analysis_imp_decl(_ahb_actual)
`uvm_analysis_imp_decl(_apb_expected)
`uvm_analysis_imp_decl(_apb_actual)

class scoreboard extends uvm_scoreboard;

   int num_ahb_compares;
   int num_apb_compares;
   int num_successfull_ahb_compares;
   int num_successfull_apb_compares;

   int NUM_AHB_TRANS;
   int NUM_APB_TRANS;

   uvm_tlm_gp queue_ahb_expected[$];
   uvm_tlm_gp queue_ahb_actual[$];
   uvm_tlm_gp queue_apb_expected[$];
   uvm_tlm_gp queue_apb_actual[$];

   uvm_analysis_imp_ahb_expected #(uvm_tlm_gp,scoreboard) ahb_expected;
   uvm_analysis_imp_ahb_actual #(uvm_tlm_gp,scoreboard) ahb_actual;
   uvm_analysis_imp_apb_expected #(uvm_tlm_gp,scoreboard) apb_expected;
   uvm_analysis_imp_apb_actual #(uvm_tlm_gp,scoreboard) apb_actual;

   `uvm_component_utils(scoreboard)

   uvm_phase run_ph;
   top_config m_config;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      ahb_expected = new("ahb_expected", this);
      ahb_actual   = new("ahb_actual", this);
      apb_expected = new("apb_expected", this);
      apb_actual   = new("apb_actual", this);
      run_ph = uvm_run_phase::get();
   endfunction : new

   function void build_phase(uvm_phase phase);
    `uvm_info(get_type_name() ,"Building scoreboard", UVM_MEDIUM)

      if (!uvm_config_db #(top_config)::get(this, "", "m_config", m_config))
        `uvm_error(get_type_name(), "ahb config not found")

      NUM_AHB_TRANS = m_config.NUM_AHB_TRANS;
      NUM_APB_TRANS = m_config.NUM_APB_TRANS;
   endfunction

```

```
56    virtual function void write_ahb_expected(uvm_tlm_gp trans);
57      uvm_tlm_gp t_copy;
58      $cast(t_copy,trans.clone());
59      `uvm_info(get_type_name(), $sformatf("SB/AHB_expected/RECV = %s ;", t_copy.
      convert2string()), UVM_HIGH)
60      queue_ahb_expected.push_back(t_copy);
61    endfunction
62
63    virtual function void write_ahb_actual(uvm_tlm_gp trans);
64      uvm_tlm_gp t_copy;
65      $cast(t_copy,trans.clone());
66      `uvm_info(get_type_name(), $sformatf("SB/AHB_ACTUAL/RECV = %s ;", t_copy.
      convert2string()), UVM_HIGH)
67      queue_ahb_actual.push_back(t_copy);
68    endfunction
69
70       virtual function void write_apb_expected(uvm_tlm_gp trans);
71      uvm_tlm_gp t_copy;
72      $cast(t_copy,trans.clone());
73      `uvm_info(get_type_name(), $sformatf("SB/APB_expected/RECV = %s ;", t_copy.
      convert2string()), UVM_HIGH)
74      queue_apb_expected.push_back(t_copy);
75    endfunction
76
77    virtual function void write_apb_actual(uvm_tlm_gp trans);
78      uvm_tlm_gp t_copy;
79      $cast(t_copy,trans.clone());
80      `uvm_info(get_type_name(), $sformatf("SB/APB_ACTUAL/RECV = %s ;", t_copy.
      convert2string()), UVM_HIGH)
81      queue_apb_actual.push_back(t_copy);
82    endfunction
83
84    virtual task run_phase(uvm_phase phase);
85     // The transction later compared
86     uvm_tlm_gp excpected_ahb_trans, actual_ahb_trans;
87     uvm_tlm_gp excpected_apb_trans, actual_apb_trans;
88
89     // raise objection to ensure all trans compared
90     if(NUM_AHB_TRANS > 0) begin
91       phase.raise_objection(this);
92     end
93
94     //if(NUM_APB_TRANS > 0) begin
95     //  phase.raise_objection(this);
96     //end
97
98     // - FORK APB AND AHB PROCESS - //
99     fork
100      forever begin
101        @(queue_ahb_actual.size() && queue_ahb_expected.size());
102
103        if (queue_ahb_actual.size() && queue_ahb_expected.size()) begin
```

```
104         excpected_ahb_trans = queue_ahb_expected.pop_front();
105         actual_ahb_trans    = queue_ahb_actual.pop_front();
106         num_ahb_compares    = num_ahb_compares + 1;
107 
108           if (!actual_ahb_trans.compare(excpected_ahb_trans)) begin
109             `uvm_error("SB/MISCOMPARE", $sformatf("Miscompares\nexpect=%s\nactual
    =%s",excpected_ahb_trans.convert2string(),actual_ahb_trans.convert2string()))
110           end else begin
111             `uvm_info("SB/AHB/COMPARE","Actual and expected is equvalent",UVM_HIGH
    )
112             num_successfull_ahb_compares = num_successfull_ahb_compares + 1;
113           end
114       end
115 
116       `uvm_info(get_type_name(), $sformatf("Number of succesfull compares: %x, of
    total compares: %x",num_successfull_ahb_compares, num_ahb_compares), UVM_HIGH)
117 
118       if(num_ahb_compares == NUM_AHB_TRANS) begin
119         if(num_successfull_ahb_compares == NUM_AHB_TRANS) begin
120           `uvm_info(get_type_name(),$sformatf("ALL %d(dec) AHB TEST PASSED.",
    num_successfull_ahb_compares),UVM_LOW)
121           `uvm_info(get_type_name(),$sformatf("Number of succesfull APB tests %d(
    dec).", num_successfull_apb_compares),UVM_LOW)
122         end else begin
123           `uvm_error(get_type_name(), $sformatf("Not all test passed. %x passed of
     %x tests.",num_successfull_ahb_compares, num_ahb_compares))
124         end
125 
126         `uvm_info(get_type_name(),"Done comparing AHB. Dropping objection.",
    UVM_LOW)
127         phase.drop_objection(this);
128       end
129     end // AHB - end forever begin //
130 
131     forever begin
132       @(queue_apb_actual.size() && queue_apb_expected.size());
133 
134       if (queue_apb_actual.size() && queue_apb_expected.size())  begin
135         excpected_apb_trans = queue_apb_expected.pop_front();
136         actual_apb_trans    = queue_apb_actual.pop_front();
137         num_apb_compares    = num_apb_compares + 1;
138 
139           if (!actual_apb_trans.compare(excpected_apb_trans)) begin
140             `uvm_error("SB/MISCOMPARE", $sformatf("Miscompares\nexpect=%s\nactual
    =%s",excpected_apb_trans.convert2string(),actual_apb_trans.convert2string()))
141           end else begin
142             `uvm_info("SB/APB/COMPARE", $sformatf("Actual and expected is
    equvalent, Trans: %s",actual_apb_trans.convert2string()),UVM_HIGH)
143             num_successfull_apb_compares = num_successfull_apb_compares + 1;
144           end
145       if(num_apb_compares == NUM_APB_TRANS) begin
146         if(num_successfull_apb_compares == NUM_APB_TRANS) begin
```

```
147             `uvm_info(get_type_name(),$sformatf("ALL %d(dec) APB TEST PASSED.",
    num_successfull_apb_compares),UVM_LOW)
148           end else begin
149             `uvm_error(get_type_name(), $sformatf("Not all test passed. %x passed of
     %x tests.",num_successfull_apb_compares, num_apb_compares))
150           end
151
152           `uvm_info(get_type_name(),"Done comparing ABP. Dropping objection.",
    UVM_LOW)
153           phase.drop_objection(this);
154         end
155         end
156       end // APB - end forever begin //
157
158     join_none // - END FORK - //
159    endtask
160 endclass
161
162 `endif // SCOREBOARD_SV
```

Listing A.3: UVM scoreboard implementation



NTNU

Norwegian University of
Science and Technology