NTNU

Norwegian University of
Science and Technology

Development of a Universal Verification Component for CPU UVM Verification

Bernhard Bakken

Master of Science in Electronics
Submission date: July 2018
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Vitaly Marchuk, Microchip Technologies Norway AS

Norwegian University of Science and Technology
Department of Electronic Systems

PROJECT ASSIGNMENT

Candidate name: Bernhard Bakken
Assignment title: Development of a Universal Verification Component for CPU UVM verification
Assignment text: The current UVM CPU verification framework contains several agents for working with different bus types. They were developed in accordance with earlier UVM methodology. Recent development showed possibility of creation of a single unified agent working with various buses by encapsulating bus functionality into interfaces, thus substantially reducing amount of resources required for developing and maintaining verification components.

1. Student needs to create universal verification component (UVC agent) by moving functionality from drivers and monitors to bus.
2. Modify current verification framework to work with UVC.
3. Find optimal way of configuring UVC to operate on different buses by using UVM configuration database.
4. Show benefits in terms of code reduction.
5. Compare advantages and disadvantages of both methodologies.

Prerequisites: understanding of CPU architecture, SystemVerilog for design and verification, basic understanding of UVM, some scripting language as Perl or Python would be an advantage.

Assigment proposer/Co-supervisor: Vitaly Marchuk, Microchip Technology Norway AS
Supervisor: Kjetil Svarstad

Abstract

The time used debugging and developing testbenches in FPGA and ASIC/IC projects is around 60% of the total time spent in verification. The last years has shown an increase in the adoption of the Universal Verification Methodology(UVM), which can help increase the maximum reuse, and decrease the time spent creating and debugging testbenches.

This thesis presents the development of a Universal Verification Component(UVC) for CPU UVM verification. An existing framework has been used as a base in the development. In the project an improvement of the existing sequence items has been done by creating a new sequence item that is functional for several protocols, and can be extended for additional functionality. Specialised functionality, that can be called through tasks, has been moved from the driver and monitor to the interface, which helps create generic components. A task implemented in the interface, which converts signals from the DUT to a sequence item, is called from the monitor and used to write to the analysis ports. There has also been developed a sequencer that is functional for various protocols in the framework. Parameterisation has been utilised for all the components in the hierarchy, in order to grant the specialised functionality for the protocols. The agent, which is parameterised with a configuration object and sequence item, builds the testbench with configuration that has been retrieved from the configuration database, by using the the handle for the various protocols' agent to find the correct path.

The amount of code lines has been used to quantify some of the efficiency of the UVC, as a decrease in code lines would most likely result in less bugs, and therefore less time spent debugging. There was a total of 28% decrease in amount of code lines for the protocols, when compared to the base framework. 21 files in the framework have also been replaced by the generic components developed in this project.

The UVC that has been developed, can be reused in several frameworks with its parameterisation. The framework that adopts the UVC has to set the configuration in the configuration database as presented in this thesis for correct functionality. This UVC can decrease the time used debugging and creating testbenches, and the time saved can help the verification engineer to reach deadline, or be used to further improve the quality of the tests.

Sammendrag

Tiden som blir brukt i debugging og å lage testbenker for FPGA- og ASIC/IC-prosjekter er rundt 60% av den totale tiden som går med i verifikasjon. De siste årene har vist en økning av å ta i bruk Universal Verification Methodology(UVM), som kan hjelpe i å gi en økning av gjenbruk, og senke tiden som blir brukt i å lage og debugge testbenker.

Denne avhandlingen presenterer utviklingen av en Universal Verification Component(UVC) for CPU UVM verifikasjon. Ett eksisterende rammeverk har blitt brukt som grunnlag i utviklingen. I prosjektet har en forbedring av ett eksisterende sequence item blitt gjort, slik at den fungerer for flere protokoller eller kan bli utvidet for mer funksjonalitet. Spesialisert funksjonalitet, som kan bli kalt gjennom funksjoner, har blitt flyttet fra driveren og monitoren til grensesnittet, som hjelper lage en generisk component som kan bli brukt av flere protokoller. En funksjon som konverterer signaler fra DUTen til sequence items er implementert i grensesnittet, er kalt fra monitoren og brukt til å skrive til analyse portene. Det har også blitt utviklet en sequencer som er funksjonell for forskjellige protokoller i rammeverket. Alle komponentene i hierarkiet har blitt parametrisert for å få spesialisert funksjonalitet i alle protokollene. Agenten, som er parametrisert med et konfigurasjonsobjekt og sequence item, bygger testbenken ved hjelp av konfigurasjon som er hentet fra konfigurasjonsdatabasen ved å bruke håndtaket for de forskjellige protokollenes agenter for å velge riktig vei.

Antall kodelinjer har blitt brukt for å kvantifisere noe av effektiviteten av UVCen, siden mindre kodelinjer vil mest sannsynlig resultere i mindre bugs og derfor mindre tid brukt på å finne de. Det var totalt 28% mindre kodelinjer for protokollene når det ble sammenlignet med det gamle rammeverket. 21 filer i rammeverket har også blitt erstattet med de generiske komponentene utviklet i dette prosjektet.

UVCen som har blitt utviklet kan bli gjenbrukt i flere rammeverk ved å bruke parameteriseringen. Rammeverket som bruker UVCen må sette konfigurasjonen i konfigurasjonsdatabasen i tråd med det som har blitt gjort i denne avhandlingen, for korrekt funksjonalitet. Denne UVCen kan også redusere tiden brukt for å debugge og lage testbenker, og tiden spart kan brukes til å nå deadlines eller for å forbedre kvaliteten på testene.

Acknowledgements

I would express my gratitude to my supervisors: Vitaly Marchuk from Microchip Technologies Norway AS and Professor Kjetil Svarstad from NTNU for their expert advice, guidance, support and encouragement throughout this project.

Contents

List of Figures

List of Tables

Abbreviations

BFM	**B**us **F**unctional **M**odel
DUT	**D**evice **U**under **T**est
HDL	**H**ardware **D**escription **L**anguage
HVL	**H**igh-level **V**erification **L**anguage
ISS	**I**nstruction **S**et **S**imulator
NVM	**N**on **V**olatile **Memory**
RTL	**R**egister **T**ransfer **L**evel
TLM	**T**ransaction-**l**evel **m**odeling
UVC	**U**niversal **V**erification **C**omponent
UVM	**U**niversal **V**erification **M**ethodology

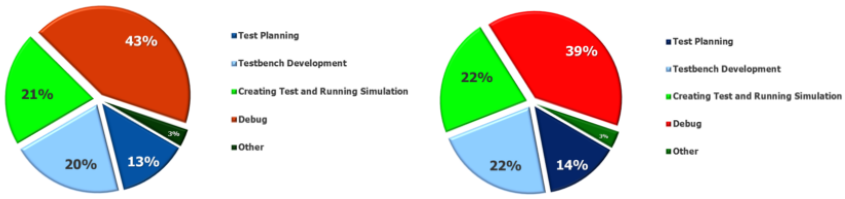
Chapter 1

Introduction

1.1 Motivation

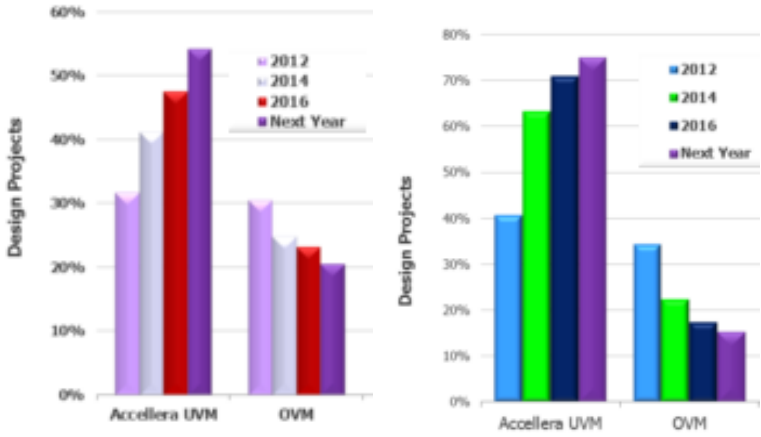
According to "The 2016 Wilson Research Group Functional Verification Study"[1] presents that the average time spent in verification in FPGA projects is 48%, and 55% in ASIC/IC projects. Figure 1.1 shows where the time is spent by Verification engineers during a project. For both projects for FPGAs and ASICs/ICs the time spent developing the testbench is around 20%, and the time used debugging can be over 40%. Maximal reuse needs to be utilised in order to reduce the development time for the testbench [5]. Reusing the same components can also reduce the time spent debugging. To reduce the time used running tests, iterations and debugging, a better, more advanced approach is needed. Figure 1.2 presents the different methodologies trends in verification used for FPGA and ASIC/IC. It shows that the adoption of UVM has drastically increased during the past years. UVM can help increase the maximum reuse.

As reuse in UVM comes more in focus, the components used in UVM should be improved. By developing a parameterised Universal Verification Component(UVC) could be a step in the right direction towards maximising reuse and decreasing the time spent developing the testbench, decreasing the total time spent in verification, and with it, increase the probability of delivering a project on deadline. The extra time granted could also be spent on quality improvement.



(a) FPGA (b) ASIC/IC

FIGURE 1.1: Where Verification Engineers spend their time[1]



(a) FPGA (b) ASIC/IC

FIGURE 1.2: FGPA Methodologies and Testbench Base-Class Libraries[1]

1.2 Contributions

Several techniques and methods were used in this project in the process of developing a UVC. The following has been achieved in this thesis:

- Modified an framework to develop a UVC.
- Developed an agent that only need configuration object and sequence item as parameters.
- Sequence item has been improved to function for several protocols.
- Driver, monitor and sequencer have been made generic by moving and encapsulating specialised functionality on the interfaces.

- Modified the configuration for the framework to be compatible with the UVC.

1.3 Methodology

In this project SystemVerilog and UVM version 1.1d have been used, with Cadence Incisive as the simulator. An existing framework for verifying a CPU has been used as a base in order to develop a unified agent for working with various busses, and the framework has been modified for it to work. During the project, the workflow has consisted of altering the framework while being true to UVM. When an alteration has been made, a script was used to execute a test for the framework. This script builds the framework, simulates and does a clean-up. It creates logs for the build and for the simulation, which was used to verify correct functionality, or debugging. SimVision has also been utilised when inspecting the waveforms for correct behaviour.

1.4 Thesis Overview

The successive chapters will go through how the process of the development of a Universal Verification Component, and re-working the framework to work with it. An overview over the chapters included in this thesis is given below:

Chapter 2 provides some background information about the UVM methodology. This includes information and terms that could come in handy throughout the thesis. First it will roughly go through a quick summary of UVM, its creation and the components that is used to build the testbench. It will then delve into how information is used and shared in the testbench. It will further explore how the different components can be configured, with the use of parameters and other methods.

Chapter 3 presents how the framework that is used as foundation in the development of the UVC is configured, and a little bit about the structure and components in it. This chapter also go in on how it could be possible to improve the existing framework.

Chapter 4 covers how the process of developing the UVC is carried out. It starts with explaining the need for a different starting point, before it advances to the description of the work process. It then progress to describe how the functionality is moved about to create the UVC, and how the framework and configuration is adjusted to match the UVC.

Chapter 5 presents the benefits of code reduction, both in the amount of code lines and number of files, between the the a specialised framework and the framework that use the UVC.

Chapter 6 is the discussion. The discussion will be based on the entire thesis, and discuss the ways of the results found, advantages and disadvantages using the UVC, parameterisation and configuration used in the solution, some inconveniences that occurred during the project, and at last it will discuss some alternative methods that could be used to solve the problem.

Chapter 2

Background

The purpose of this chapter is to introduce important concepts and terms that are used in the modelling approach and discussions in this thesis. In the search for information about generic UVCs, it was revealed that there is very little, to no information regarding the development of a pure generic UVC.

2.1 Universal Verification Methodology

Universal Verification Methodology (UVM) is a methodology for functional verification of design units, created by Accellera. UVM is based on Open Verification Methodology (OVM) that is created by Cadence and Mentor, and VMM from Synopsis. The UVM standard is built on the principle of cooperation between EDA vendors and customers. The building blocks from the class library makes it possible to quickly develop well-constructed and reusable verification components and test environments[6]. The UVM package contains a class library that consists of three main types of classes[3]:

- *uvm_components*, used to construct a class based hierarchical testbench structure
- *uvm_objects*, used as data structures for configuration of the testbench
- *uvm_transactions*, used in stimulus generation and analysis

UVM uses a hierarchy built on classes in its testbenches. These classes are derived from the *uvm_component* base class. The hierarchy is based on a class relationship, Figure 2.1 shows the hierarchical layers that are commonly used in UVM. The top level class in a UVM testbench, often called "test", is responsible for configuring the testbench, initiating the construction process by building the next level down in the hierarchy, and by initiating the stimulus by staring the main sequence. The environment and the agent are components used in order to enable reuse.

The Design/device Under Test (DUT) is the code/device that is to be verified by the testbench. It is often connected to the testbench through interfaces. It receives stimulus from the testbench, and returns the output of that stimulus.

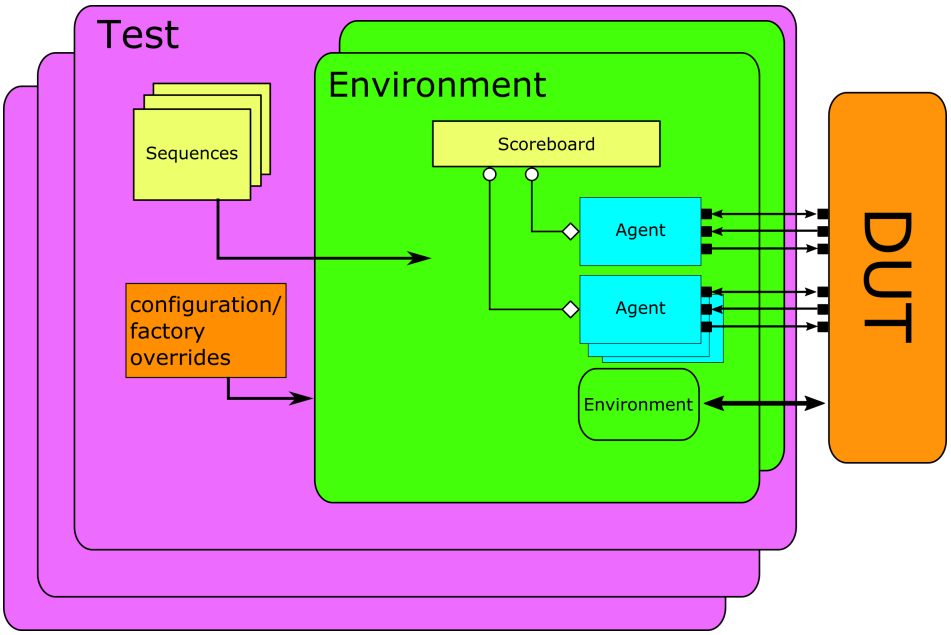


FIGURE 2.1: Block Level UVM Test Bench - Hierarchical Layers

2.1.1 Horizontal and Vertical Reuse

In order to save time and resources, and increase productivity, UVM promotes reuse. Reuse in UVM can be divided into two sections, horizontal and vertical, which describes the context of what verification artefacts are reused. Horizontal verification means that a component can be used in another system or project,

"block to design", but still at approximately the same level of abstraction [7]. Vertical reuse is used when a component is used in another hierarchical abstraction level, "block to subsystem to full chip". Vertical reuse is often used when reusing sequences, and testbenches in the same system. It is often achieved by encapsulation and extensibility. Utilising vertical reuse may increase verification efficiency by writing less code, which also gives less code to maintain, and less time on initial debug. It also gives parallel development of testbenches and a architecture that can be integrated with little effort into other modules and subsystems.

2.1.2 Test

The IEEE Standards Dictionary: Glossary of Terms & Definitions[8] defines test as: specific customisation of an environment to exercise required functionality of the DUT.

The test is in the top of the hierarchy. Tests are derived from the *uvm_test* class. Typically the UVM test performs three main functions[9]:

- Initiate the top-level environment
- Configure the environment
  - With factory overrides or the configuration database
- Apply stimulus by invoking UVM sequences through the environment to the DUT

2.1.3 Environment

IEEE[8]: The container object that defines the testbench topology.

The UVM environment groups together verification components that are interrelated. The top-level environment usually contain all the components that are targeting the DUT. Components that are encapsulated by the environment are usually the agent, scoreboards and even other environments[9].

2.1.4 Scoreboard

IEEE[8]: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually refers to the entire dynamic response-checking structure.

The scoreboards main function is to check the correctness of the DUT by comparing the DUTs output with the expected values[9]. The scoreboard can be generalised into two parts: predict, which determines the correct functionality of the DUT, and evaluate which checks if the observed results of the DUT matches the predicted results. Separating the prediction task from the evaluation task gives the best scoreboard architecture [3]. This is because it gives the most flexibility for reuse by allowing for substitution of predictor and evaluation models.

2.1.5 Agent

The IEEE Standards Dictionary: Glossary of Terms & Definitions[8] defines an agent as an abstract container used to emulate and verify DUT devices; agents encapsulate a driver, sequencer, and monitor.

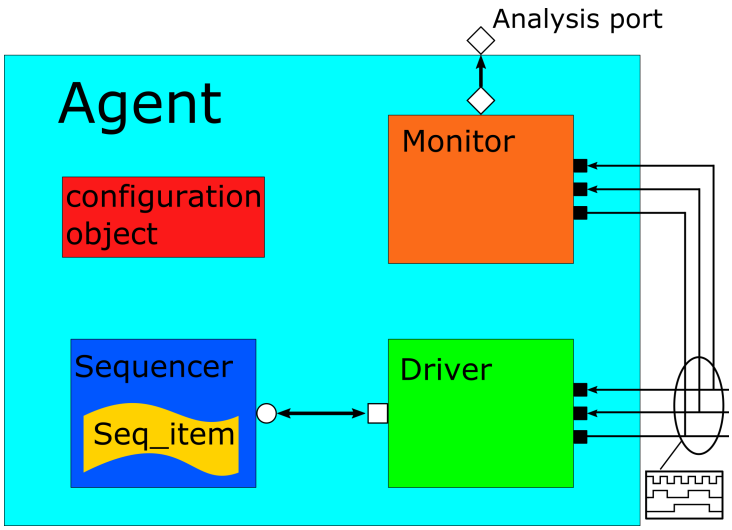


FIGURE 2.2: Active agent

In addition to a driver, sequencer and monitor there might also components related to coverage monitors or scoreboards. As seen in Figure 2.2 There is an analysis

port connected to the agents monitor. This makes it possible for any external analysis component to connect without interfering with the agent. The agent can also contain a configuration object, that can hold information like:

- Sub-components created under the agent
- Handle to the virtual interfaces
- Functional behaviour of the agent

The agent can be configured active or passive, more on how configuration is done is given in Chapter 2.1.19. An active agent generate the stimulus and drive the DUT, the driver, sequencer and monitor is therefore present. A passive agent, see Figure 2.3, does not drive the DUT signals, it only samples them, and thus, it does not contain a driver or a sequencer.

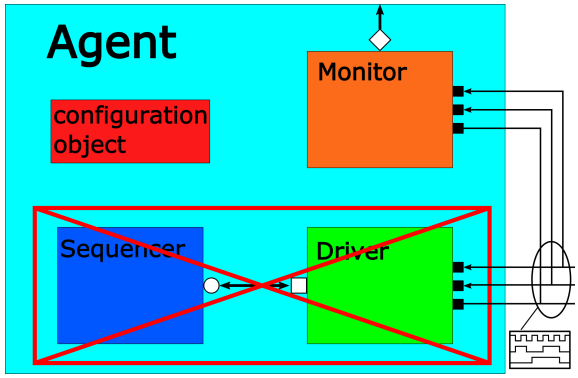


FIGURE 2.3: Passive agent

2.1.6 Driver

IEEE[8]: A component responsible for executing or otherwise processing transactions, usually interacting with the DUT to do so.

The driver receives individual sequence item transactions from the sequencer via a TLM port, and applies it to the DUT interface[9]. The driver therefore spans the abstraction levels by converting the transaction-level stimulus to pin-level stimulus.

2.1.7 Monitor

IEEE[8]: A passive entity that samples DUT signals, but does not drive them.

The monitor samples the DUT interface, capturing information about the transactions. The information can be sent to the rest of the testbench for further analysis[9]. The monitor spans the same level of abstraction as the driver, by converting pin-level activity to transactions. The monitor typically uses a TLM analysis port to broadcast the transactions created. Processing on transactions, like coverage collection, checking, recording etc, can be done by the monitor or delegated to other components via the analysis port.

2.1.8 Sequencer

IEEE[8]: An advanced stimulus generator which executes sequences that define the transactions provided to the driver for execution.

A sequencer serves as a router of transactions. The sequencer component feeds the driver with transactions with stimulus in the form of sequences and sequence items from a parent sequence[3].

2.1.9 Sequence Item

A sequence item is a class-based transaction representing stimulus passed from a sequence to a driver[3]. A sequence item is also known as a transaction. A transaction is a class object that represents a communication abstraction. It is a collection of information needed to model a unit with communication between two components. The information may include variables, constraints, and other fields and methods needed to generate and operate the transactions[9]. In order to model the communication to the level of abstraction wanted, transactions can be composed, decomposed, extended, layered and manipulated.

2.1.10 Interfaces

Interfaces are used to encapsulate communication, which facilitates reuse, between design blocks, and verification blocks, enabling a transition from the abstract level

used in the testbench to lower RTL(Register-transfer level) and structural levels of the design [10]. At low level an interface is an named bundle of nets and variables, and can be accessed through a port. At higher levels the interface can encapsulate functionality as well as connectivity. An advantage using interfaces is its flexibility, as it can be parameterised the same way as a module, and can contain parameters, constants, variables, functions and tasks, and continuous assignments[10]. Modports is commonly used in interfaces. The modport controls the use of tasks and functions, and provides information about the direction for the module interface for certain modules. Implementing tasks in the interface allows for a more abstract level of modelling, as the master module can call the tasks with no need for references to the wires.

2.1.11 Transaction Level Modeling

Transaction level modeling(TLM) is a modelling style for building highly abstract models of components and systems[9]. An advantage of this abstraction is that simulation speed can increase, observing the traffic easier and debugging is easier with TLM than RTL[11]. TLM communication creates the possibility to communicate between components utilising functions. Where tasks and functions in one component are available in another component. UVM provides a set of interfaces and channels for use with transaction-level communication. The use of TLM interfaces promotes reuse, as it isolates the components from the interface. This means that the component can be swapped for another, given they have the same interface.

```
seq_item_port.get_next_item(req);
    ... //drive the response onto the interface
seq_item_port.item_done();
```

The code snippet above shows how the driver can get a transaction from the sequencer during the run-phase. The *get_next_item* is a blocking function, and it will block until there is an item that is available from the sequencer[12]. Opposed to using the *get*-function, once *get_next_item* has been called, the *item_done* function has to be called in order to indicate that the request is complete to the sequencer.

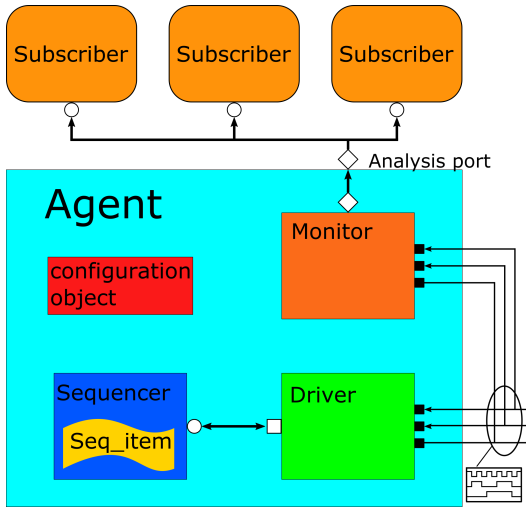


FIGURE 2.4: Agent with analysis port and subscribers

The analysis port can be used as a broadcast port[3]. The analysis port does not care if it is not connected, or connected to one or more ports. Figure 2.4 shows how the analysis port connects from the monitor, to the agents analysis port, before it is connected to several subscribers analysis export. The subscriber component is usually programmed to subscribe from analysis ports, it could for example be a checking or coverage collector. In the figure the monitor can use the push transaction function *port.write(tx)* in order to write to the analysis port, the subscriber should then have a *write*-function that is read-only. The analysis port is not supported with the factory, and needs to be created using the raw constructor "analysis_port = new("analysis_port", this);"

2.1.12 Transaction Methods

It is common to perform operations on transactions, like copying the transaction, printing or comparing. UVM provides a standard set of methods to do this. The user transaction class is extended from the **uvm_sequence_item** class, which comes from the **uvm_object** class type, and thus inherits utility methods: *copy, compare, print, pack, unpack, record* and more[13]. These methods are non-virtual and should not be overridden. If there are custom transactions that needs to be implemented, the user should override the do_* methods, which are called by these utility methods.

Do_print is called by *print* and *sprint*, and allows the user to decide what is printed. To ensure correct output format of the *print* and *sprint* operation, the printer should be used by all the *do_print* implementations. Which means that instead of using *$display*, the *do_print* should be called through the printer's API[14].

The *do_copy* is called by the *copy* method. It is an empty method, and should therefore be user-defined if the field-macros are not used. The *do_copy* method should make a deep copy of a data object[3]. A deep copy is when the value of the individual properties in a data object are copied to another, in contrast to a shallow copy where the pointer is copied. The implementation of the *do_copy* should call *super.do_copy*, and *$cast* the **rhs** argument to the derived type before the copy[14].

To support the viewing of data objects as transactions in a waveform the *do_record* can be used. As the other *do_methods* it is called by the *record* function. This method is also empty, and the implementation should call the appropriate recorder methods.

There are however other methods than the *do_methods* that is important. The *create* method, which should be used if the *'uvm_object_utils* macro is not utilised, the *clone* method, which calls the *create* method before calling the *copy* method, and the *convert2string* method[14]. The *convert2string* method is a virtual function and works as a user-definable hook that allows users to provide object information as a string. The default *convert2string* method could be seen as a placeholder that returns an empty string [13]. It is recommended to override the *convert2string* method in every transaction class with a proper formatted string of the transaction variables. When extending a transaction class from a base transaction class it is suggested to extend the *convert2string* method calling *super.convert2string*, and not reformat all the transaction variables.

2.1.13 Polymorphism

Polymorphism is one of the key-concepts in object-oriented programming, and it allows for the substitution of objects that have identical interfaces for each other at run-time[2]. The use of a variable of a superclass type, is with polymorphism, allowed to hold subclass objects and to directly reference the methods of those subclasses[10]. Polymorphism can be partioned in two: dynamic and

static. Static polymorphism is also known as method overloading, as it use a method many times, but with different parameters, and decided at compile-time. Dynamic polymorphism happens at run-time and is also known as method overriding. It is where a method is overridden with the same signature in different classes.

2.1.14 Factory

An object for creating other objects in object oriented programming is called a "factory". It is in other words a function or method that returns objects of a varying prototype or class from a method call[2]. When it comes to class-based programming, the factory works like an abstraction of the constructor class. It is possible to define an abstract factory class to declare an interface for how objects are created throughout an application. Figure 2.5 presents how an abstract factory can be utilised. The client uses the interfaces of the abstract product and abstract factory to create a product with the right specifications. The concrete factory implements the operations to create the objects, and the abstract product creates an interface for a type of product object.

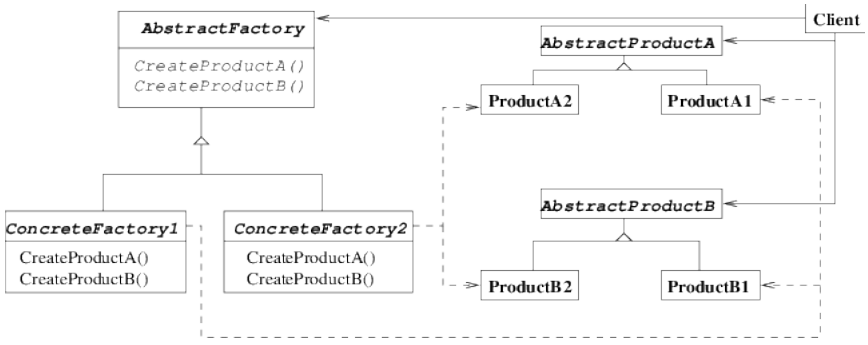


FIGURE 2.5: Abstract Factory pattern[2]

To allow components to manufacture objects without needing to specify the exact class, UVM provides a factory. This means that it is possible to allow for an object to be substituted with an another, without having to edit the structure of the testbench, or the testbench code. The factory takes advantage of Object oriented programmings(OOP) polymorphism, see Chapter 2.1.13. The mechanism used by the factory is called an override, and it can be by instance or type [9]. The factory

can be seen as a special look-up table that create requested components or transaction types. The factory should be used instead of the *new()* class constructor[15]. The *new()* constructor will only create a transaction or component of the specified type. It fixes the type during construction, meaning no run-time changes are possible. It is therefore not possible to employ dynamic polymorphism. If there are changes to the components there will be need for change in the source code.

Using the factory to create the constructor grants several advantages, as it enables the choice of object type to be overridden from the test[16]. When using the factory, the components should be registered using the macros *'uvm_component_utils* or *'uvm_object_utils*. The ***::type_id::create()*** command creates an object of **type_id** stored in the factory, and is needed in order to use overrides. By using the override command, tests can make **type_id** substitutions. During the build phase the components are creating a tree-like hierarchy when creating the testbench structure. This process happens top to bottom, where every component names and builds its children, and passes a pointer, using **.this** pointer, from itself to the child[15]. If the components and transaction types do not match the structure of the testbench during the build-phase, the top-level test can change its type, and then the factory uses overrides on the rest of the testbench.

2.1.15 Generic Programming

The Merriam-Webster's dictionary[17] defines "generic" as "relating to or characteristic of a whole group or class". It can also be characterised as "not specific". When it comes to generic programming, the intent is to facilitate reuse, by writing code once and invoking or instantiating it several times with generic parameters[18]. The ability to write code for an algorithm that is independent of parameters that will be specified later, can be considered the concept of generic programming. SystemVerilog, as opposed to Verilog which only allows parameterisation of certain values, adds the possibility of full parameterisation of all data types[10]. The parameterised data types are implemented through the use of type definitions in parametesised classes. To avoid writing similar code for different size and data types, it is useful to define a generic class.

2.1.16 Parameterisation

Parameters increase the flexibility, but it also increases the complexity of the code[19]. In SystemVerilog it is possible to make classes with their own set of parameters that may be overridden when declaring a class variable, these classes may be called generic classes. These classes are not true data types before the parameters have been declared. When the parameters has been declared the classes can be called for a specialised class [10]. A parameterised class can be declared as following:

```
class example #(int size = 8);
    bit [size-1:0];
endclass
```

Here the size integer is default 8, but can be given another value when creating the class. If there is no default value given, it has to be set before the class can be used. A parameterized class can also be extended by another parameterised class. Underneath is a few examples of the headers of extending parameterised classes.

```
class C  #(type T = bit); ... endclass // base class
class D1 #(type P = real) extends C; // T is bit (the default)
class D2 #(type P = real) extends C #(integer); // T is integer
class D3 #(type P = real) extends C #(P); // T is P
class D4 #(type P = C#(real)) extends P; // for default T is real
```

Here class **D1** extends class **C**, and uses the default type from the inherited class. Class **D2** extends class **C**, but the default type in class **C** is integer instead. **P** is a parameterised class, and by extending **C** which is using **P**, the extended class, **D3**, will also be parameterised. Class **D4** uses the type parameter **P** as a base class, and the name must therefore resolve to a class type after elaboration[10].

It is possible to group the parameterisation classes into two hierarchies: containment hierarchy and inheritance hierarchy [18]. The containment hierarchy works as a "has-a" relationship of object oriented programming (OOP), where a class definition can instantiate other class variables. It is then possible for the class

definition to use its parameters to parameterise the contained class members variables. This is typically done in verification environments, as they group the class components. For example the uvm test, agent, driver and monitor are each extended from **uvm_component**. The driver and monitor are included in agent, and there can be several instantiations of agents in the test class. Given there are unique class definitions used for instantiating the classes it is therefore possible to get several specialisations of agents, drivers and monitors when declaring the test.

2.1.17 Inheritance

The inheritance hierarchy is the "is-a" relationship of OOP. This is where the properties and methods of a extended class is inherited directly from the base class, and makes it possible to reference them from the extended class[18]. Parameters may then be propagated from the extended class up to the base class. For each instance of an extended class that is instantiated, the base class will also be instantiated.

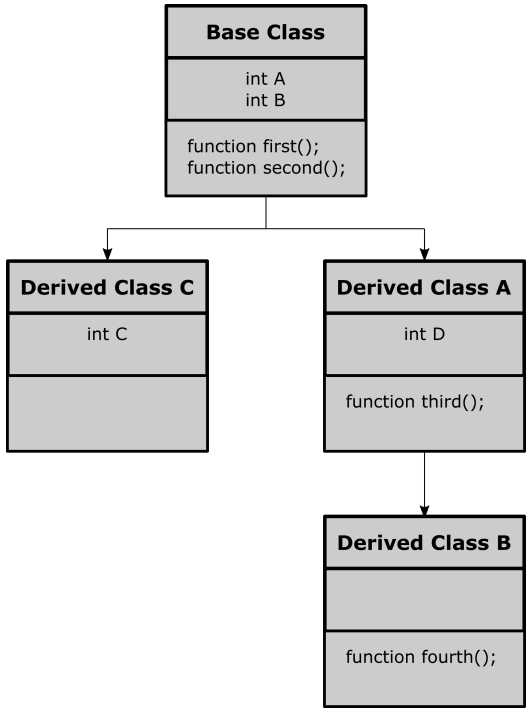


FIGURE 2.6: Inheritance

Figure 2.6 shows how a classes can be derived from a base class. The three classes are A, B and C are all extended from the base class. Table 2.1 presents the variables and functions each of the classes will have. As class C is directly inherited from the base class, it will not have any of the variables or functions that is added in class A or B. Class B is the only class with the function *fourth()*, which means that the other classes can not use the function.

Class	**Variables**	**Functions**
Base	int A, B	first, second
Derived A	A, B, D	first, second, third
Derived B	A, B, D	first, second, third, fourth
Derived C	A, B, C	first, second

TABLE 2.1: Overview over inherited variables and functions

In order to create static class property for a generic class, it has to become a concrete specialisation[18]. A specialised class can be created with the use of *typedef*. The use of parameterised classes can be good for horizontal reuse, as it makes it easier to use in different projects. It could also be a good fit for the base classes in UVM. In order to register parameterised classes with the factory, the *'uvm_component_param_utils* and *'uvm_object_param_utils* macros must be used[9].

2.1.18 Configuration Objects

Configuration objects are used to keep a high reusability and easily configurable testbench. A configuration object is a class that contains items necessary to configure a single target [20]. The configuration object can hold information about how the component that contains the object shall create sub-components, pass handles, and the behaviour for the component. Randomisation can be a useful feature when using configuration objects, and can be done by running the *.randomize()* function. It is also possible to set constraints to variables if it is wanted to restrict the variable. Extending the base class object, and setting new constraints can help create a new tests, without having to rewrite code, helping with the reusability. In addition is it possible for several components to have the same reference to a configuration object by passing around the handle to the object[21]. All the changes done to the object's contents will be visible to all the components with its reference. Utilising configuration objects also reduces the risk of applying

the configuration to the wrong target, as classes are strongly typed. It is also possible to see if a configuration has been provided by looking at the object handle value, if it is of null unambiguously, it is not provided.

2.1.19 Configuration

In order to create a verification framework that can be easily reused, a key concept is that it should be as configurable as possible, without having having to rewrite every part of the testbench. Delivering information to various parts of the testbench from a central location, and with variables and parameters, reduces code rewriting. Variables are set at runtime, and can be organised into "configuration objects", and accessed through a resource/config database[3]. Resource databases are an in-memory database that contains objects of different types[20]. The parameters must be set during compilation, as these can not be changed during runtime. In order to be reusable, testbench elements, like components, sequences, sequence items etc., needs knowledge about its environment, and how it should operate in it. This information can be classified as configuration information. Having knowledge of the configuration information, the testbench elements can alter its topology or behaviour accordingly [22]. The most common way configuration information is passed to the elements are:

- class parameters
- constructor arguments
- function calls

For more on class parameters see Chapter 2.1.16. In UVM there are two types of databases that are closely related, the configuration database, *uvm_config_db*, and the resource database, *uvm_resource_db*. The configuration database and the resource database are closely connected, as *uvm_config_db* is an extension of *uvm_resource_db*. The *uvm_resource_db* works as a look-up table, and the hierarchy is not important[23]. The *uvm_config_db* is a type-specific configuration mechanism that offers a facility for specifying hierarchical configuration values of desired parameters[9]. The *uvm_config_db* is often used when sharing virtual interfaces. In order for the testbench to monitor or drive the DUT, it needs access to the interfaces. The instantiation of the interfaces happens at the top of the framework,

where it checks for them, and then sets an handle to the virtual interfaces using the configuration database. The class header for the resource database and the configuration database is given below

```
class uvm_resource_db#(type T=uvm_object)
class uvm_config_db#(type T=int) extends uvm_resource_db#(T)
```

As seen, the resource database class uses a parameterisation where the default type **T** is an **uvm_object**, while the configuration database standard is an **int**. It is possible to set or get configuration information by using *set-* or *get-* functions of the uvm_config_db or uvm_resource_db classes[23]. These functions are static and must therefore be called using the "::" operator. The syntax for the *set-* and *get-* function for **uvm_config_db** is:

```
uvm_config_db#(<type>)::set(uvm_component cntxt, string inst_name,
    string field_name, <type> value)

uvm_config_db#(<type>)::get(uvm_component cntxt, string inst_name,
    string field_name, ref value)
```

Where **cntxt** is the hierarchical starting point of where the database entry is accessible, the object hierarchy whereas. **inst_name** is the path that limits the accessibility of the database entry. **field_name** is the name of the object, the lookup label. Objects using *set()* or *get()* must use the same label, or else the receiving part will fail to find the object in the database[24]. **value** is the object handle shared in the database, the value that is to be stored in the database, it is of a parameterised type **type**, which is set to **int** by default.

The *set*-function uses **cntxt** and **inst_name** in order to specify the "address" where the object handle is stored to control the recipient of the object [24]. The *get*-function can select from where it wants to retrieve information, it can for example be from elsewhere in the hierarchy. It is common to use **this** pointer in **cntxt** in order to specify the current scope for both the *set-* and *get*-function. For **inst_name**, the *get*-function often use an empty (""), since it usually gets

objects destined for itself. The *set* function uses it to address the object to the appropriate sub-block.

With the use of **cntxt**, **inst_name** and **field_name** it is possible to make a number of different paths to the same object. When referencing down the hierarchy it is common to use the **this** pointer. When referencing upwards, the **null** pointer or the *uvm_root::get()* function can utilised to access the hierarchy root, and then the path down the hierarchy is given by by **inst_name**. The database combines the **cntxt**, **inst_name** and **field_name** parameters to make a key used for searching through the database. There are three metacharacters, "*****, + ,**?**" , that can be used when creating the path. "*****" means 0 or more characters, "+" means 1 or more, and "**?**" means exactly 1 character. "*****" is usually called a wildcard

In addition to depositing resource items destined for a particular UVM object or component instance into a database before the instance has been constructed, the lookup mechanism supports keys containing wildcard and regular expression patterns[21]. This mechanism can match only a part of a object's full instance name, which supports two use cases:

- Single resource targets multiple objects, which all receives the same resource
- A resource can target an object without knowledge about the full pathname or if the object has been created.
  - Helps with vertical reuse since it can appear in different levels of the hierarchy.
  - Allows configuration objects to be constructed by a top level module and give information to components before they are constructed.

Every time the *uvm_config_db*'s *set-* or *get-* method is used, there is an entire scan of the configuration database in order to match strings. Minimising the usage of accesses to the configuration database, avoid wildcard matching, and avoid the *wait_modified()* method is recommended, since it will take longer to find a match if the database is large and the search is little specific[4]. The *wait_modified()* function is blocking, and is only triggered if the set function is called using the same exact **cntxt**, **inst_name** and **field_name** [20].

2.2 Comprehensive UVM

There are several ways to utilise UVM and its components. A UVC could ease the implementation of future frameworks, as it contain components needed in the framework. When implementing a UVC synchronisation must also be considered. It also is possible to create UVCs that are designed for multi-purposes and are acceleration ready, generic and configurable[25].

2.2.1 Universal Verification Component

A UVC is a verification component for use in UVM. It is a multi-faceted definition and has different meanings in different contexts. The topologies that can be defined to be a UVC can be boiled down to[3]:

- Protocol UVC
  - Each verification component connects to a DUT interface and communicates with a single protocol
- Fabric/compound UVC
  - A verification component that contains a configurable number of instances of the protocol UVC configured and hooked up coherently as a unit. Purpose to verify a structured fabric with multiple interfaces of the same protocol
- Layered UVC
  - Provides a higher level of abstraction than basic pin protocol, with two common ways of construction
    - A UVC which does not connect to pins, but provides an upper layer of abstraction to an existing protocol UVC
    - A UVC which wraps and extends a lower-level protocol UVC

A UVC should make use of the factory registration API and macros provided with UVM so that its components can participate in factory creation[3]. This way the user decide on what portions should be factory substituted. Simple protocol UVCs are normally not environments, but agents that are instantiated individually for each interface. A protocol UVC should also normally not contain scoreboards.

2.2.2 Dual Top for Accelerated Verification

As design size and complexity of automated testbenches keeps on growing, the need of hardware assisted acceleration has increased as well. To achieve this a methodology based on co-emulation is recommended, which also promotes reuse[26]. Co-emulation modelling describes the process of modelling and simulating a mixture of software models represented with an un-timed level of abstraction, simultaneously executing and inter-communicating through an abstraction bridge, with hardware models represented with the RTL level of abstraction, and running on an emulator or a simulator[27]. It is in other words a RTL DUT running in a hardware emulator that can interact with a testbench running on a workstation. The testbench can still be used in simulation after it has been created to be emulation ready. There is a desire for a high reusability in testbenches that can be used in emulation, and are therefore often designed thereafter.

Typically the DUT-testbench setup use a single SystemVerilog module as top level. The top level module contains the DUT and the interfaces associated to it, protocol modules, connection and support logic[3]. Figure 2.7 shows the typical encapsulation of a DUT-TB setup, it is typically a container for both the testbench and the DUT. It also contains the the connections and support logic, like the clock generation.

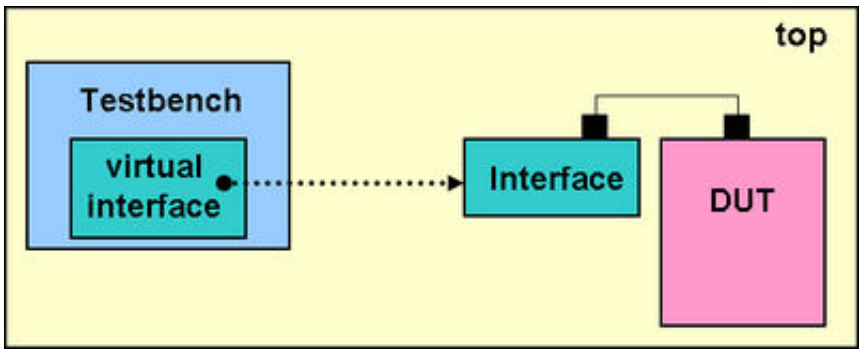


FIGURE 2.7: Single top[3]

To create a working co-emulation ready testbench, there are some requirements that needs to be fulfilled[26]. It needs to follow the principles of co-emulation. It is recommended to split the testbench into two top modules as seen in Figure 2.8. One that module to wrap the DUT, interface, protocol modules, clock generation

logic, DUT wires, registers etc. The other module is the module that creates the testbench. The two top modules are in their own domains: the timed/synthesisable HDL(Hardware Description Language) side where the DUT, BFM(Bus Functional Mmodel), and clock/reset generation is, and the untimed HVL(High-level Verification Language)/TB side, where the testbench generation and analysis code is. When two top modules is used, it is called "dual top". The HDL domain needs to be synthesisable and contain all clock synchronous code that is to be used in the DUT, the clock and reset generators, and bus cycle state machines for driving and sampling the DUT interface signals. The HVL domain, contradictory to the HDL domain, needs to be strictly un-timed. This means that there should not be any use of time explicit advances[26]. These advances are for example clock synchronisations, # delays, wait or other time statements, as these should only be used in the HDL domain.

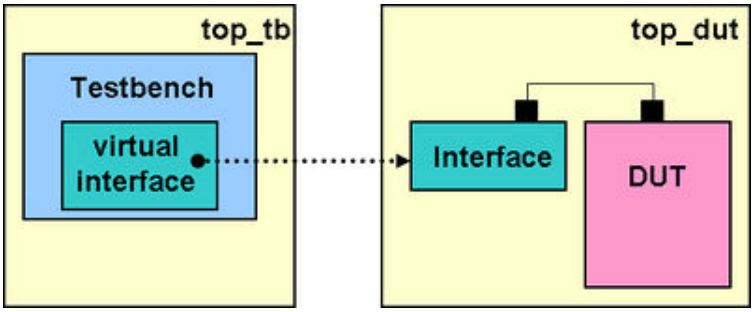


FIGURE 2.8: Dual Top[3]

When using accelerated transactors it is possible to shift the testbench load to the emulator. A transactor can be referred to as a Bus Functional Model(BFM). A transactor decomposes an un-timed transaction to a series of cycle-accurate clocked events, or, conversely, composes a series of clocked events into a single message[27]. In the hardware-assisted verification specific context, a transactor is a SystemVerilog interface or module on the HDL side[26]. The module in Figure 2.9 has a signal-level interface with the DUT and a transaction-level interface with the HVL side. With a partitioning the domain, performance can be maximised because the testbench and communication overhead is reduced, and intensive pin wiggling can be done in the dedicated timed domain, running at emulation speed[4].

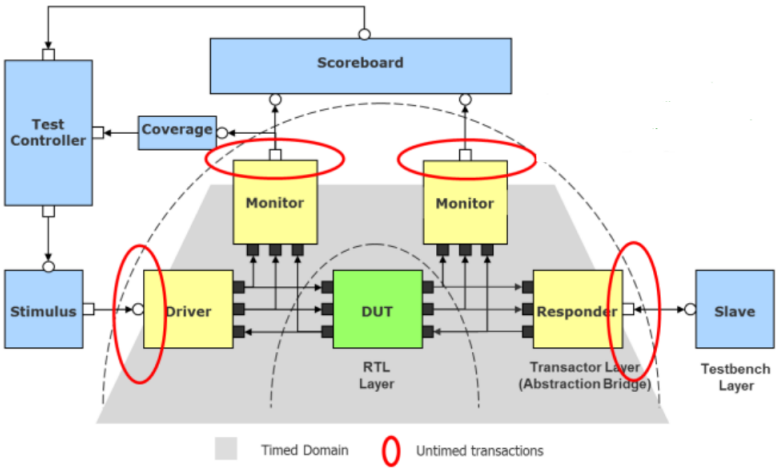


FIGURE 2.9: UVM layered testbench[4]

2.2.3 Synchronisation

A race condition is a flaw in a system or process that is characterised by an output that exhibits an unexpected dependence on the relative timing or ordering of events[28]. It is possible to split race conditions into two different types: hardware races and simulation induced races. The physical nature of combinational logic is typically the reason for hardware races. An example of a hardware race is when the inputs to a logic gate changes, this causes a finite delay before the output has changed.

Simulation induced races are undesirable consequences of the event-driven simulation algorithm used[28]. It is not intrinsic to the design or its physics, and it is an desire to avoid them. The simulator process event one at a time and unavoidably serialise the events that occur in the same time slot. The concurrent hardware activity is therefore modelled as a set of ordered actions by the simulator. These races can cause the simulator to cause a faulty design or to simulate a correct design, which is actually incorrect.

It is important that the testbench avoids race conditions with the DUT. In UVM it is therefore expected that the driver synchronises with a clock signal that is available in the interface to to maintain correct transfer of values from the driver to the interface variables[29]. This synchronisation grants important advantages,

like simplified driver code and ease of extending the driver without having to worry about duplicate synchronisation timing. The use of a clocking block in the interface will help with race conditions, but may be incompatible with emulation.

Modports connections and interfaces, mentioned in Chapter 2.1.10, can specify signals and nets, and communicate with the DUT. However, they do not denote any explicit timing disciplines, synchronisation requirements, or clocking paradigms. Clocking blocks can identify and capture the timing and synchronisation requirements of the blocks being modelled. The clocking block assembles signals that are synchronous to a particular clock and makes their timing explicit[10]. It also enables the user to write at a higher level of abstraction, as tests can be written with the use of cycles and transactions, instead of signals and transitions in time. Using clocking block constructs will facilitate the specification of assertion, cycle operations, and synchronous interfaces[29]. It also grants race free communication between the interface and DUT, as it enforces a time when signals are sampled.

To give a powerful and efficient means of describing the communication between, and synchronisation of concurrently active processes it is possible to use **event** objects[10]. The **event** data type provide a handle to a synchronisation object, which can be explicitly triggered and waited for. An identifier declared as an **event** data type is called a "named event". The "$->$" operator will unblock all processes waiting on the triggered named event. It is only the effect of a named event that is observable, when the it is triggered, and not the trigger itself.

Chapter 3

Verification Infrastructure

There exist an incomplete UVM framework for verification of a CPU. This framework has been created in accordance with earlier UVM methodology, and will be used as a foundation in the development of a Universal Verification Component. In this chapter there will be an introduction to how the framework is set up, by presenting how the configuration is done, and clarify the structure and components in the hierarchy. This chapter will also elaborate on how an improvement to the framework can be done.

3.1 Existing CPU UVM Framework

The existing framework is created for the verification of a CPU, and contain several protocols for interacting with it. The framework is developed in accordance with earlier UVM methodologies, with some with some unusual features. What is unusual in this framework compared to others, is that it can use two CPUs, one with and one without cache, in the verification process. It is therefore possible for the sequencer to feed the two CPUs and ISS (Instruction Set Simulator) with random instructions, which is stored in a non-volatile memory(NVM), and be compared with each other.

3.1.1 Protocols

There are several protocols for communication between the testbench and the DUT. The CPU UVM framework contain eight protocols. These protocols include an implementation for an agent, driver, monitor, configuration, interface, sequence-item, sequencer, and sequences for that protocol. The components are supposed to be implemented such that they can be used to verify the CPU with and without cache on the CPU. Some of the protocols' drivers accesses the NVM to retrieve data or instructions, if there is nothing in the memory, the driver will request a transaction from the sequencer. There are also protocols for other functionality, like on chip debug, interrupt requests and to halt the CPU externally.

3.1.2 Top Level

The framework's build process is ordinary according to UVM, where it is the test that initialises and starts the simulation of the testbench. Looking at the folder structure, the existing framework includes:

- Test base
- Extended test for active agents
- The environment
- An environment configuration object
- Scoreboards
- Agents for the protocols
- A virtual sequencer
- A set of virtual sequences
- Instruction set Simulator (ISS)

Where the test base is extended from the uvm_test class in UVM, and is used as the base to build the tests that verify the CPU. During the build phase in the test base, the environment is created and there is a check whether the interfaces for the protocols are present in the environment configuration, then there are put handles to the virtual interfaces, before they are set the configuration database. The test base then gets ready to run and report test content that is specified by extended tests. The extended test for active agents utilise the configuration database to

retrieve the environment configuration, and set the **is_active** convenience value to "UVM_ACTIVE" for the protocols configuration objects.

3.1.3 Environment

The environment creates a virtual sequencer and the agents for the protocols in the build phase. It also checks the configuration database if it should create the scoreboards. It creates all the components using the factory method described in Chapter 2.1.14. The environment proceeds to connect the virtual sequencer to the sequencers the agents created, and connect the analysis ports from the Pbus, Dbus, Irq, Iobus, Iobus_qr- monitors to the ISS through the scoreboard if the scoreboard enabled.

The code below shows how the agents, virtual sequencer and scoreboards are declared in the environment in the testbench, as well as how the agents are instantiated. None of the components need to specify any parameters in order to be used. There is also used a naming convention for the declaration of the different components.

```
class uenv extends uvm_env;
...
  iobus_agt          m_iobus_agt;
  iobus_qr_agt       m_iobus_qr_agt;
  pbus_agt           m_pbus_agt;
  dbus_agt           m_dbus_agt;
  ocd_agt            m_ocd_agt;
  irq_agt            m_irq_agt;
  cpu_externals_agt m_cpu_externals_agt;

  uvsequencer        m_vsequencer;

  cpu_snapshot_agt   m_cpu_snapshot_agt;
  cpu_scoreboard     m_cpu_sbrd;
  cpu_cpu_cache_scoreboard m_cpu_cpu_cache_sbrd;
        ...
  m_pbus_agt      = pbus_agt  ::type_id::create("m_pbus_agt"   , this);
  m_dbus_agt      = dbus_agt  ::type_id::create("m_dbus_agt"   , this);
```

3.1.4 Configuration

The framework configuration introduces an extra hierarchy for the verification components' configuration objects. Figure 3.1 shows how the different configuration objects are encapsulated by the environment's configuration object. The environment's configuration object also contain a bit to disable the ISS for debugging purposes, the first address for the program to jump and the first address for the interrupt service routine, and *do_print* function that prints the configuration of the objects.

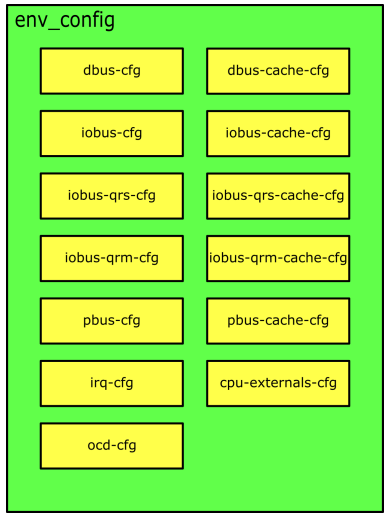


FIGURE 3.1: Environment's configuration object

In order to get or set values in the verification component's configuration object, this framework use the configuration database to retrieve the environment configuration before it access the verification components configuration objects' value hierarchically. An example of this can be seen under, where the Iobus' agent is set active.

```
if (!uvm_config_db#(uenv_config)::get(this, "*", "env_config", env_config))
    `uvm_fatal(get_name(), "Can't get env_config from uvm_config_db");
env_config.iobus_cfg.is_active=UVM_ACTIVE;
uvm_config_db#(uenv_config)::set(this, "*", "env_config", env_config);
```

As there are a lot of similarities in the different protocols, there is a template base class for the configuration that the configuration objects of the different protocols can inherit from. This template contain different variables and functions, like *convert2string* and *do_print*, that should be common, and that can makes it possible to build the verification component as intended. The configuration template specifies that every inherited class should implement a task that generates a wait for the slave and bus signals. It is also added an handle to the interface, which is specialised when the class is extended. It is the Pbus, Dbus, Iobus and Iobus_qr's configuration class that extends from the template, while the other protocols' configuration objects had not been ported to use the configuration object template at the time, containing some of the same variables and functions. The variables available in **config_template** used for configuration of the testbench are:

- is_active
- coverage
- agent_verbosity
- lock_sequencer
  - Set to prevent interruption from other sequences
- allow_data_generation
  - Allows generation of data in the driver
- allow_inst_generation
  - Allows generation of instructions in the driver
- allow_generate_swait_bwait
  - Slave and master wait

Figure 3.2 illustrates how the configuration information that is set in the environment's object from the configuration database, is passed to the sub-components' configuration object. The environment's configuration object is included in every agent, and the protocol specific configuration copied to the sub-components.

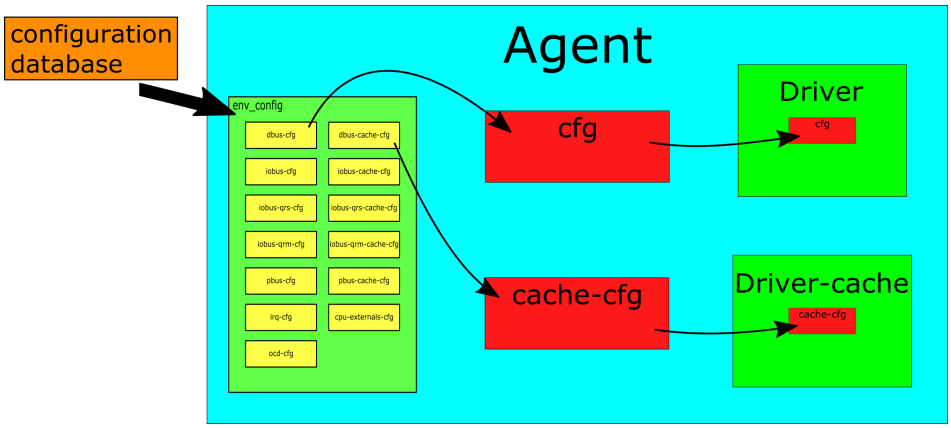


FIGURE 3.2: Configuration passing

3.1.5 Virtual Sequencer and Sequencers

Combining sequences can be used in order to create a hierarchy of stimuli, or to generate stimuli in parallel to multiple interfaces [30]. A sequence that controls stimulus generation using several sequencers is a virtual sequence [3]. The virtual sequences in the framework controls and times the sequences run on a single bus interface. This approach allows users of the framework to easily create various verification programs. The virtual sequencer holds handles to all the sequencers in the framework, as shown in the code below. In the virtual sequences the declared sequencer is the virtual sequencer, which can access to all the sequences.

```
class uvsequencer extends uvm_sequencer;
  `uvm_component_utils(uvsequencer)

  iobus_sqr         m_iobus_sqr;
  iobus_qr_sqr      m_iobus_qr_sqr;
  pbus_sqr          m_pbus_sqr;
  dbus_sqr          m_dbus_sqr;
  ocd_sqr           m_ocd_sqr;
  irq_sqr           m_irq_sqr;
  cpu_externals_sqr m_cpu_externals_sqr;
  ...
```

3.1.6 Agents

The agents' structure is similar to a normal agent, as described in Chapter 2.1.5, with the creation of a monitor, driver and sequencer. However, there are agents that create an additional driver depending on whether the cache-configuration declares it as an active agent. The handle to the interface is passed to the monitor and driver from the configuration object, as presented in Figure 3.3. The interface is set with the configuration database in the test base.

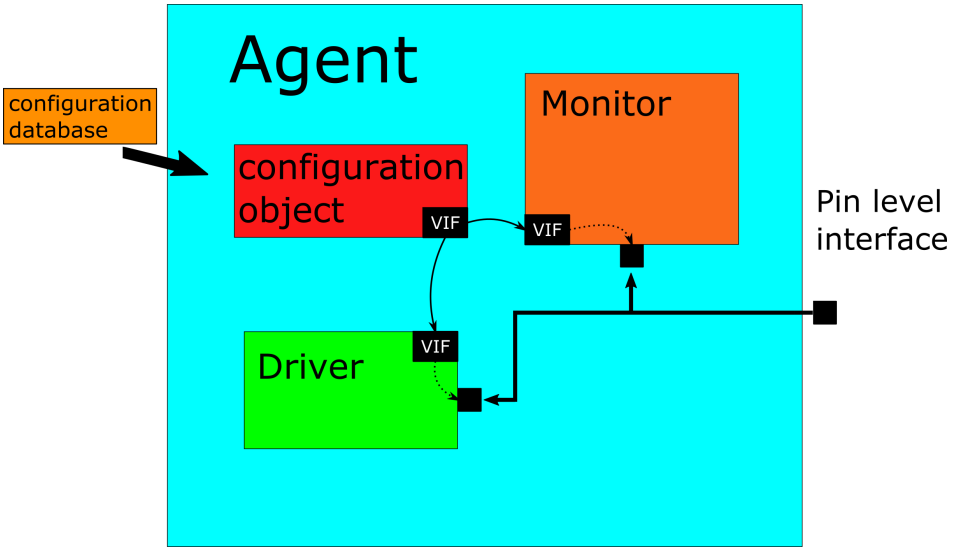


FIGURE 3.3: Virtual interface passing

3.1.7 Drivers

The drivers in the framework are constructed as standard drivers, as described in Chapter 2.1.6. Nevertheless, there are some differences in the driver between the different protocols. The drivers for the different protocols varies, as the busses are required to drive the busses according to their specifications, and the protocols' natural differences. Many of the drivers hold variables and constraints that are specific for the protocol, for driving the DUT correctly.

3.1.8 Monitors

As with the drivers, the monitors are also in accordance with the UVM methodology. Even though there are differences between the monitors of the different protocols, there is a pattern that occurs. Many of the monitors use hierarchical access through the virtual interface to access and check the signals from the DUT. The code snippet underneath is taken from the run-phase in the Dbus-monitor, and Figure 3.4 helps illustrate it.

```
class dbus_mon extends uvm_monitor;
    ...
    task run_phase(uvm_phase phase);
      dbus_item item;
      forever begin
        @vif.cb;
        if(!vif.cb.dbus_s.swait && !vif.cb.dbus_s.bwait
            && vif.cb.dbus_m.re && !vif.cb.reset) begin
          item  = dbus_item::type_id::create("item");
          item.enable_recording("DBUS ITEM");
          this.begin_tr(item, "DBUS ITEM");
          item.re     =vif.cb.dbus_m.re;
          item.we     =vif.cb.dbus_m.we;
          item.dbus_m_addr=vif.cb.dbus_m.addr;
          item.mdata  =vif.cb.dbus_m.mdata;
              @vif.cb.dbus_m.addr;
          item.sdata  =vif.cb.dbus_s.sdata;
          ap.write(item);
        end
      this.end_tr(item);
      end
endtask: run_phase
```

It is possible to see how it first creates a sequence item which is type-dependent to the protocol, checks for a specific combination of signals through the clocking block that resides in the interface, copies the signals over to the item, and at last writes this item to the analysis port. The *begin_tr* and *end_tr* which indicates the start and stop of the transaction, making it possible to be used in for example a waveform.

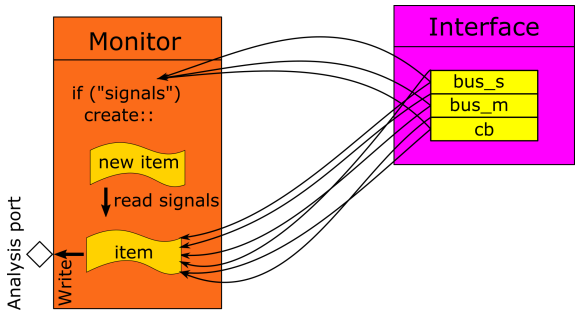


FIGURE 3.4: Monitor writes to analysis port

3.1.9 Sequencer and Sequences

The frameworks sequencers are extended from the **uvm_sequencer** and specialised with the sequence item for the corresponding protocol. For each sequence there is a sequencer declared for that specific sequence. All the sequencers in the framework has approximately the same functionality, there are however some sequencers that stands out from the rest. This is the sequencer for the Pbus, which has an access to the configuration database, and the Irq-sequencer, which builds two unlimited FIFO's.

3.1.10 Interface

The interface for each protocol is naturally different for each individual protocol. The interfaces use packed structs to access the DUTs signals. The interfaces also include a clocking block, and some include properties and assertions to assert correct functionality.

3.1.11 Sequence Item

Each of the protocols have their own sequence item. Where there are several similarities between the the protocols' sequence items. The ones with the most in common are: Pbus, Dbus, IObus and IObus_qr. These sequence items all have a variables for:

- a string name
- read data
- reset
- read enable
- bus-address
- preload bit
- done bit
- response bit

Theses sequence items, excluding Pbus', also have write data and a write enable bit. The read and write data size differs between one and two bytes for the different items. Most of the items have constraints that are specific for their variables. The Iobus and Iobus_qr have identical sequence items. The other protocols' sequence items differ as they are specified for individual use.

In addition to common variables each of the busses are using do_* methods, as described in Chapter 2.1.12. The reoccurring methods that are used is *do_record*, *do_print*, *do_copy* and *convert2string*, and the CPU_snapshot sequence item have a *do_compare* function.

3.1.12 Instruction Set Simulator

The ISS, instantiated in the scoreboard, get sequence items from all busses. It use these items to predict the CPU state after an instruction is executed, and sends snapshots via an analysis port to the scoreboard for comparison with the DUT snapshots. The ISS will in other words simulate the same instructions as the CPU, and compare the simulated results with what is generated by the DUT.

3.2 Improving the Existing Framework

As described in the previous section, it is possible to see that there are many similarities between the different protocols. In the existing framework there is a lot of copy-paste code, just to follow the standard UVM methodology. The drivers and monitors can later be reused, and it is therefore a relatively decent approach. A question that should be asked is if there is any possible way to improve the existing framework. The answer to this question, is that it might be improved with a more complex solution. A Universal Verification Component

which encapsulates common functionality for most busses and let the user adjust the behaviour of this UVC according to his need, could be used. This could also decrease the amount of code needed to write, including copy-paste code, and gain a functional framework earlier than if a UVC would not be utilised. An additional advantage with using a UVC is that it could result in higher reusability, explained in Chapter 2.1.1, which again could decrease the total time spent in verification, with the increased probability of delivering the project on deadline.

3.2.1 Attempt to Create a Unified Agent

There has been an attempt to start the process of creating a unified agent, but only for some of the protocols, on which only on the monitor and agent has been modified. This has been done by creating a base template component, and then extending it for each of the different protocols. The derived class overrides the parameterisation of the base class with its specialised types from the protocol. The agent base class was parameterised with the driver's, monitor's, sequencer's and configuration object's data type, as shown in the code below. The code also shows how the agent utilises a function that must be implemented in the extended classes in order to retrieve the correct configuration from the configuration database. It can also be seen how the driver for the CPU with cache is created.

```
virtual class generic_agt #(type sqr_type=int,
                            type drv_type=int,
                            type mon_type=int,
                            type cfg_type=int) extends uvm_agent;
...

virtual function void build_phase(uvm_phase phase);
     get_configs_from_uvm_config_db;
     mon = mon_type::type_id::create("mon", this);
    if (cfg.is_active == UVM_ACTIVE) begin
      sqr = sqr_type::type_id::create("sqr", this);
      drv = drv_type::type_id::create("drv", this);
      drv.cfg=cfg;
    end
    if(cfg_cache.is_active == UVM_ACTIVE) begin
      drv_cache = drv_type::type_id::create("drv_cache", this);
      drv_cache.cfg=cfg_cache;
```

```
    end
endfunction: build_phase
```

The code below shows how an agent extends from the generic agent and specialises it. It is also shown how the function for retrieval of correct configuration is implemented.

```
class dbus_agt extends generic_agt#(dbus_sqr, dbus_drv, dbus_mon, dbus_config);
...
virtual function void get_configs_from_uvm_config_db;
    if (!(uvm_config_db #(uenv_config)::get(this,"", "env_config", env_config)))
      `uvm_error(get_name(),"No env config in uvm_config_db")
    this.cfg=env_config.dbus_cfg;
    this.cfg_cache=env_config.dbus_cache_cfg;
endfunction
```

The base parameterisation arguments for the monitor was the virtual interface, configuration, and sequence item. The base monitor class only included the constructor and the build phase, which constructed an analysis port. Using this approach, the extended monitor is still required to implement the entire run-phase, for every protocol, as the code shown in Chapter 3.1.8.

3.2.2 Considerations when Creating a Unified UVC

Creating a unified UVC may not be a bed of roses, as obstacles and difficulties may arise, as it will be more complex than a non-generic counterpart. Things to consider when starting the creation of a more generic and unified verification component is:

- Item is split into signals in driver according to classic UVM, but now item should be moved into interface. How to parameterise interface for a given sequence item
- Driver and monitor should become compatible with every protocol
- What to put into the configuration database

- Retrieving and configure correct information for a specified protocol
- Communication between interface to the rest of the hierarchy
- Correct passing of configuration objects throughout the hierarchy
- Correct build and connect throughout the hierarchy
- Avoid accessing functions or variables that do not exist
- Parameterisation
- How functionality can be expanded, if built-in functionality of UVC is insufficient
- Unified reporting mechanism
- Differentiate between generated components
- Avoid racing conditions between interface and DUT

Chapter 4

Modifying the Verification Infrastructure

This chapter will go through the process that has been taken in the development of a UVC. It starts with describing initial steps taken in order to have a code that can be used for comparison with the finished product. The chapter will then continue to explain the process that has been used, and alterations to the sequence item. Chapter 4.4 starts elaborating the process of moving functionality in order to create the UVC. The rest of the chapter will also go through steps that has been taken in order to adapt the UVC to the framework.

4.1 Creating a Starting Point

The framework described in the previous chapter was initially intended to be used as a starting point for the creation of the UVC. However, the use of inheritance in the framework was not considered optimal to use as a base. Figure 4.1 shows what agents in the framework extended from the "generic" agent. The extended agents all had a function that retrieves configuration through the configuration database for the specific protocol. The derived classes inherited the functionality to create the driver, monitor etc., and how they are connected from the generic agent. The monitors for the same protocols were also derived from a generic monitor.

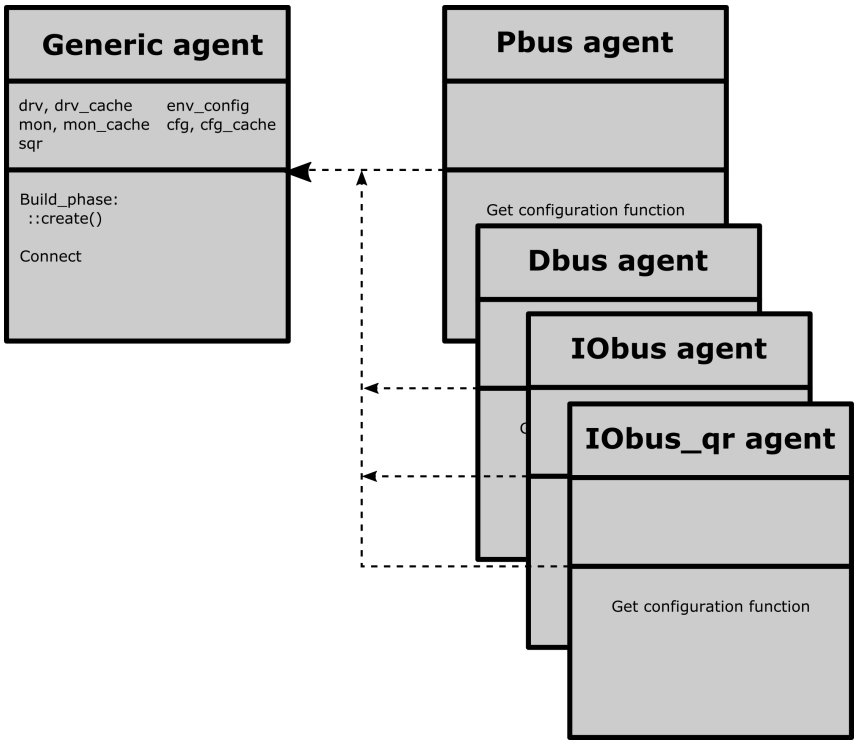


FIGURE 4.1: Agents extended from base class

The framework was modified to use use less inheritance by reverting the extended classes back to their own speacialised base class. In the newly created framework, the different protocols all contained their own, complete functionality, as presented in Figure 4.2. Everything in the protocols was modified, with the exception of the configuration objects in the Dbus, Iobus, Iobus_qr and Pbus protocols, which still inherited some variables and functionality from configuration object's base class.

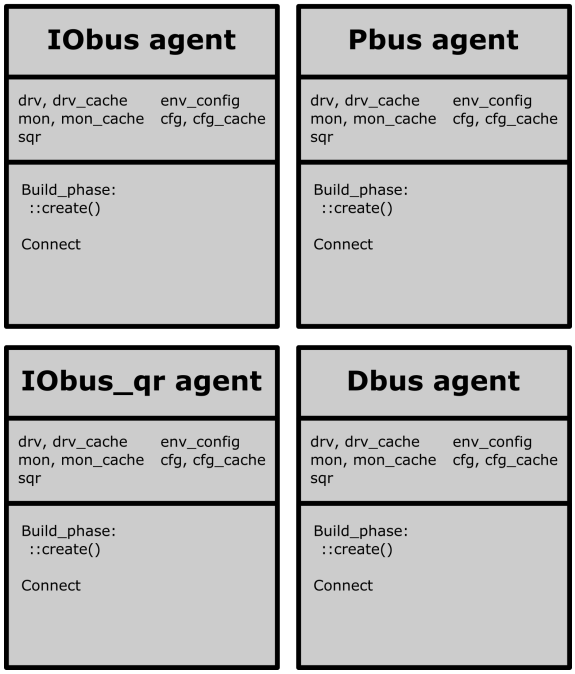


FIGURE 4.2: Specialised agents

Using a framework, that does not rely a lot on a complex parameterisation scheme as a starting point, could make it easier to see benefits and drawbacks of the changes in the components, compared to the final framework. Table 4.1 presents the total amount of code lines in the framework. Since there are eight protocols, where each protocol holds the same six sub-files, the table is partitioned such that the code lines for the different components are added together, giving the total amount of code for all of the sub-files combined. The code lines were counted by using a "regular expression", `^.*\S+.*$` , on every file, as it count every line

that is not empty. Eventual comments were then subtracted to give an accurate indication of active code lines.

Sub-component	**Lines of code**
Agents	258
Configuration objects	215
Drivers	254
Interfaces	117
Sequence items	286
Monitors	236
Sequencers	59
Total	1425

TABLE 4.1: Number of code lines in non-generic code

4.2 Starting from the Bottom

As an initial approach to create the UVC, modifications to the agent were commenced. This was quickly discovered not to be a good idea. This is because, as Figure 4.3 illustrates, modifying or replacing a block that is high up in the hierarchy needs to be compatible with all the blocks under. When starting with the agent, all the sub-components had to be altered to fit with the modified agent, which meant that small changes would be very time consuming.

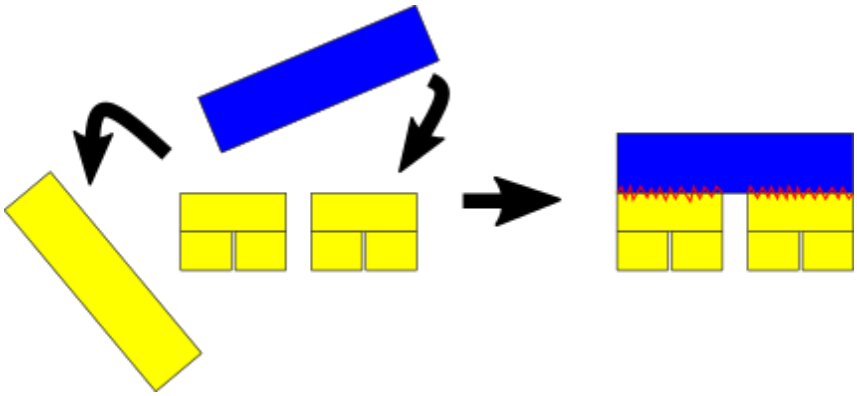


FIGURE 4.3: Changing the top block

In contradiction to the top to bottom method, a method that starts from the bottom, and do changes upwards in the hierarchy was utilised in this project.

Figure 4.4 illustrates how replacing, or modifying of one block at the bottom of the hierarchy, only has to be adapted to the block above in the hierarchy. Starting the modifications at the bottom, instead of the top, made it possible to verify that the framework was functional and operated as intended, at a more frequent rate. This was because the modifications were smaller, and less time was being used debugging.

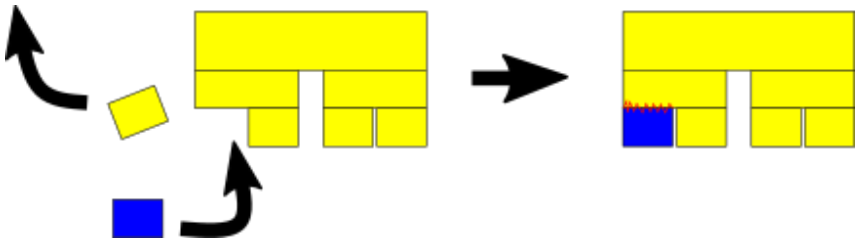


FIGURE 4.4: Changing bottom block

Making adjustments to the framework from top to bottom might also make debugging and finding a fault more demanding and time consuming, as there might be dependencies between objects, or that a fault might propagate in the hierarchy. Figure 4.5 illustrates that when using the bottom up method, the bug will most probably be in the new/modified block, while in the other method the bug could be propagated from an unknown block further down in the hierarchy.

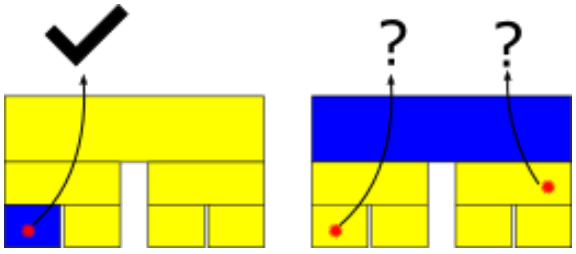


FIGURE 4.5: Bug propagation in the different methodologies

4.3 Improving the Sequence Items

It was seen that the sequence items used in the verification infrastructure had potential to become more efficient, as it could be possible to decrease the total amount of code lines, and to create an faster and easier way adding new items

for new protocols. As mentioned in Chapter 3.1.11 most of the sequence items included in the framework share a lot of the same, or similar variables and functions. By looking at the similarities from the different protocols and utilising inheritance, the generation of a "generic" sequence item, that could be used by most protocols, or be extended for further functionality, was started. In the development of this item, the Pbus, Dbus, Iobus and Iobus_qr items were used as a base, and there was also taken a look at the protocols' drivers and monitors for special cases.

For the the generic sequence item to function as intended for the different protocols, it was crucial that it could handle its objectives. Since the variables in the sequence items might not exist or have different sizes, it was first created a sequence item only consisting of the common variables, and the do_* methods associated with these variables. It was originally thought to use this sequence item as a base, before extending it for further functionality or variables. A mayor drawback with this solution was that the extended items still had a lot of code that had to be added, such as having to implement the do_* methods for the added variables. Further development of the sequence item resulted in the different busses with different width was included in the sequence item. To compute with the different widths, the class was parameterised as shown under along with the rest of the variables included in the generic sequence item.

```
virtual class generic_item#(int msize=16, int asize=16, int ssize=16)
        extends uvm_sequence_item;

        rand logic   [ssize-1:0]    sdata;
        rand bit     [asize-1:0]    bus_m_addr;
        rand logic   [msize-1:0]    mdata;
        rand t_cpu_version          cpu_version;
        logic                       reset;
        logic                       re;
        logic                       we;
        string                      s_name;
        bit                         preload_nvm=1'b0;
        bit                         done;
        bit                         response=1'b0;
        rand int                    delay;
        rand logic [1:0]            gen_wait;
        ...
```

Where **mzise** decides the size for the write-data, **asize** for the width of the bus-address, and **ssize** sets the size for the read-data. In addition was the variable name for the bus-address changed from a specific name, like **pbus_m_addr**, to a more general "**bus_m_addr**". This change was done throughout the hierarchy, for all the components that used it. In addition to this were the **delay** and **gen-wait** variables from the driver moved to the item. Even though the some of the variables necessarily are not used in the item for a specific bus, they are still used by other busses, and was therefore included. It also grants more positive than negative outcomes in the end. One of these positives is the reduction of the implementation of do_* methods in the extended classes. Constraints that could be found in the sequence items and drivers were consequently moved to the extended class of the item.

```
virtual class generic_item#(int msize=16, int asize=16, int ssize=16)
        extends uvm_sequence_item;
...
function void do_copy(uvm_object rhs);
    ...
function void do_record(uvm_recorder recorder);
    ...
virtual function string convert2string();
        ...
virtual function void do_print(uvm_printer printer);
    ...
```

The do_* methods, see Chapter 2.1.12, implemented in in the generic sequence item can be seen in the code above. Even though some the variables in the item was not used, it does not have an impact on the *do_record* method. If more functionality is needed on an added item, it is possible to extend the generic item, add additional variables, use super.build to extend the functions from the generic item, and add other functions if necessary.

```
`define make_generic_item(classname,msize,asize,ssize)\
typedef class generic_item;\
  class ``classname extends generic_item#(``msize, ``asize, ``ssize);\
        function new(string name = "");\
                super.new("");\
        endfunction\
        `uvm_object_param_utils(``classname#(``msize, ``asize, ``ssize))\
endclass
```

In addition to having the ability to extend the class, there has also been defined an macro that takes **classname**, **msize**, **asize** and **ssize** in as arguments, and constructs a generic item with these. The macro is given above, it has however not been utilised in the framework.

4.4 Moving Functionality

After the sequence items had been improved the focus was moved to the blocks higher up in the hierarchy, like the driver, monitor, sequencer and configuration object. It was these components that should become the foundation in the creation of the UVC. Common for the driver, monitor and sequencer was that each of them are specialised with the protocols item and that they had to connect to the corresponding interface. All of the components would eventually become generic, with the exception for the configuration object, which has to be specific for each of the protocols, as it would be used to determine how the UVC should be built.

4.4.1 The Interfaces

As mentioned in Chapter 2.1.10, can the interfaces in SystemVerilog encapsulate functionality and connectivity. Taking advantage of this, and the fact that UVM utilises Transaction Level Modeling, see Chapter 2.1.11, has been used in the development of the UVC. Concepts like dual top and abstracting the communication, mentioned in Chapter 2.2.2, were also in mind in the development of the interface and its communication role in the framework. These concepts were considered as it makes for a natural division between the driver and interface. It was also considered as it could help create an emulation ready framework in the future.

```
interface dbus_if #(type signal_type = int,  type item_type = int)
  (input logic clk);

  signal_type signals;
  item_type item;
```

The first thought was to utilise parameterisation in the interfaces, as shown in the example snippet above. This is possible since the interfaces can be modelled as a module, and therefore also parameterisable. The parameters considered was for signals and for items. Where the signals are the internal signals related to the protocol, and the item should come from the sequencer. With this set-up, the signal and item has to be parameterised throughout the hierarchy, including the the environment, which may cause an extra inconvenience in comprehending the verification infrastructure.

```
interface dbus_if #(type item_type = int)
  (input logic clk);

  mem_byte_s_t                dbus_s;
  mem_byte_m_t                dbus_m;
  item_type item;
```

The parameter for the signals was removed and replaced with packed structs that could be used to access the DUT. This made the components accessing the interface less bound, as it was possible to add several structs, as shown in the code above. Additional tasks and changes that are added and done to the interface will be given Chapter 4.4.3.

```
interface dbus_if
(
  input logic clk, reset
);
  mem_byte_s_t                dbus_s;
  mem_byte_m_t                dbus_m;
  logic                       dbus_master_sc    =1'b0;
  logic                       dbus_flash_access =1'b0;
  logic                       dbus_io_access    =1'b0;

  dbus_item                   item;
```

It was later discovered that the same could be done with the item type. This is because the interfaces has to be tailored for every protocol, and does therefore

not need to be parameterised, as it can declare the items that is needed. The code snippet above presents how the final interface would be, where the interface access the DUT's signals with the use of packed structs, and the declaration of sequence items is used. It is also possible to see that with this solution, it is possible to access several signals, and that there is no use of parameterisation. The communication between the driver, monitor and interface will be through tasks and sequence items, and is explained more in detail in Chapters 4.4.3 and 4.4.4.

4.4.2 Configuration Object

In order to keep the amount of code in the configuration objects to a minimum, and have a structure that makes sense, the configuration object class template base is kept. The protocols can inherit from this class for faster integration of a new protocol, which helps facilitate reuse. The function to generate wait signals was moved to the interface. This was done because all the variables used were accessed through the interface, and by moving it, makes it easier to get an overview of the verification infrastructure. The base template for the configuration objects is given in the code below.

```
virtual class config_template#( type vif_name=string )extends uvm_object;

  vif_name vif;

  uvm_active_passive_enum is_active   = UVM_PASSIVE;
  bit coverage                        = 1'b1;
  int agent_verbosity                 = UVM_NONE;
  bit allow_data_generation           = 1'b0;
  bit allow_inst_generation           = 1'b0;
  rand bit allow_generate_swait_bwait;
  rand bit lock_sequencer;


  //Wait functions
  virtual task wait_for_reset;
        ...
  virtual task wait_cycle(int delay);
        ...
```

```
  virtual function void disable_interface_assertions;
        ...
  virtual function void enable_interface_assertions;
        ...
  virtual function string convert2string();
        ...
  virtual function void do_print(uvm_printer printer);
        ...
```

It is possible to see what functions can be inherited, that the virtual interface has to receive its data type through parameters and how the other variables are initialised. Some of the variables has been declared as **rand**. This means that when the object is randomised, these variables will be randomised. This is used to increase the constrained random verification, which is core in UVM. There are functions that can be used to enable or disable assertions in the interface, which can used for debugging purposes.

```
class pbus_config extends config_template #(virtual pbus_if);
  `uvm_object_utils(pbus_config)

   int pbus_seq_min_random_count = 2;
   int pbus_seq_max_random_count = 5;
   bit allow_inst_generation     = 1'b1;
```

The extended configuration objects has been specialised with the virtual interface handle, for the specific protocols, as seen in the code above. It also shows that there are added and changed variables for this specific object. All the protocols in the framework has been modified to extend from the base class. How the virtual interface is set with the configuration database to the configuration object is given Chapter 4.4.6.

4.4.3 Attacking the Driver

The drivers in the framework have different specialised functionality, and as Figure 4.6 illustrates, the functionality of all of these drivers needs to fit into one generic driver. The approach taken to achieve this was by starting with one of the drivers,

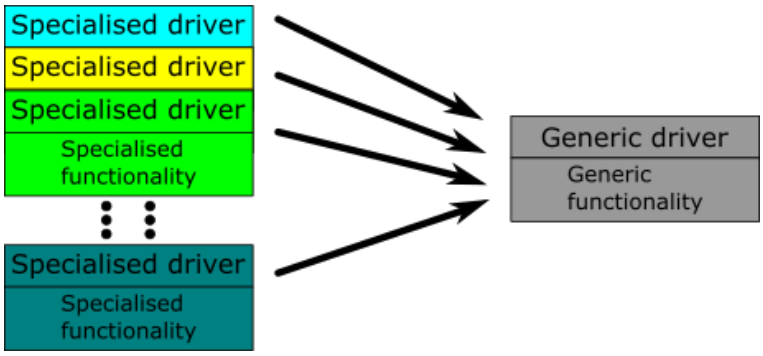


FIGURE 4.6: From many specialised drivers to one generic driver

in this case the Dbus, and modify it for generic functionality. Figure 4.7 presents a rough illustration of the approach taken. The specialised driver, which resides in the Dbus-agent, have the same specialised type for the virtual interface and configuration object as well as its specialised functionality(1). The driver was then changed for a generic driver, which still kept the types that belonged to the specialised driver(2). The last step generalised the data types used in the driver and finding a way to pass the correct data types to the driver, for example by utilising parameterisation(3).

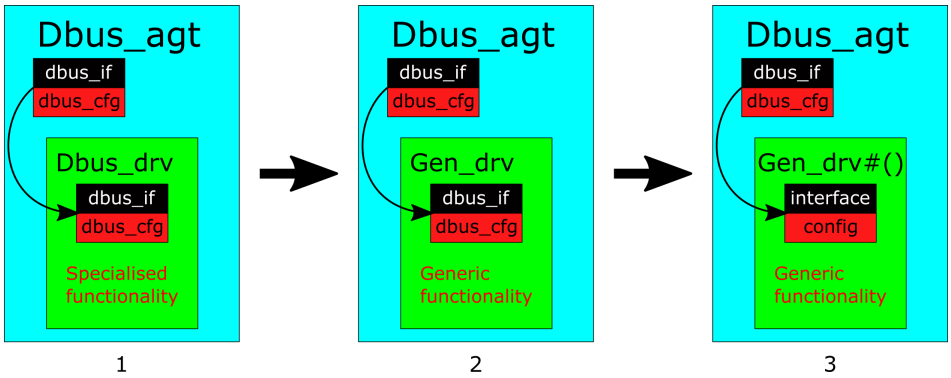


FIGURE 4.7: Creation of generic driver

An advantage of altering a driver with this approach was that it was possible to make several small changes instead of a few big ones. This most probably saved time debugging, and made it easier to verify that the functionality of the testbench had not changed. This approach also gave a good base for what should be included when this class was brought in and adapted to the next protocol.

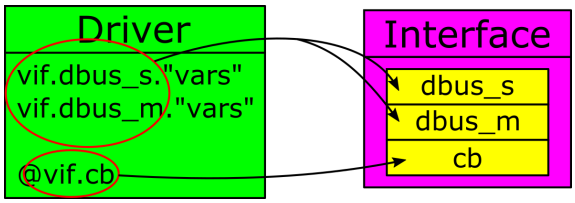


FIGURE 4.8: Driver access variables through the interface

As mentioned was the Dbus the first protocol out. The first thing that had to be done was to change the handle for the function that generates wait for master and slave, from **cfg** (configuration object) to **vif** (virtual interface). The functions and tasks in the driver accessed the signals of the DUT directly through the interface, illustrated in Figure 4.8, using the handle for the signals and clocking block is in the driver. These accesses were common for the drivers and monitors, and thus the initial idea was to generalise the name. The code snippet under presents how the signal-names from the driver was altered, such that it would not say **dbus_s**, but **bus_s**. This was done in both the driver and the interface. There was several problems that arose with this approach. The biggest problem may be that some of the interfaces did not contain the signal attempted accessed. This could cause constraints that decrease the reusability of the driver, which is against the purpose in this project. In addition to this, other big changes had to be done to the testbench, like changing the tests and all the busses' structs. The tests' accesses to these signals made it hard to keep control of what signals were from what protocols, As it generally was an unstructured approach, the changes was reverted and this solution did not happen.

```
//original
    vif.pbus_s.swait <=1'b0;
    vif.pbus_s.bwait <=1'b0;

//new
    vif.bus_s.swait <=1'b0;
    vif.bus_s.bwait <=1'b0;
```

In an attempt to gain high reusability, and generalise the driver, the idea of using tasks and items to communicate, instead of signals, were set into action. Figure 4.9 illustrates how a task could be called from the driver to generalise the it, and

still get the specialisation that is needed of the protocols through the interface whose handle is in the driver. In this project it is impossible to move all the code over from the driver to the interface. It also does not make any sense to do so, as the problem would be moved, and the functionality of the driver is important.

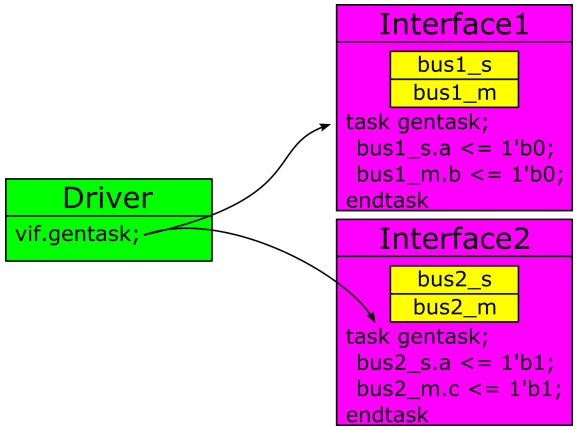


FIGURE 4.9: Use of tasks to communicate with interface

In order to achieve generic functionality on the driver, and move the specialised parts over to the interface, a set of tasks that is implemented in the interface were mixed into the drivers functionality. The tasks that were called upon in the driver can be empty if there is no need for it in the protocol. This induced the opportunity to have several smaller tasks, that may or may not have functionality in the interface. With a set-up in the driver that manages to execute the different tasks in such a way that it can be used for every protocol, the driver can be considered generic.

```
interface dbus_if
...
    task init_content;
      dbus_s.swait <=1'b0;
      dbus_s.bwait <=1'b0;
      dbus_s.sdata <=16'h0000;
    endtask : init_content
```

Originating from the Dbus, a set of tasks was developed. The first task, *init_content*, is used to initialise signals on the DUT. For the Dbus protocol, it initialise the

signals from the slave to zero, as seen in the code above, such that there are no wrong initial values on the bus.

After the driver checks the configuration object whether data generation is allowed, it will start to drive the DUT by setting signals. As mentioned in Chapter 3.1.1, the Dbus driver has a memory that is used when driving the bus. The master bus' read and write bits are checked, and if they they are low, the driver will request a transaction from the sequencer. This memory was moved to the interface, as this was type specific for the protocol. Since the driver had to use this for correct functionality, a task that sends the sequence item over to the interface was developed. In the code below, it is possible to see that the *init_content* task is called before the forever loop with the creation of the sequence item. The created sequence item is used in the *generate_content*-task.

```
class gen_drv#(...
...
    vif.init_content;
    forever begin
        req=item_type::type_id::create("req", this);
        vif.generate_content(req);
    ...
```

It was the use of sequence items that made it possible for correct communication between the the UVC and interface. As a problem arose when the data did not exist in the memory, and the sequencer had to be called, there was need for additional functionality in the driver. As a call to the sequencer can not be made from the interface, and to maintain correct functionality with the memory checking, a fix was needed.

```
class gen_drv#(...
...
if (vif.pull_req == 1'b0) begin
  seq_item_port.get_next_item(req);
  vif.generate_content_IF(req);
  seq_item_port.item_done();
end
```

The solution to this problem was the use of a bit, that was added in the interface. This bit, called **pull_req**, indicates if there is a need to pull a request from the sequencer. If the **pull_req** bit is low, the driver will retrieve the requested item from the sequencer, as seen in the code from the forever-loop in the driver above.

```
interface dbus_if
...
    task generate_content(dbus_item req);
        if(dbus_m.re) begin
          pull_req = 1'b1;
          if(DRAM.exists(dbus_m.addr))      //Check if data exists in DRAM:
            drive_bus(DRAM[dbus_m.addr]);
        end
        else if(dbus_m.we)
          begin
            pull_req = 1'b1;
            req.sdata=dbus_m.mdata;
            DRAM[dbus_m.addr]=req;
          end
        else
          pull_req = 1'b0;
      @cb;
    endtask
```

The implementation of the *generate_content* task in the Dbus interface is given above. The task checks if there is a need for a pull request, and sets the **pull_req** bit accordingly. As seen, as long as read or write enable is not low, the **pull_req**-bit will never go low. The *generate_content* function in the interface has a **@cb** at the end of the function. This is to get correct functionality with the forever loop in the driver. The driver will check the **pull_req** bit for every iteration in the forever loop, and the clocking block force the loop to a run at controlled pace.

generate_content_IF is the second task called from the forever loop. As seen in the code below, the task copies the address from the master bus to the items address, and copies the item into the memory, before the bus is driven. The *generate_content_IF* should be used to drive the the sequence item that is requested from the interface for all the protocols.

```
interface dbus_if
...
    task generate_content_IF(dbus_item req);
      req.bus_m_addr=dbus_m.addr;
      if(^dbus_m.addr===1'bx)
        $error("dbus_if, dbus_m.addr contains X, addr=%b", dbus_m.addr);
      DRAM[dbus_m.addr]=req;
      if(!DRAM.exists(dbus_m.addr)) begin
        $warning("Skipping RAM, driving req directly");
        drive_bus(req);
      end
      else
        drive_bus(DRAM[dbus_m.addr]);
endtask : generate_content_IF
```

The last task in the driver was *generate_swait_bwait_only*. This function randomises the variables, primarily gen_wait and delay, before it calls *genereate_swait_bwait*. The functionality of the task is kept the same, with the exception that an additional task, *generate_swait_bwait_IF*, is created. This task is called from *generate_swait_bwait_only*, after the randomisation and check whether it should generate the wait, from the configuration object. The *generate_swait_bwait_IF*-task is added in order for the driver to be used for many protocols, and it is therefore possible to add additional specialised functionality if necessary.

4.4.4 Modifying the Monitor

The monitor is a crucial component in the UVM testbench, and it is important that it functions correctly. As mentioned in 3.1.8, is the monitors using a special handshake for when it should kick in and monitor the DUT. In the creation of a monitor for use on different protocols, the same approach as the driver was used, starting with one protocol and altering it, small pieces at a time.

The monitoring of the signals was the first obstacle explored in the development of making the monitor generic. As with the driver, the use of tasks and a sequence item was utilised. A task named *convert2Item* was created. This task was first implemented and called in the driver, before the implementation was moved to the interface. Figure 4.10 illustrates how the monitor and interface utilise the

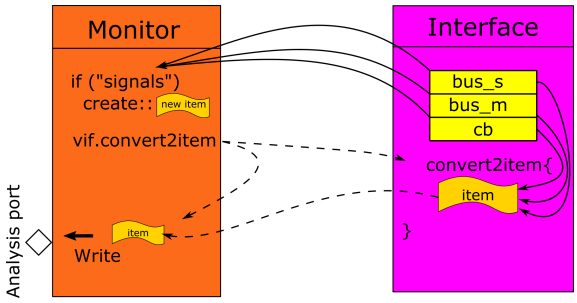


FIGURE 4.10: Converting signals to item

convert2Item task, which is given below for the Dbus. The monitor will check the signals from the bus, which can be considered the handshake, before the an "empty" sequence item is created. Instead monitoring the signals from the interface directly, as it used to, the *convert2Item* task is called and the specific signals are copied onto the sequence item, which is written to the analysis port.

```
interface dbus_if
...
    task convert2Item(dbus_item item);
            item.re     =cb.dbus_m.re;
            item.we     =cb.dbus_m.we;
            item.bus_m_addr=cb.dbus_m.addr;
            item.mdata  =cb.dbus_m.mdata;
      @(cb.dbus_m.addr);
           item.sdata  =cb.dbus_s.sdata;
    endtask : convert2Item
```

An interesting challenge was how to handle the handshake. The first solution considered was to remove the handshake, and sample at every clock edge. However, this solution did not work well with recording transactions for usage in visual aids, like waveforms. This is because, even though the monitor checks the signals correctly, there would be a transaction at every clock edge, ruining the duration of the transaction. This would not result in any visual aid for the verification engineer.

```
//////Interface:
always @(cb)
```

```
  begin
    run_bit = (!cb.dbus_s.swait && !cb.dbus_s.bwait && cb.dbus_m.re && !cb.reset);
  end

//////Monitor:
forever begin
  @vif.cb;
  if(vif.run_bit)
    //start monitoring
end
```

The second solution was to add a new bit in the interface, almost like the **pull_req**-bit, that decides whether or not the monitor should start its transaction. An alternative to the bit-solution was to use **events**, as described in Chapter 2.2.3. The code above shows how the use of a **run_bit** can be implemented in the monitor and interface. The drawback of using the **run_bit** solution is that it is necessary to trigger on the interface's clocking block in the monitor before checking the bit's value. By using **events**, see code below, it will trigger when there is an event instead. In this infrastructure **events** are utilised, as the initial thought was to keep clear of accessing the clocking block from the components.

```
//////Interface:
always @(cb)
  begin
    if (!cb.dbus_s.swait && !cb.dbus_s.bwait && cb.dbus_m.re && !cb.reset)
      -> e_mon;
  end

//////Monitor:
forever begin
  @(vif.e_mon);
    //Start monitoring
end
```

With the use of the task that retrieves the item from the interface, and as the handshake got a solution, the rest of the monitor is quite simple. It is parameterised with default data types, like **config_type**, that will be overwritten when implemented in the the environment. It then continues to register the class with

the factory, creates the analysis port before it goes on to run the monitor, which can be seen in the code below.

```
class gen_mon#(...
...
    task run_phase(uvm_phase phase);
        run_monitor;
    endtask: run_phase


    virtual task run_monitor;
      item_type item;
      forever begin
        @(vif.e_mon);
          item  = item_type::type_id::create("item");
          item.enable_recording(cfg.name);
          begin_tr(item, cfg.name);
          vif.convert2Item(item);
          ap.write(item);
          end_tr(item);
      end
...
```

The code also shows the solution to another issue that appeared in the process of generalising the monitor. It was regarding the recording, and the transaction name that is attached. For the specialised monitors the name had previously been added with a string, but would result with the same name for every monitor if this was done in the generic class. To get a distinct separation between the different monitors, a new variable was included in the configuration object. This variable takes in a string for adding the name of the protocol, and was added in all the configuration objects. Additionally to have a clear separation between the original bus and the bus that use cache, the cache-bus' name will automatically add "_cache" in the build phase in the agent.

4.4.5 Parameterisation

Using the initial verification infrastructure as a base, the development of the new unified agent was started. The agent was initially parameterised with the sequencer, driver, monitor and configuration as arguments. This idea was taken from the original framework. It was initially thought that this would be an easy way to parameterise the agent, but as it is possible to see in the code to declare the agent below, the parameterisaton in the environment became big and repetitive as the sub-components also are parameterised. The code underneath is not completed, it is only the driver that is generalised, meaning that a fully generic agent with this parameterisation would be even bigger, and therefore be prone to errors and become time consuming for the verification engineer. This was also the parameterisation before the interfaces was simplified.

```
class uenv extends uvm_env;
...
  generic_agt#(     dbus_sqr,                          //sqr_type
                    generic_drv#(                      //drv_type(
                        dbus_if#(dbus_item),           //vif_name#(item_type)
                        dbus_item,                     //item_type
                        dbus_config),                  //config_type
                    dbus_mon,                          //mon_type
                    dbus_config)      m_dbus_agt;      //config_type
```

To prevent such a complex and unnecessary set-up, the parameterisation of the agent had to be altered. Instead of parameterising the agent with the the sub-components that created a chain of different parameterisations, it was taken a closer look on how the agent could be parameterised in a compact and effective way.

This was done by looking at what the sub-components of the agent was parameterised with, which revealed that they shared the same parameters. The agent was therefore parameterised with the virtual interface type, item type, and configuration type. As described in Chapter 4.4.1 was the parameterisation of the virtual interface simplified, causing it to no longer need sequence item. Declaring the agent in the environment was therefore simplified, as seen under.

```
generic_agt#(   dbus_if,                    //vif_name
                dbus_config,                //config_type
                dbus_item)      m_dbus_agt; //item_type
```

Later it was discovered it was possible to further simplify the parameterisation of the agent, and is described in Chapter 4.4.10.

4.4.6 Changing the Configuration

Since the UVC was under development is the "generic"-agent presented above actually just the Dbus-agent that builds the "generic"-Dbus' sub-components. This was because the access to the configuration database still retrieved the configuration through the environment configuration object with a hierarchical look-up, where the look-up string still had not become adapted to the generic component. In order to achieve a correct generic build, the different agents' configuration objects were moved from the environment's configuration object, that was mentioned in Chapter 3.1.3. This removed the extra layer of hierarchy that had to be accessed in the original framework. Figure 4.11 shows how the agents configuration object should get the configuration directly from the configuration database, and then pass this configuration to its sub-components.

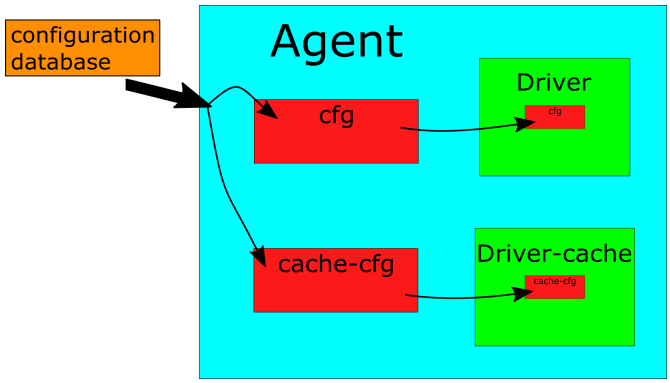


FIGURE 4.11: Configuration object set by database

In order to move the configuration objects from the environment's configuration object, it was necessary to alter the structure of the test. The configuration object was moved to, and declared, in the test base. It was also necessary to modify the

set functions for the configuration database, in the test base and all the test-classes that extended it. A consequence of altering the set-functions, was that accesses to the configuration database also had to be modified.

The issue with retrieving the correct configuration for the agent still persisted. Figure 4.12 illustrates how the environment should be used to build the rest of the hierarchy. The environment is created in the test, and the specialised generic-agents in the environment. As mentioned could the agents no longer use the hierarchical look-up from the environment's configuration object. In order to retrieve the correct configuration, the use of the hierarchy and the fact that the components are very similar was exploited. Utilising the configuration database's *get-* and *set-* function, alongside with the use of wildcards, see Chapter 2.1.19, made it possible to pass the configuration through the generic hierarchy.

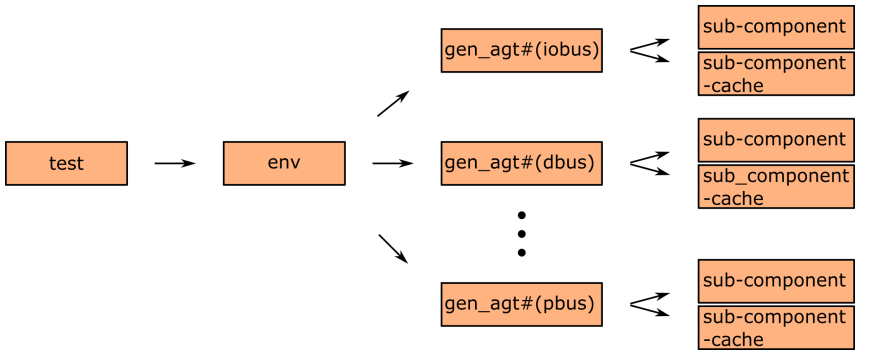


FIGURE 4.12: Verification infrastructure build order

The agents' handles was used as the splitting point for the different agents. This along with the use of the wildcard, *, and a phrase for specifying the end made it possible to set the right configuration to the specified generic agent. The code below shows how the configuration for the Dbus-agent is set from the test base. Take notice that the configuration database needs to know the configuration objects type, and that it is specified. This method of setting the configuration made it possible for the agents to utilise a generic *get* function, but still retrieve the specialised configuration.

```
uvm_config_db #(dbus_config)::set(this, "*m_dbus_agt*", "config_type", dbus_cfg);
```

In order to get the object in the agent, the configuration database's *get* function was applied. The code below shows the *get*- function, called from the generic agent. As seen has all the variables and strings become generalised. The agent was parameterised with the configuration object type, **config_type**, which has to be set to a specified type in the environment, since the configuration database needed the type of the object in order to be used.

```
if (!(uvm_config_db #(config_type)::get(this, "", "config_type", cfg)))
      `uvm_fatal("Agent", $sformatf("Missing configuration in agent %m"));
```

How the virtual interfaces are set to the configuration object also had to be modified. The code under shows how the code is retrieved and set using the configuration database in the test, for the use with and without cache.

```
if (!uvm_config_db #(virtual dbus_if)::get(this, "","dbus_if", dbus_cfg.vif))
    `uvm_fatal(get_name(), "Can't read dbus_if from config_db");
uvm_config_db #(dbus_config)::set(this, "*m_dbus_agt*", "config_type", dbus_cfg);

if (!uvm_config_db #(virtual dbus_if)::get(this, "","dbus_if_cache",dbus_cache_cfg.vif))
  `uvm_fatal(get_name(), "Can't read dbus_if_cache from config_db");
uvm_config_db #(dbus_config)::set(this, "*m_dbus_agt*",
        "config_type_cache", dbus_cache_cfg);
```

4.4.7 Changing the Cache Setup

The agent is supposed to build components for a verifying a CPU with and without cache, as mentioned earlier. In the initial verification infrastructure the agent only built the driver, and not the sequencer and monitor. It also built the cache-driver every time, even if it was not used. To make the building process more effective, an additional configuration bit was added to the configuration object. This bit decides whether or not the caches components should be built and if an additional call should be made to the configuration database for the cache's configuration, see Figure 4.13.

If the cache-enable bit is active, it will add "_cache" to the name variable in the configuration object, as mentioned earlier. It then gets the configuration from the

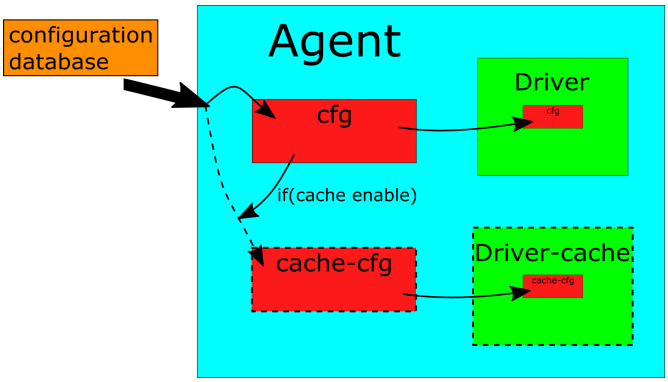


FIGURE 4.13: Configuration objects with cache enable bit

configuration database, before it builds the newly added monitor. Following, it checks the cache-specific configuration if it should be built as an active agent, and builds the driver and the new sequencer, as a normal verification component. The implementation of how the agent's build-phase handles the cache is given below. The agent utilises the same bit in the connection phase to check if it should connect to the cache-components.

```
class gen_agt#(type config_type = int, type item_type = int)
                 extends uvm_agent;
    ...
  function void build_phase(uvm_phase phase);
    //build regular agent...
    ...
    if(cfg.enable_cache) begin
      if (!(uvm_config_db #(config_type)::get(this, "", "config_type_cache", cfg_cache)))
        `uvm_fatal("Agent",$sformatf("Missing cache-configuration in agent %m"));

        cfg_cache.name = $sformatf(cfg_cache.name, "_cache");
        mon_cache = gen_mon#(config_type, item_type)::type_id::create("mon_cache", this);
        mon_cache.cfg = cfg_cache;
        if(cfg_cache.is_active == UVM_ACTIVE) begin
          sqr_cache = generic_sqr#(item_type)::type_id::create("sqr_cache", this);
          drv_cache = gen_drv#(config_type,item_type)::type_id::create("drv_cache", this);
          drv_cache.cfg=cfg_cache;
        end
    end
```

The agent in this framework will therefore work as a two-in-one agent, where it creates a double set of sub-components, with the use of two different configuration objects. The sub-components are the however the same type, and will receive the same sequences from the virtual sequencer. An additional monitor can be used to compare results with the other.

4.4.8 Just a Little Sequencer

The sequencer is the component with the least differences between the different protocols. The generic sequencer was extended from the **uvm_sequencer** and parameterised with a sequence item. The sequencer was registered with the factory using the *'uvm_component_param_utils(generic_sqr#(item_type))* macro. In the build phase there is a call to the configuration database to access to the environment configuration object, which can be used by the sequences. As the generic sequencer took the specified sequencers spot, there was changes that had to be done in order to achieve a functional testbench:

- The virtual sequencer had to swap the old sequencer with the new parameterised sequencer.
- All the sequences needed to change its declared sequencer.
- Small changes to some of the sequences
  - As the configuration object for the Pbus protocol is no longer in the environment configuration.

4.4.9 Further Improvement of the Components

After a functional and working hierarchical build was made, it was time to start the integration of the other protocols. Since the Dbus could be counted as integrated, the Pbus protocol was decided to be next in line, as this protocols functionality was the most different from the Dbus'.

In order generalise the Pbus' functionality into the generic driver, the same approach as earlier was utilised, where the agent was from the Pbus protocol and the driver was altered to become generic. One of the crucial differences between

the protocols, was the way they handled the transactions in the driver. The Pbus' driver utilised while-loops and response from the items, as well as copying of items, as apposed to the Dbus' protocol. The code used in the original Pbus driver is given below. The other protocols were also taken into consideration at this stage. One thing that stood out was the request to memory. Which meant that the **pull_req** bit should be reused.

```
class pbus_drv extends uvm_driver #(pbus_item);
...
    task generate_content;
        integer addr;
        pbus_item req_cp;
        forever
         @vif.cb
            //Check if data is present in NVM
            if(!tb_uvm.flash_memory.exists(vif.pbus_m.addr))
             //Block generates data, put it into NVM, and then put it on pbus_
             begin
               addr=vif.pbus_m.addr;   //Save current address in local variable
                 do begin
                   port.get_next_item(req);
                   //Create a copy of request to be put into NVM
                   req_cp=pbus_item::type_id::create("req_cp", this);
                   req.pbus_m_addr=addr;
                   req_cp.copy(req);
                   //Put generated sequence item to NVM
                   tb_uvm.flash_memory[addr]=req_cp.sdata;
                   if(req.response) begin
                       rsp=req;
                       rsp.set_id_info(req);
                       seq_item_port.item_done(rsp);
                   end
                   else seq_item_port.item_done();
                   //Jump to drive task if preload_nvm bit is not set
                   if(!req.preload_nvm)
                     break;
                   addr=addr+2;
                 end while (1);
             end
endtask
```

To reduce the amount of sequence items created in the driver, the Pbus driver was altered, as it created and used a sequence item copy. The handling of a response was also included in the generic driver. One of the biggest obstacles was how to deal with the loop and correct address for every cycle. Since there was no use for a while-loop in the data generation, this had to be separated.

The new driver utilises the same set of tasks, for the use in both data and instruction generation. This reduces the amount of tasks necessary in the interface. The memory access and the variable to hold and increment the address, was moved to the interface. The **pull_req** bit has also become a necessary part for all interfaces. As the driver need the sequence item's **response** bit, this means that all the sequence items needs to include this bit.

```
class gen_drv#(type config_type = int, type item_type = int)
      extends uvm_driver #(item_type);
    ...
    task run_phase(uvm_phase);
        vif.init_content;
        fork
          if(cfg.allow_inst_generation || cfg.allow_data_generation) begin
            forever begin
              generate_data;
            end
          end
          generate_swait_bwait_only;
        join_none
    endtask
```

The run phase of the generic driver, as seen above, checks if the configuration object allows for generating data or instructions, and then runs the *generate_data* for the generation. The code that is added to the *generate_data* task for the data instruction generation part of the driver is given below.

```
class gen_drv#(type config_type = int, type item_type = int)
      extends uvm_driver #(item_type);
    ...
    virtual task generate_data;
      req=item_type::type_id::create("req", this);
```

```
    cfg.vif.generate_content(req);
    if (cfg.allow_inst_generation) begin
      do begin
        seq_item_port.get_next_item(req);
        cfg.vif.generate_content_IF(req);
        if(req.response) begin
            rsp=req;
            rsp.set_id_info(req);
            seq_item_port.item_done(rsp);
        end
        else seq_item_port.item_done();
      end while(cfg.vif.pull_req == 1'b0);
    end
    ...
```

In this task the driver first creates the request item and calls *generate_content* from the interface. The *generate_content* task for the Pbus-protocol, shown in the code below, retrieves the address from the signals in the interface, if the address does not exist in the memory. The driver then separates the instruction generation and data generation by looking at the configuration object. If it is data generation, the functionality is like before, if **allow_inst_generation** is high, it starts to loop, and exits the loop depending on the **pull_req** bit. The loop starts with requesting an item from the sequencer, then it calls the next task from the interface. If the **response** bit in the item is set, *item_done* will be called with the response. The task called in the interface will set the data to the memory, and increment the address. It will also set the **pull_req** bit depending on the sequence items **preload_nvm** variable.

```
interface pbus_if
...
    task generate_content(pbus_item req);
        @cb;
        pull_req = 1'b0;
            if(!tb_uvm.flash_memory.exists(pbus_m.addr))
             begin
               addr=pbus_m.addr;
             end
    endtask

    task generate_content_IF(pbus_item req);
```

```
        req.bus_m_addr=addr;
        tb_uvm.flash_memory[addr]=req.sdata;
        if(!req.preload_nvm) pull_req = 1'b1;
        addr = addr + 2;
    endtask
```

As it is possible to see, is the **pull_req** bit used to keep the the loop running in the *generate_data* task. The *generate_content* function sets the status of the **pull_req** bit, checks the memory and sets the initial address. The incrementation of the address needs to be be done in the loop, in other words in the *generate_content_IF* function, which also sets the **pull_req** bit to exit the loop, as seen above.

4.4.10 Minor Modifications

Parameterisation of the agent was earlier described in Chapter 4.4.5, and how it was parameterised with the virtual interface type, item type and configuration type. It was taken closer examination of the framework to determine if there were any simplifications that could be accomplished. As the interface type is parameterised with the item type, the idea that the virtual interface might be redundant as a parameter came to light.

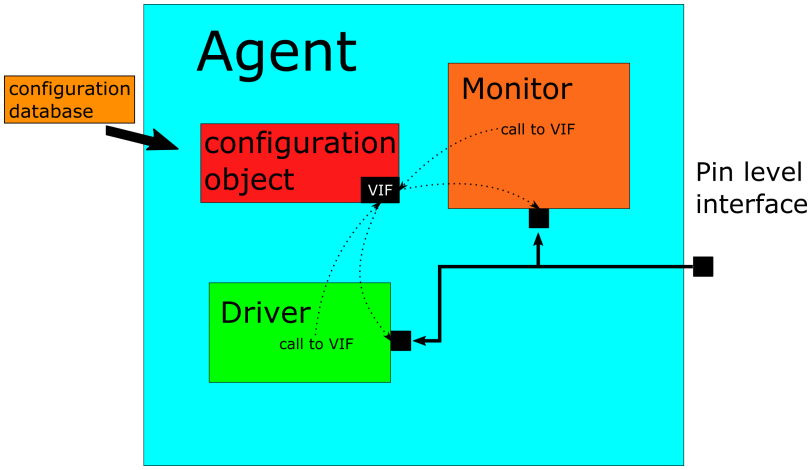


FIGURE 4.14: Handle to virtual interface changed

Instead of passing the handle to the virtual interface from the configuration object to the monitor and driver, which is commonly done in UVM, the handle was not

passed, but utilised through hierarchical access. Figure 4.14 illustrates how the handle to the interface, that the configuration object receives from the configuration database, is used by the driver and monitor with a call, and not passed to the driver and monitor. The call to the virtual interface was done with **cfg.vif.***, instead of **vif.***. The final parameterisation for the agent therefore consist only of the item and the configuration object. The code for the generic class header with the declarations of generic components is shown below.

```
class gen_agt#(type config_type = int, type item_type = int) extends uvm_agent;

  `uvm_component_param_utils(gen_agt#(config_type,item_type))

  generic_sqr#(item_type)              sqr, sqr_cache;
  gen_drv#(config_type,item_type)      drv, drv_cache;
  gen_mon#(config_type, item_type)     mon, mon_cache;

  config_type                          cfg, cfg_cache;
```

It can be seen that the driver, monitor and agent use the same parameterisation, and the sequencer does utilise a configuration object. It is also possible to see that each of the sub-components are declared for the use with cache and without.

```
class irq_agt extends uvm_agent;
  `uvm_component_utils(irq_agt)

  irq_sqr                          sqr;
  gen_drv#(irq_config,irq_item)    drv;
  irq_mon                          mon;
```

The Irq protocol is different from the other protocols as it adds an extra analysis port. However, small alterations made it possible for the protocol to use generic components. As seen in the code above, which is taken from the Irq-agent, the generic driver is utilised and specified with the correct types. The monitor in this protocol was also altered, it was extended from the generic monitor, as seen below, and then added the extra analysis ports and task it uses to call the interface for the *convert2Item-*, and an extra added *convert2Item_seq-* function.

```
class irq_mon extends gen_mon#(irq_config, irq_item);
```

4.5 Adopting the UVC

The remaining agents started their adoption of the UVC. As the framework already was customised for the specialised agents, it was not possible to just swap the agent in the environment. In order for correct functionality with the rest of the framework several actions was made in order to make the protocols compatible:

- Environment
  - Declaring the agent
  - Creating the agent
- Add generic sequencer to the virtual sequencer
- If the configuration object was in the environment's configuration object, it had to be removed
- Add the protocols' configuration object
- Modifying the configuration in tests
  - Also necessary for extended tests
- Added name variable to configuration object, if this was not already done.
- Use/extend generic sequence item or add necessary variables to sequence item.
  - delay
  - gen_wait
  - response
- Changes to sequences
  - Change configuration

  - Use generalised instead of specialised names, i.e. bus_address, not pbus_address

- Changes in the interface
  - Move the functionality from the old driver and monitor to the interface, using the tasks discussed in this thesis.
  - Clocking block
  - Pull_req-bit
  - e_mon-event
- Remove old, unused components
  - Driver, monitor, sequencer and agent

The generic agent was adopted by most of the protocols. It is possible to see the declarations in the environment in the new framework in the code below. It is also included how one agent is created in the build phase.

```
class uenv extends uvm_env;
...
  gen_agt#(pbus_config, pbus_item)                    m_pbus_agt;
  gen_agt#(dbus_config, dbus_item)                    m_dbus_agt;
  gen_agt#(iobus_config, iobus_item)                  m_iobus_agt;
  gen_agt#(ocd_config, ocd_item)                      m_ocd_agt;
  gen_agt#(cpu_externals_config, cpu_externals_item)  m_cpu_externals_agt;

  irq_agt                                             m_irq_agt;
  iobus_qr_agt                                        m_iobus_qr_agt;

  uvsequencer                                         m_vsequencer;

    ...

   m_dbus_agt  = gen_agt#(dbus_config, dbus_item)
            ::type_id::create("m_dbus_agt", this);
```

The agents that have been included in the environment is declared and created with their specialised parameters. The configuration objects used in the agents

have been extended from the configuration template. The sequence items have either been extended from the generic sequence item or used its own specialised one. The generic sequencers that was declared in the virtual sequencer can be seen in the code given below.

```
class uvsequencer extends uvm_sequencer;
  `uvm_component_utils(uvsequencer)
  generic_sqr#(iobus_item)          m_iobus_sqr;
  generic_sqr#(pbus_item)           m_pbus_sqr;
  generic_sqr#(dbus_item)           m_dbus_sqr;
  generic_sqr#(ocd_item)            m_ocd_sqr;
  generic_sqr#(cpu_externals_item)  m_cpu_externals_sqr;
  irq_sqr                           m_irq_sqr;

  iobus_qr_sqr                      m_iobus_qr_sqr;
```

There are differences between the old framework, described in Chapter 3, compared to the new framework. For both the environment (Chapter 3.1.3) and virtual sequencer (Chapter 3.1.5), is the old specialised declarations for the agents and sequencers, swapped for a generic component that utilise parameters.

Chapter 5

Results

The results discovered in this project are presented in this chapter. It will be taken a look at how many lines of code there are in the frameworks, as it is a way to quantify the work that has been done. It can also be stated that with more code, there will be more bugs, which again can lead to more time debugging. As the Iobus_qr protocol has not been integrated under the new framework yet, it will be ignored in the results. Also, as there has been minimal changes to the sequences and no changes to the CPU_snapshot and scoreboards as these are outside of the scope of the project, these are also ignored.

Taking the initial framework and altering it to become independent of base classes, as described in Chapter 4.1, before developing the new framework, allows for a comparison between the two frameworks. As the framework that is independent of base classes, and all the classes are specialised, it is referred to as the specialised framework.

5.1 The Non-generic Code

Table 5.1 shows the amount of code lines in the top of the hierarchy in the specialised framework. Even though the top of the hierarchy is not a part of the verification component, it is included as there has been alterations to the rest of

the framework during the development of the UVC, like alterations to the configuration.

Component / file	**Lines of code**
config_template	46
test_base	70
test_base_active_agents	31
uvm_base_test	16
uenv	74
uenv_config	53
uvsequencer	18
Total	308

TABLE 5.1: Lines of code in the top of the hierarchy for the specialised framework

The total amount of code lines in the protocols are presented in Table 5.2. The amount of code-lines in the table is an aggregate from all the components and objects, as each of the protocols contain every file that is presented in the table.

Sub-component	**Lines of code**
Agents	211
Configuration objects	182
Drivers	236
Interfaces	105
Sequence items	221
Monitors	168
Sequencers	53
Total	1176

TABLE 5.2: Lines of code in the protocols, specialised framework

The total amount of files used for the protocols is 42. The amount of code lines in the protocols and the top of the hierarchy is 1484.

5.2 Generic Code

When the framework uses the UVC, the code is considered as generic. At the end of the previous chapter it was shown all the generic agents that had replaced the specialised agents. Table 5.3 presents the amount of code lines in the top of the hierarchy for the generic code. It also shows the difference of code lines compared

to the specialised framework. An increase in the amount of code lines from the specialised framwork's code is marked with red, while a decrease is marked with green. There has been a small overall increase in the amount of code lines in this part of the hierarchy. This is mainly coming from moving the protocol's configuration objects from the environment's configuration object, along with the need for extra lines of code used to access the configuration database.

Component / file	**Lines of code**	**Difference**
Config_template	46	0
test_base	92	22
test_base_active_agents	55	24
uvm_base_test	16	12
uenv	86	-12
uenv_config	33	0
uvsequencer	18	0
Total	346	38

TABLE 5.3: Lines of code in the top of hierarchy for the new framework

The amount of code lines for the components and objects that are developed to be generic, or extended from, is shown in Table 5.4. The new components removes the need for some of the old components. Table 5.5 presents the components that have been replaced with the generic. There is a total of 21 files that has been replaced with four generic files. The yellow cells marks the classes that has been extended from another class.

Generic files	**Lines of code**
Agent	52
Monitor	29
Driver	61
Sequencer	16
Sequence item	71
Total	229

TABLE 5.4: Lines of code in developed files

Table 5.6 presents the amount of lines of code for the protocols with the all the components, including the generic. As earlier is the cells with an reduction of code highlighted in green, and the one with an increase is highlighted with red. There is a overall a reduction in code lines, except for the interfaces. This is because a lot of the functionality that has previously been on the driver and monitor has been

Protocol	Agent	Monitor	Driver	Sequencer	Item
Pbus	X	X	X	X	-
Dbus	X	X	X	X	-
Iobus	X	X	X	X	-
OCD	X	X	X	X	
IRQ		-	X		
CPU_externals	X	X	X	X	

TABLE 5.5: Files removed

moved to the interface. The total amount of code lines for the whole framework is 1266, a reduction of 218.

Component/file	Lines of code	Difference in code lines
Agents	86	-125
Configuration objects	75	-107
Drivers	61	-175
Interfaces	446	341
Sequence items	158	-63
Monitors	64	-104
Sequencers	30	-23
Total	920	-256

TABLE 5.6: Number of code lines in the protocols, new framework

In order to use waveforms, the transactions had to be working. Figure 5.1 shows that the transactions in the waveform diagram is working. It is also possible to see the correct bus names for the transactions, as well as the "_cache" extension".

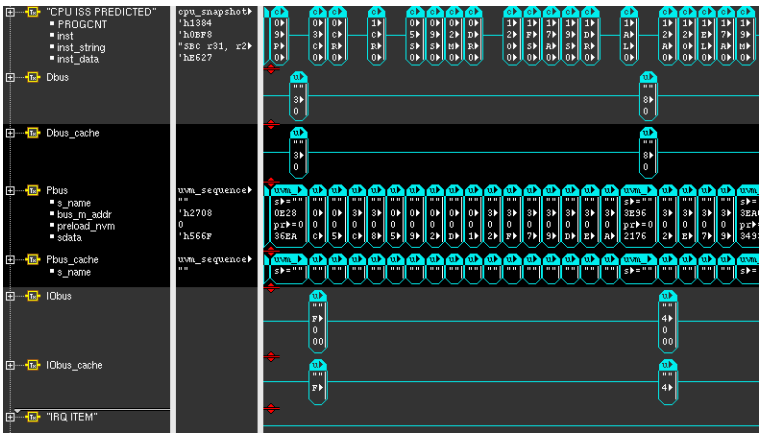


FIGURE 5.1: Transactions are registered

Chapter 6

Discussion

This chapter discuss the results from the previous chapter, before it will continue to discuss the use of parameters and the configuration that has been done. Advantages and disadvantages with using a UVC will also be discussed, before mentioning some inconveniences in the project. The end of the chapter will discuss different alternative solutions that could have been used in this project.

6.1 Code Reduction

The specialised framework's code had a total of 1484 lines of code in the framework without the files that has been omitted, where 1176 of these were from the protocols' components. The amount of code is spread quite evenly across the different components, with the exception of the sequencer, which naturally will contain less code. The amount of code in the agents could have been greater had the cache-functionality been added. The amount of code lines could have been even greater if the the template for the configuration objects had not been utilised for some of the protocols.

The generic code totalled to 1266 lines of code for the whole framework. The total reduction in code adds up to 218 lines of code, a 14.7% decrease. Excluding the higher levels from the equation, the decrease for the total amount of code in the protocols adds up to 256 lines, a 21.8% decrease. This is a substantial amount

of decrease, and suggest that the generic code may help reduce the time spent creating the components.

Even though there is a overall decrease in the amount of code lines in the protocols, the interfaces stand out. The interfaces had a total increase of 341 code lines, a stunning 325%. There are several reasons for this enormous increase in the interfaces. The first reason is that most of the specialised functionality from the driver and monitor was moved here, which naturally increases the amount of code required to implement the interface. Additionally was the *generate_swait_bwait* function that previously was contained in the configuration objects, moved to the interface. When most of the functionality that is specialised for a protocol is moved to the interface, it may reduce the amount of time used to probe the framework looking for code. This may further decrease of the total time spent in verification, which again can lead to more time spent on quality instead of development and debugging.

The code increase in the top of the framework does indicate that there is a little more configuration in the new framework. It is mostly from moving the configuration object from the environment's configuration object, which hides a lot of the overhead code that comes when using the get- and set- functions for the configuration database. Moving the configuration objects does, however, reduce the hierarchy, and make the access to the individual objects easier with the new framework. The advantage of using the hierarchical methodology is that setting the configuration for several of the configuration object is easier, which could help the verification engineer when creating the tests.

In addition to the decrease in code, there is also a huge reduction in the amount of files used in the new framework, with the four generic files replacing 21 files from the old framework. The sequence item that was created is also extended by three of the protocols. Even though there are a lot of the same constraints used in the items, the items are extended and not parameterised with the constraints because it makes for an easier implementation in the rest of the hierarchy, in addition to the fact that there might be additional or different constraints for other protocols. The reduction of the amount of files necessary in the framework, shows that there is a high reusability of the components. This is also seen in the possibility to use the same agent, with only specifying the type of configuration and sequence item.

It can also be seen as it is possible to use the other components in another agent, or that it is possible to inherit from them.

6.2 Parameterisation and Object Oriented Programming

The use of parameterisation and OOP in this project has primary been in order to enable reuse and decreasing the copy-paste code in the framework. Using parameterisation can be considered as a dual-edge sword, as it is quite complex to implement and the code base can become more verbose. In this project the parameterisation has mainly been used to specify the data type utilised throughout the verification component. However, parameterisation is also used to specify the size of parameters in the generic sequence item. This makes it possible to use the generic sequence item directly, and if there are constraints or other functionality needed, it could be extended with the parameters specified in the class declaration.

Parameterising the agent with a configuration object and the sequence item, makes it easy to implement, as the agents configuration object and sequence item names should follow naming conventions. The parameters are propagated throughout the hierarchy, and used in the sub-components. It is the use of parameterisation and correct configuration that enabled the opportunity to develop the Universal Verification Component in this project. Chapter 4.4.10 described how the handle to the virtual interface was moved from the monitor and driver, to the configuration object, which is set from the test with the configuration database. This reduces the amount of parameterisation necessary, but still keep the same functionality as if they would be called from the monitor and driver. The extra hierarchical step to call functions, does not make much difference, as the components ideally should only need be instantiated in the environment, and the specialisation added in the interface.

6.3 Configuration

Correct configuration is a crucial part in UVM and in this project. The use of the configuration database has been utilised in great extent in order to achieve

correct configuration for the different protocols. In the development process this was one big obstacle that the entire project needed to function correctly. Since the environment use the same agent in order to build different protocols, the method of setting and getting the correct configuration was modified. The use of wildcards and the agent's handles, is a solution that relies on the verification engineer that creates the tests to have control over the names of the handles for the different protocols. Nonetheless, this should not be a problem, with the correct usage of naming conventions. Despite this, there is still not thought of a method for correct passing of configuration if there should be more than the standard configuration, that consist of the non-cache and cache setup. This might, however, be solved by an alteration to the *uvm_config_db*'s **field_name**.

6.4 Advantages and Disadvantages

The framework that has been developed has its positive and negative sides to it. On a positive side it is a framework that can work for many protocols, but on the other side it might reduce the degrees of freedom for the functionality in the interfaces.

The framework that has been created have several advantages compared to the previous framework. The first is the reduction in code, and with less code there will be less debugging. It is also less files that is needed to be considered, and working with less files will decrease the places problems and bugs may arise. As mentioned in Chapter 1.1, the total time verification engineers spend developing the verification environment and debugging in a project can be over 60%. Utilising a UVC where the code is bug-free, will ultimately reduce the total amount of code and where bugs may arise in the framework, which can significantly reduce the time developing and debugging. The time saved may contribute to that the deadline is kept, and it can be used in further development of tests for better coverage. The UVC that has been developed should also be capable of being used for other DUTs, as most of the specialised functionality is moved to the interface, which has to be implemented by the verification engineer. Another advantage is that every component has been created like standard UVM, which makes it possible to use functionality like the factory as described in Chapter 2.1.14.

A disadvantage is that the UVC relies on that the code written by the user is compatible with the rest of the framework. The user also needs knowledge about how the configuration object is built, and how to set the configuration from the configuration database to the correct protocol. Another drawback with the framework might be that if there is needed more complex setup, it is either needed to extend the components, or create new. This would, however, also have been the case in another framework. As the framework is new and has not been tested on other devices there might still be some small alterations that can be made.

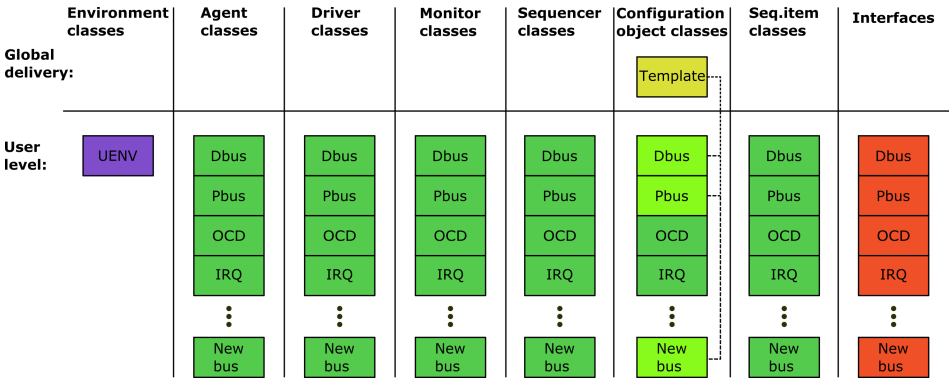


FIGURE 6.1: Class overview for specified framework

Figure 6.1 illustrates an example of what should be included in the specified framework, which was used as a base in the project, in order for it to be usable. As seen must every class and interface needed for a functional protocol be implemented by the user. There is one template that is given for the configuration object classes, that can be extended when creating a new protocol. All the other classes has to be created from scratch using this framework.

Figure 6.2 illustrates the same as the previous figure, but with a framework that utilises the UVC. Here the generic files and templates will be handed down from the UVC, and is therefore considered as a global delivery. The user still has to create the environment, where he can instantiate the generic components directly when creating new protocols. When adding a new protocol, the verification engineer can use the generic classes, extend them, or create new classes from scratch. The configuration object should be extended from the template.

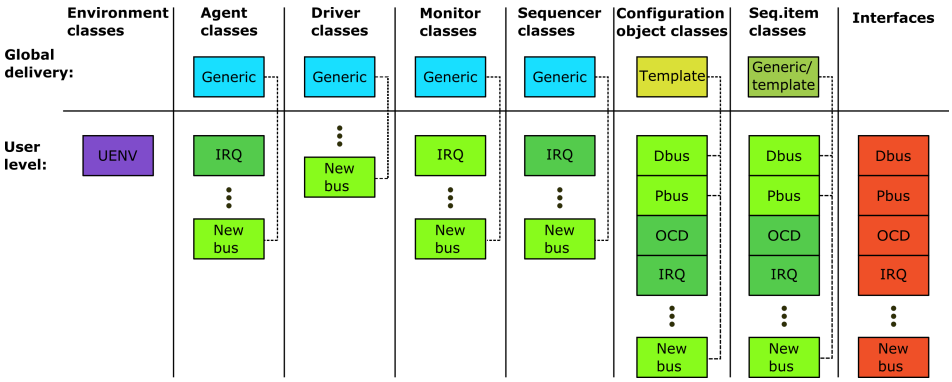


FIGURE 6.2: Class overview for framework with UVC

6.5 The Inconveniences

The project of creating a Universal Verification Component for CPU verification started with altering the existing framework, which utilised some inheritance, to a framework that did not utilise it to the same extent. The components in this framework were all specialised for every single protocol. This gave the opportunity to have a base, of which the new framework could be built on. It also made for a good comparison between two. One which was specialised and has the need to create components for every protocol, and one which is more complex, but where the components could be used for most of the protocols. However, there were some inconveniences with this.

6.5.1 Incomplete code

Chapter 4.4.7 described about the extension of the functionality when the cache is enabled. This is one example from the initial framework that suggest that the initial framework was not complete, and had room for improvement. Since the initial framework might have had other flaws, they may also have propagated to the new framework, as functionality has been moved. With the expanding of the agent functionality regarding the cache, also means that the verification engineer that will be continuing with the framework needs to be aware of such functionality and set the correct configuration when creating the tests.

6.5.2 Lost in Translation

One inconvenience in the project was that a lot of the functionality of the components has been moved around, back and forth to the interface. Thus the function-names that has been used has lost some of its initial meaning, which could make it harder for the verification engineer to understand the code.

6.5.3 The Iobus_qr Protocol

As previously mentioned, was the code lines for the Iobus_qr omitted in the results. This was done to create a better image of how the UVC will impact the framework, as the Iobus_qr protocol was out of the scope of the project, and has yet to be altered. There are some things that should be taken a look at when adapting this protocol. The agent in the initial framework creates a total of four configuration files, and pass some of these on to one monitor. One way of utilising the generic components could be to split the functionality of this protocol into two different protocols, creating two monitors and drivers. A problem that might occur is that they access the same interface, and could cause a clash in configuration and accesses. Another way might be to do the same as the Irq protocol and utilise the inheritance to extend functionality for the components. There is also need to change the configuration depending on the solution that is chosen.

6.5.4 The Irq Protocol

The Irq protocol, as described in Chapter 4.4.10, did not have its agent, monitor and sequencer replaced. This is because of the irregularity in the protocol, where it creates additional analysis ports. The new framework utilises inheritance in order to handle irregularities in the functionality. This does, however, show that even though the component is used, it is still reusable by inheritance.

6.6 Alternative Framework Setup

It might be possible to solve the project using different methods than the ones that has been utilised in this thesis. One method might be a continuation of the initial

framework, that extend the the agent and sub-components to add the specialised functionality. This method simplifies the retrieval of configuration. It might, nonetheless, create the need for the user to re-write code, or write almost similar code for the various protocols. Extending from a base class could, nevertheless, reduce the amount of parameterisation that is used throughout the hierarchy, as it would rely on a heavier OOP-approach. It might also reduce the amount of code in the framework, but there will still be the same amount of files, and if there is not much functionality in the component the reduction of code might be sub-optimal.

It could also be possible to have another method of communication between the driver and monitor. Instead of the verification component making calls to the interface, the use of naming events could be used. The tasks in the UVC will then only be run if an event is triggered in the interface. If communication from the interface relies solely on events, clocking blocks might not be utilised. As described in Chapter 2.2.3 does the clocking block guarantee race free communication between the DUT and verification environment, and the use of events might also not be optimal.

To further improve the generic sequence item, it could be possible to add other variables to it. Having a sequence item with a lot of variables etc. on it would however also imply that there would be a lot of redundant code for some protocols. It could also be possible to use very generic variable names, but this could also cause for more confusion with the user, and therefore work against the purpose of saving time.

Chapter 7

Conclusion

This thesis has described the development of a Universal Verification Component for CPU UVM verification. The agent that has been developed contain a generic driver, monitor and sequencer. The UVC has been created by moving functionality from the drivers and monitors to the interface. The driver and monitor that has been developed use generic task-calls to the interface to retrieve the specialised functionality for the protocol. To include the agent in an environment, it has to be parameterised with the protocol's configuration object and sequence item. It is also necessary to set the configuration for specific protocols as described in this thesis. The UVC has been created to work on two CPUs, with and without cache, at the same time. There is a bit in that can be set in the configuration object that decides if it should create components for it. A sequence item has also been developed, which can be used directly or extended to add additional functionality.

The new framework that has been developed reduces the total amount of code needed to implement the protocols in the framework with 21.8%. It also reduced the total amount of files used for the protocols from 42 down to 25. The decrease in code and files may lead to less time debugging and development of a testbench in the future, which means more time can be used improving tests and get better coverage of the device. UVM promotes reuse, and the developed universal verification component is an excellent example of how one component can be reused and decrease the development and debug time of a testbench in a project.

7.1 Future Work

Despite good results on the protocols that has been added to the framework, there is still one protocol that needs to be included in the framework, as mentioned in the discussion. The UVC should also be tested on other CPUs, and be tested further for bugs. In order to speed up verification, the current framework can be modified to support hardware emulation by introducing a dual top approach, as described in Chapter 2.2.2. It is possible with a further reduction of signals between the UVC and the interface, and emulation may be the future in verification as the DUTs increase in size and complexity.

Bibliography

[1] Wilson Research Group. Functional Verification Study. 2016. URL https://blogs.mentor.com/verificationhorizons/blog/2016/08/08/prologue-the-2016-wilson-research-group-functional-verification-study/.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Number 2 in Addison-Wesley Professional Computing Series. Addison-Wesley, 37 edition, 1994. ISBN 0-201-63361-2.

[3] Mentor Graphics' Verification Methodology Team. Uvm cookbook. *UVM Cookbook*. URL https://verificationacademy.com/cookbook/uvm.

[4] Hans van der Schoot and Anoop Saha. UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up. *DVCon*, 2015.

[5] ANOOP SAHA. From Simulation to Emulation A Fully Reusable UVM Framework. Technical Report TECH12100-w, Mentor Graphics, 7 2014.

[6] Young-Nam Yun, Jae-Beom Kim, Nam do Kim, and Byeong Min. Beyond UVM for practical SoC verification. *IEEE-978-1-4577-0711-7*, pages 158–162, 2011.

[7] Mark Litterick. Pragmatic Verification Reuse in a Vertical World. 2013.

[8] IEEE Standards Dictionary: Glossary of Terms Definitions (CDROM). *IEEE Standards Dictionary: Glossary of Terms Definitions (CDROM)*, .

[9] Accellera Systems Initiative Inc. Universal Verification Methodology (UVM) 1.2 User's Guide. October 2015. URL http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[10] IEEE Standard for SystemVerilogUnified Hardware Design, Specification, and Verification Language. pages 1–1315, . doi: 10.1109/IEEESTD.2013.6469140.

[11] Khaled Salah Mohamed. *IP Cores Design from Specifications to Production Modeling, Verification, Optimization, and Protection*. Springer International Publishing : Imprint : Springer. ISBN 978-3-319-22035-2 978-3-319-22034-5. OCLC: 968791013.

[12] VMM Central. Uvm_sqr_if_base #(REQ,RSP). Accessed 2018-05-01 . URL `https://www.vmmcentral.org/uvm_vmm_ik/files3/tlm1/sqr_ifs-svh.html#uvm_sqr_if_base%23%28REQ,RSP%29.get_next_item`.

[13] Clifford E Cummings. UVM Transactions - Definitions, Methods and Usage. *SNUG*, 2014.

[14] Accellera Systems Initiative Inc. Universal Verification Methodology (UVM) 1.2 Class Reference. June 2014. URL `http://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf`.

[15] Clifford E Cummings. The OVM/UVM Factory & Factory Overrides How They Work - Why They Are Important. *SNUG*, 2012.

[16] John Aynsley. Easier UVM for Functional Verification by Mainstream Users. 2011.

[17] "Generic". accessed 2018-05-01. URL `https://www.merriam-webster.com/dictionary/generic`.

[18] David Rich and Adam Erickson. Using Parameterized Classes and Factories: The Yin and Yang of Object-Oriented Verification. Technical Report TECH8190-w, Mentor Graphics, 4 2009.

[19] Bryan Ramirez and Michael Horn. Parameters and OVM Cant They Just Get Along? *DVCon*, 2011.

[20] Mark Glasser and Santa Clara. Conguration in UVM: The Missing Manual. *DVCon India*, 2014.

[21] Jonathan Bromley. Slicing Through the UVMs Red Tape. *DVCon Europe*, 2016.

[22] Mark Glasser. Advanced Testbench Conguration with Resources. Technical Report TECH9850-w, Mentor Graphics, 3 2011.

[23] Vanessa R. Cooper and Paul Marriott. Demystifying the UVM Configuration Database. *DVCon*, 2014.

[24] Hierarchal Testbench Configuration Using uvm_config_db. Technical Report AP.CS3989, Synopsis, June 2014.

[25] Comprehensive UVM/OVM Acceleration. Technical Report 21501, Cadence Design Systems, Inc., 12 2011.

[26] Hans van der Schoot, Anoop Saha, Ankit Garg, and Krishnamurthy Suresh. Off To The Races With Your Accelerated SystemVerilog Testbench. *DVCon*, 2011.

[27] Standard Co-Emulation Modeling Interface(SCE-MI) Reference Manual. Technical Report 2.4, Accellera Systems Initiative Inc., November 2016. URL `http://www.accellera.org/images/downloads/standards/sce-mi/SCE-MI_v24-Nov2016.pdf`.

[28] Clifford E. Cummings and Arturo Salz. System Verilog Event Regions, Race Aviodance & Guidelines. *SNUG Boston*, 2006.

[29] Stuart Sutherland and Tom Fitzpatrick. UVM Rapid Adoption: A Practical Subset of UVM. *DVCon*, 2015.

[30] Marcio F S Oliveira, Christoph Kuznik, Wolfgang Mueller, Wolfgang Ecker, and Volkan Esen. A SystemC Library for Advanced TLM Verification. 2012.