Markus Ortoff BSc

Development of a Reusable Constrained Random Verification Environment for Lightweight Serial Protocol Handlers

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing.Dr.techn. Peter Söser

Institute of Electronics

Guillermo Conde, PhD.
Infineon Technologies Austria AG, Design Center Villach

Graz, December of 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

In today's continuously increasing complexity of integrated circuit designs, functional hardware verification is a must-have sign-off procedure in pre-silicon verification. This thesis focuses on the functional constrained random verification of the digital part of a lightweight serial protocol handler and also provides insight on how to set up, respectively, develop an appropriate constrained random verification environment. The work itself is divided into seven chapters. The first one introduces functional hardware verification in general, the traditional approach as well as the used methodology. This is followed by the second chapter, which contains an explanation regarding lightweight serial communication and protocols, their differences as well as an example of a lightweight serial protocol. The third chapter introduces significant object-oriented SystemVerilog aspects and its very important role in functional hardware verification. The follow-up chapter covers the constrained random environment's development, building blocks and architecture. Chapter five addresses aspects of the design-under-verification. The sixth chapter elaborates how to set up a regression suite and furthermore, how to generate a tool-supported verification report; this final chapter is followed by the conclusion.

Keywords:

Constrained Random, SystemVerilog, Transaction Level, Lightweight Serial Protocols

Kurzfassung

Mit der ständig wachsenden Komplexitätszunahme und der damit verbundenen Erhöhung der Integrationsdichte ist funktionelle Hardware Verifikation ein sogenanntes finales Abnahmeverfahren in der Präsilizium Verifikation. Diese Arbeit beschäftigt sich mit funktioneller Hardware Verifikation, anhand der Entwicklung einer wiederverwendbaren 'Constrained Random' Verifikationsumgebung für leichtgewichtige, digitale, serielle Protokollschnittstellen. Die Arbeit ist in sieben Kapitel unterteilt. In der Einleitung wird das bisherige, traditionelle Verfahren kurz erläutert und mit dem aktuellen Stand der Technik anhand der verwendeten Methodik verglichen. Im folgenden Kapitel wird auf leichtgewichtige, serielle Kommunikation und Protokolle eingegangen sowie ein Beispiel eines leichtgewichtigen, seriellen Protokolls präsentiert. Im darauffolgenden, dritten Kapitel werden die Aspekte der Design- und Verifikationssprache System Verilog näher beleuchtet und speziell die verwendeten Features beschrieben. Im vierten Kapitel wird die Architektur der Verifikationsumgebung erläutert und näher auf die Zusammensetzung der Komponenten eingangen. Im fünften Kapitel wird das Design-unter-Verifikation kurz vorgestellt. Das sechste Kapitel erläutert das Thema Regression-Testing sowie die toolunterstützte Berichterstellung, gefolgt von der abschliessenden Schlussfolgerung.

Contents

List of Figures

Chapter 1

Introduction

1.1 Functional Hardware Verification

Functional Hardware Verification can be described as a process of verifying if a Design-Under-Verification (DUV) conforms to its specification, primarily in terms of functionality. Therefore, various languages, techniques, methodologies, tools and flows exist on the market and can be used as well as utilised to complete this process. In terms of languages, SystemVerilog (SV) with its object-oriented programming (OOP) capabilities has become one of the major supporting technologies in Functional Hardware Verification.

Nowadays, the most common verification methodologies as stated in Mehta [1] are:
Universal Verification Methodology (UVM), Unified Power Format (UPF), Analog/ Mixed Signal (AMS), SystemVerilog Assertions (SVA), SystemVerilog Coverage (SFC), Coverage-Driven Verification (CDV) and Constrained Random Verification (CRV), Static Timing Analysis (STA), Clock Domain Crossing (CDC), Logic Equivalence Checking (LEC), etc.

This thesis focuses on constrained random, respectively, coverage-driven verification by applying concepts of object-oriented programming in SystemVerilog, SystemVerilog Assertions and SystemVerilog Functional Coverage as well as regression testing.

The following section elaborates the traditional verification approach.

[1] ASIC/SoC Functional Design Verification [Mehta, 2017]

1.2 Traditional Directed Test Approach

The traditional directed test approach of verifying the correctness of a design in terms of functionality, as simplified in Figure 1.1, is to choose a requirement and write a test to verify it. After the test is finished, debugged and simulated, the resulting waveforms have to be checked for correctness. This process has to be repeated until all features, respectively, requirements are verified.

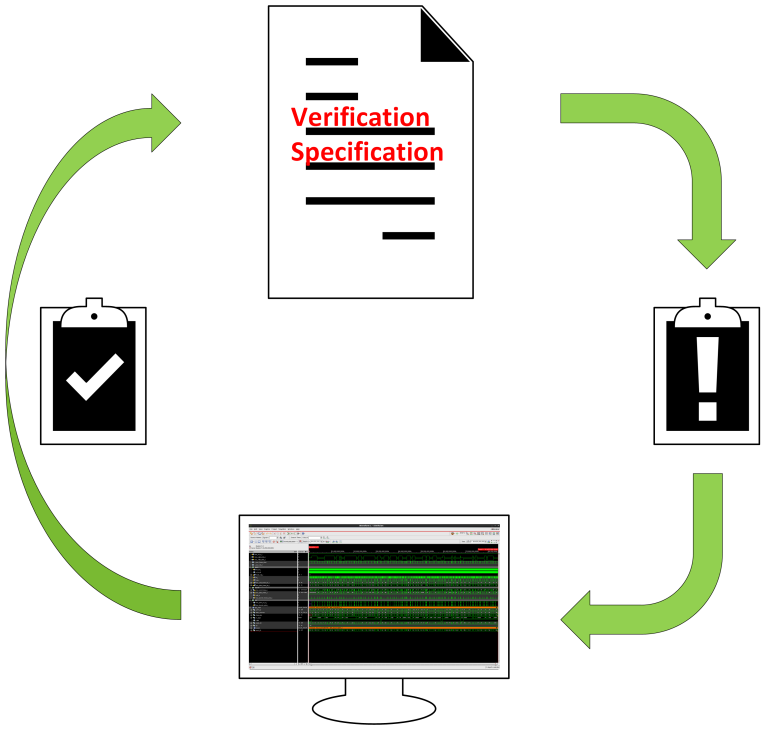

Figure 1.1: Pick & Write, Verify and Repeat.

In order to stimulate the design during simulation, it is always necessary to develop a test bench. Typically, this environment is written in the same Hardware Description Language (HDL) as the design is implemented in and the simulation is then carried out with a script-based and directed test flow. Since the complexity of designs continuously enhances and HDLs are not designed to support the verification/ validation process, the traditional directed test approach involves writing hundreds of test cases (even in a small design) to verify all features required.

In sum, the process of verification consumes approximately 60 to 70 percent of the time

during the development cycle of a module, for instance, verification of a design with e.g. three configurable inputs. Input A consists of 508 individual configurations, input B of 64 and input C of only 4. If these configurations and all possible combinations shall be verified, it requires the engineer to write approximately 128.000 directed tests.

In fact, this traditional approach is needed in the beginning of design implementation to feedback results regarding the basic functionality, however, this should not be the only methodology or technique applied. Since HDLs are actually not designed for verification, it is clear that this methodology entails various disadvantages, such as:

- Lack of metrics and regression mechanism
- Only predicted behaviour is tested
- Very limited re-usability
- Extremely time-consuming
- Requires deep knowledge of the design
- Difficult to maintain

Consequently, this custom-flow does not support the process of extracting any coverage information, thus, it is difficult to provide relevant metrics and reflect the verification's status; this can result in mismatches between requirements and tests.

The traditional test bench is quite specific for a given design and mainly requires great effort to be adjusted to different projects. Even if one has managed to create a certain script-based verification and simulation flow, the status cannot be reflected without measurable metrics.

Writing test cases or scenarios with the traditional approach further requires deep knowledge of the design in terms of timing and signalling. One has to create a stimuli file with every individual signal transition described by simulation times when to assert or de-assert a specific signal. Furthermore, waveforms have to be checked manually for correctness, however, this process also implies that this approach is very time consuming, hard to maintain and prone to errors.

Actually, using directed tests only, critical scenarios and corner cases can be overlooked; especially scenarios the engineer does not explicitly think of. Undoubtedly, the directed test approach is sufficient to verify the design-under-test (DUT) basic functionality, but to ensure correctness, a more sophisticated and automated approach needs to be applied.

1.3 The Need for Automation

SystemVerilog, referred to as the "Unified Hardware Design, Specification, and Verification Language"[2], was developed to combine design and verification capabilities to one single language and an important aspect concerning the DUV that it can be designed with any of the major HDLs but can be examined with the OOP capabilities of SV.

SystemVerilog combines the capability of building a virtual hardware setup (e.g. tester in a lab) around a given micro architectural design and furthermore, allows to control the DUV with the capability of object-oriented software automation, as proposed by Doulos [Doulos, 2012]. This results in a flexible and reusable environment, especially when it comes to regression and metrics extraction.

Once one is familiar with the architecture (example explained in Chapter 4) of such an environment, it is possible to create, extend and alter tests and scenarios to specific needs. Since SystemVerilog is supported by all big EDA vendors such as Cadence, Mentor & Synopsys, test benches implemented in SystemVerilog can be integrated into different projects. However, the re-usability is not seamless and requires some fine-tuning effort.

The usage of tools and regression-support enables the verification engineer to reflect the status of the verification process by cross-referencing different forms of coverage metrics, such as functional coverage (including formal assertion coverage) and code coverage to their corresponding requirements. After the process of planning and cross-referencing is done, it is possible to generate a suitable report concerning the work done, as explained in Chapter 6.

Industry 4.0 focuses on automation since applying automation wherever and whenever

[2] Language Reference Manual [IEEE, 2013]

possible safes time. Indeed, one can save a vast amount of time with automated processes, but first they have to be automated. This particular issue is today's biggest challenge, it appears to be merely impossible to verify complex designs to a degree of 100%, therefore it is always a trade-off between verification and documentation.

The key to test bench automation is randomly generated stimuli, which then is applied to the DUV inputs. Furthermore, the test bench has to be able to apply the stimulus to the design on the input side automatically and obviously, to collect the information generated by the DUV at the output in order to check the result against a reference model. In case of a correct result, it shall be stored in a coverage database. If the resulting output information is not correct, it shall be discarded and reported. In such an event, the engineer has to analyse the test regarding bugs in the DUV or a failure originating from the environment.

1.4 Constrained Random Verification

This methodology can be implemented in SystemVerilog in a way that the resulting environment utilises the object-oriented programming (OOP) capability offered by the language. Basic important features are explained in more detail in the corresponding Chapter 3 SystemVerilog.

Constrained Random, respectively, Coverage-Driven Verification is built upon the three "Cs"[3]: Correctness, Completeness and Constraints. First, the output produced by the DUV has to be checked automatically for correctness. Second, completeness is measured and reflected by collecting coverage information with the aid of SystemVerilog Functional Coverage. Last, constraints have to be implemented in order to increase the coverage and probability to detect corner cases.

[3] As stated by John Aynsley in the video Introduction to UVM. The video is available at https://www.doulos.com/knowhow/sysverilog/uvm/, (accessed 7 December 2017)

1.4.1 Correctness - Automatic Checkers

As described in Section 1.3, today everything needs to be automated and therefore, the checkers need to be autonomous as well. In other words, a test scenario created to verify certain features has to be developed and debugged. Debugging in this stage of the development cycle, is also done by visual inspection of the waveforms. Once the test scenario is up and runs properly, there is no need for manual debugging or checking and one can concentrate on the verification's completeness.

1.4.2 Completeness - Coverage

As soon as some scenarios are developed, the process of implementing coverage for measuring the status of the verification can be triggered. The "Coverage Cookbook", published by Mentor Graphics Corporation [Mentor Graphics, 2012], provides a detailed introduction of how to tackle different kinds of coverage metrics, their measurement, analysis as well as how to keep track of the overall verification progress. As described in "Coverage Cookbook", one single metric does not enable to characterise the verification sufficiently. That is why coverage is further broken down into its main overall terms for classification of coverage metrics:

- Code coverage,
- Functional coverage
- and Assertion coverage.

Code Coverage

Code coverage, which originates from software testing in the early 1960s by Miller and Maloney [Miller and Maloney, 1963], holds the advantage to be simply enabled via some command line switches and is a reliable indicator to highlight how much of the design is exercised by a certain scenario and additionally, reflects the quality of the stimuli. The goal is to reach 100% of code coverage, which might not always be possible, therefore, one can

exclude certain untouchable parts, leaving a comment with explanation for documentation purpose.

To measure and analyse code coverage in more detail, it is broken down into the following single metrics such as: toggle, line, statement, block, branch, expression and finite state machine coverage. Most of these metrics are self-explanatory and for further details and explanation "Coverage Cookbook" by Mentor Graphics Corporation [Mentor Graphics, 2012] is recommended.

Code coverage only indicates parts of a design that have already been activated. It does not provide any information regarding correct functionality of the design itself. As a consequence, a second metric needs to be implemented, which is referred to as functional coverage.

Functional Coverage

Functional coverage can be recorded through SystemVerilog Assertions and Covergroups, see Sections 3.7 and 3.8. In comparison to code coverage, functional coverage involves great effort and has to be planned and implemented carefully by the verification team. The verification manager is in charge of creating a verification plan, which shall include all requirements that need verification. This is done by thoroughly translating the specification of a design into requirements. The initial plan with all requirements found is then implemented by the verification team. The translation of all requirements into functional coverage points, sequences and/or properties results in the so-called coverage model. In the process of verification, the coverage model is continuously refined to the extend of being satisfiable. With the coverage model finished and enough output information generated, the verification team is then able to close the verification process by generating a detailed report concerning the verification.

Unquestionably, it is not sufficient to reach 100% functional coverage with, for instance, code coverage at approximately 80%. In such a case might be functional coverage missing. The goal is to reach 100% of satisfiable functional coverage where all parts of the design have been exercised and all planned coverage items have been verified to be correct, regarding

their specification.

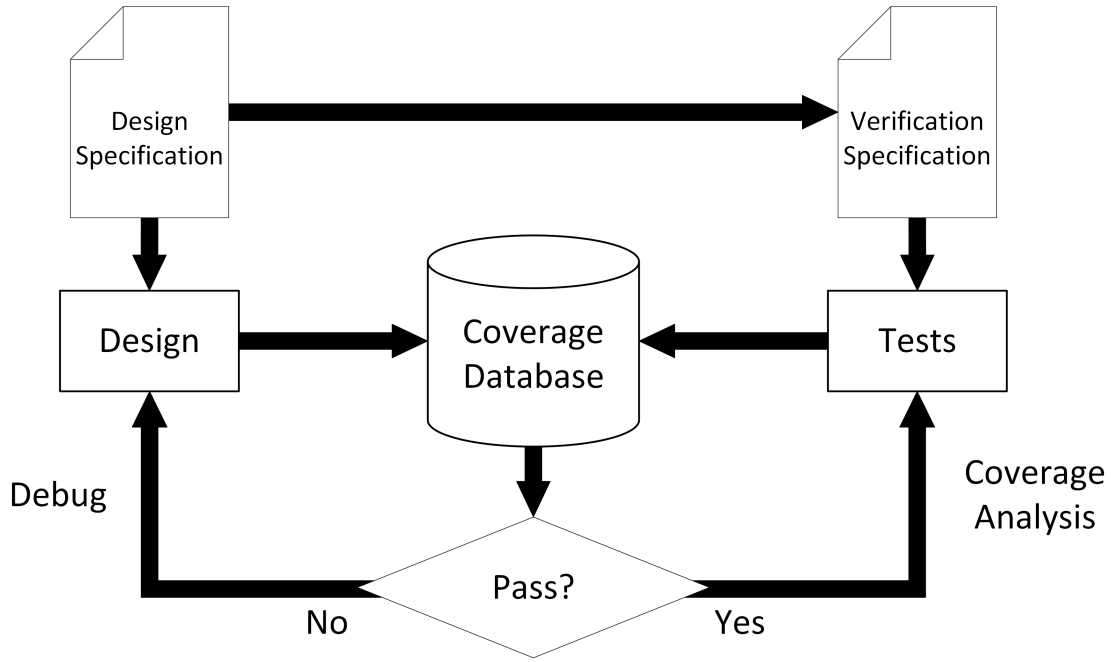

Figure 1.2: Coverage Refinement Flow [Spear, 2010].

1.4.3 Constraints - Increasing Coverage

Constraints provide a possibility to shape random input stimulus, thus, reaches interesting corner cases much faster. The SystemVerilog built-in constraint solver analyses the code of specific classes and is then able to solve constraints. The stimulus itself is still random, but the statistical distribution of the solution space vector is based on the previously solved constraints as explained by Doulos [Doulos, 2012]. Since this process is done by the environment on-the-fly and automatically, the test bench simply has to produce numerous input stimuli by allowing longer simulation runs. In case the coverage happens to be saturated and does not increase significantly, one option is to re-seed the random number generator and restart the simulation. Another possible option is to alter some constraints by either tightening or releasing them.

1.5 Abstracting Complexity with Transactions

Profound knowledge concerning the design's functional intent, which must be acquired by reading the design specifications of a device carefully, enables the verification team to translate the purpose and implement corresponding transactions. The latter can be seen as an atomic operation executed on the DUV, similar to transmitting a serial data frame. Moreover, these transactions serve as a control structure. Information stored in a transaction can be data, message length, error injection flags and even functions, e.g. to calculate parity or similar tasks needed. Furthermore, such information is used by various parts of the test bench in order to fulfil their duty, for instance, stimulating (driver) inputs as well as collecting (monitor) and predicting (checker) outputs.

The following chapter serves as a brief introduction concerning lightweight serial communication and protocols.

Chapter 2

Lightweight Serial Communication and Protocols

This chapter covers the principle of serial communication, followed by a brief introduction regarding serial protocols and ends with an explanation of a lightweight serial protocol on the example of the MicroSecond Channel (MSC) [IPExtreme®, 2007] protocol developed by Infineon Technologies AG [Kelling et al., 2005] in 2005.

2.1 Serial Communication

In serial communication, data is sent sequentially over a channel or bus whereas in parallel communication, all data bits are sent simultaneously. Therefore, in serial communication less pins are required, which makes it very efficient for communication inside integrated circuits as well as to peripherals. Data transmission is controlled by a master device and underlies the principles of Master-Slave-Communication. In general, it has to be distinguished between two and three, respectively, four wire communication.

Two wire communication operates with only two signal lines: Serial Data (SDA) and Serial Clock (SCLK) line; a slave is addressed via address bits contained inside the frame transmitted. In comparison, three, respectively, four wire communication includes addi-

tional signal lines. Both types share the Serial Clock (SCLK) line although the data line is separated into Serial Data In (SDI) and Serial Data Out (SDO). Such a separation has the advantage to enable simultaneous upstream as well as downstream communication referred to as full-duplex communication. A four wire communication can be distinguished by an additional Chip Select (CS) line, which is used to enable communication by selecting a certain slave device directly via an input pin.

In synchronous serial communication, data is sent with a single clock source, whereas in asynchronous serial communication, data is sent from a different clock domain. Therefore, in asynchronous serial communication it is necessary to provide a mechanism to acknowledge the reception of data or a transaction in general. Such a mechanism is called a handshake and is defined in the underlying communication protocol specification.

2.2 Communication Protocol

> "A communication protocol is a behavior convention that defines the temporal order of the interactions between the peer entities as well as the format (syntax and semantics) of the messages exchanged."[1]

In other words, a communication protocol is a specified way of how two entities are able to exchange information in a defined manner. Time sequence diagrams are created in order to represent the protocol interactions in a graphical way such as in Figure 2.1, a successful and a rejected connection set up.

Moreover, the complexity of a protocol is defined by various characteristics and is written down in the protocol specification document. If the protocol is standardised, the specification is released, for instance, by the International Organization for Standardization or similar entities in charge of standardisation processes.

There are numerous standard protocols and some of them are very similar when observed from the protocol point of view whereas others are contrary and not comparable at all. Simpler protocols such as SPI, I2C and the MSC protocol represent sufficient examples

[1] Protocol Engineering p.30 [Knig, 2012]

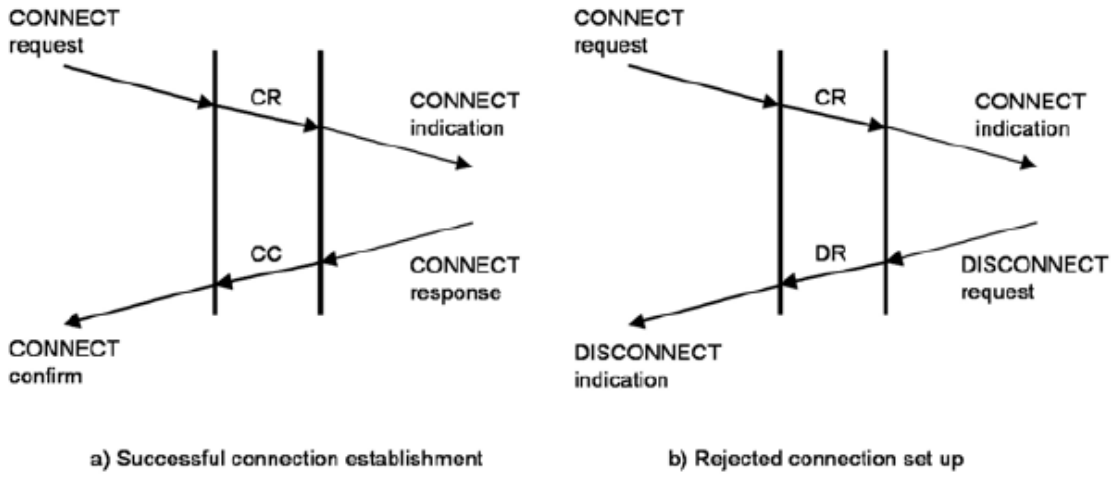

Figure 2.1: Protocol Sequences as Time Sequence Diagrams [Knig, 2012].

of lightweight serial protocols. Whereas, for instance, CAN, LIN and FlexRay are heavy-weight serial protocols in terms of complexity and protocol overhead.

The following section explains the MicroSecond Channel (MSC) protocol, as an example for a lightweight serial protocol.

2.3 A Lightweight Serial Protocol

The MicroSecond Channel (MSC) protocol [IPExtreme®, 2007] is a single-master/multi-slave asymmetric serial protocol developed by Infineon Technologies AG [Kelling et al., 2005] and conceived for connecting a slave (e.g. peripheral device) to a master (e.g. a microcontroller) via serial link. In this protocol, the master-to-slave communication (downstream) is synchronous serial-to-parallel, whereas the communication from slave-to-master (upstream) is asynchronous parallel-to-serial. In addition, full-duplex communication is supported by the MSC standard, which is explained in the following subsections.

2.3.1 Master-to-Slave Downstream Communication

As shown in Figure 2.2 , the serial input SI is synchronised with the rising edge of the FCL clock and sampled by the slave with the falling edge of FCL clock. The chip-select SSY is active low and synchronised with the rising edge of an FCL clock. The active phase of a downstream frame transmission starts with the falling edge of SSY and ends with its rising edge. During the passive phase, SSY is high and data at SI is ignored by the slave. In downstream communication, two types of frames can be distinguished, depending on the start bit of the serial data in SI, the so-called selection bit (SELBIT in Figure 2.2):

- Command frame (SELBIT = ”1”)
- Data frame (SELBIT = ”0”)

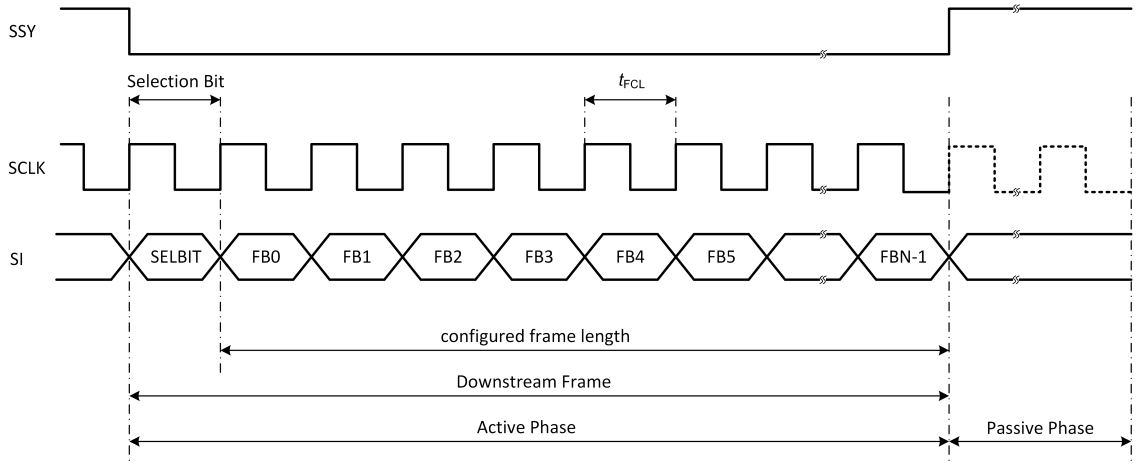

Figure 2.2: A general MSC Downstream Frame.

Two different clocking modes can be distinguished; the continuous clock mode's clock runs continuously, regardless of whether in active or passive phase. In contrast, the discontinuous clock mode's clock runs only during the active phase of a transmission.

2.3.2 Command Frame Description

A command frame is identified with the first bit SELBIT = "1" (see Figure 2.3), followed by the command frame bits, ordered with the least significant bit (LSB) first.

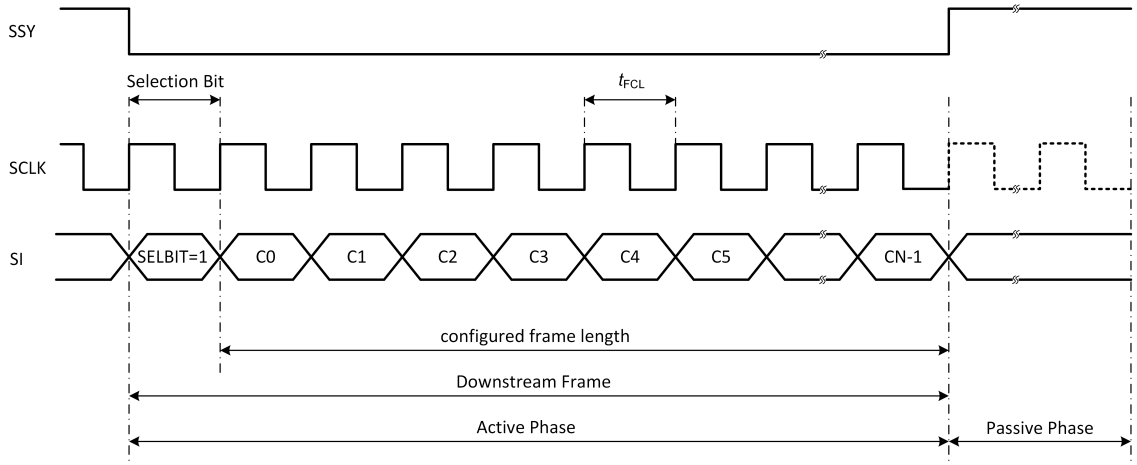

Figure 2.3: A specific MSC Command Frame.

The slave considers a command frame valid if the number of received command frame bits is equal to the configured length.

2.3.3 Data Frame Description

A data frame is identified with the first bit SELBIT = ”0” (see Figure 2.4), followed by the data frame bits, ordered with the LSB first.

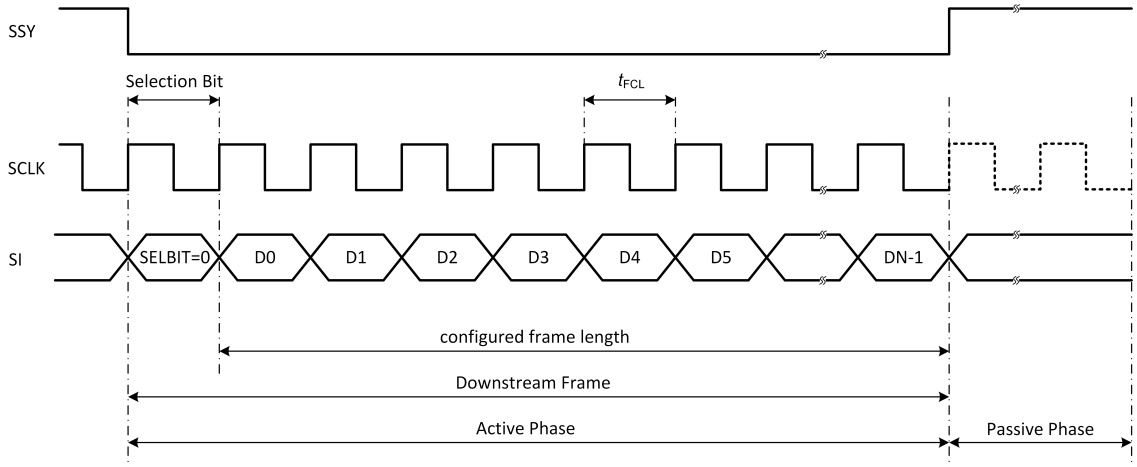

Figure 2.4: A specific MSC Data Frame.

The slave considers a data frame valid if the number of received data frame bits is equal to the expected length.

2.3.4 Slave-to-Master Upstream Communication

The serial upstream channel's output on SDO is continuously at high level, such as additional Stop bits when not used for communication, shown in Figure 2.5. Therefore, an upstream frame starts with an active low bit (Start bit) before the actual upstream data bits (UFB0 to UFBN-1) with their LSB first and is followed by the Parity bit and 1 up to 4 Stop bits. The baud rate of the upstream channel is a divided version of the main clock FCL and can be $FCL/2^X$ where X is an integer from 1 to 8.

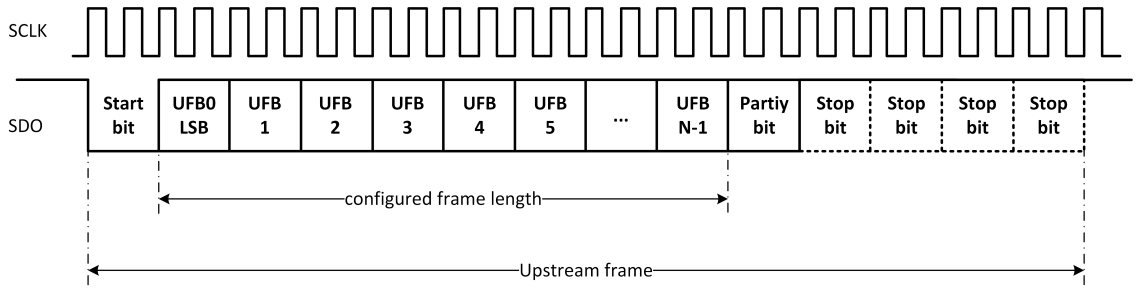

Figure 2.5: MSC Upstream Frame illustrates Clock Divider X set to 1.

In brief, there are important SystemVerilog features; the most basic OOP concepts used to create the verification environment are highlighted in the following chapter. The basics are described based on the concepts mediated by the "SystemVerilog Language Reference Manual" [2] and further illustrated through a combination of theoretical and practical examples taken from "SystemVerilog for Verification"[3] as well as "Writing Testbenches using SystemVerilog"[4] and adopted according to this thesis's purpose.

[2] [IEEE, 2013]
[3] [Spear, 2010]
[4] [Bergeron, 2006]

Chapter 3

SystemVerilog Language Features

This chapter's purpose is to serve as an introduction of object-oriented programming concepts and features as well as to provide a brief overview of possibilities of SystemVerilog entangled in pre-silicon functional hardware verification combined with object-oriented software development applied on a microelectronic device before the process of manufacturing can start.

SystemVerilog is the "Unified Hardware Design, Specification, and Verification Language.":

> "The definition of the language syntax and semantics for SystemVerilog, which is a unified hardware design, specification, and verification language, is provided. This standard includes support for modelling hardware at the behavioural, register transfer level (RTL), and gate-level abstraction levels, and for writing testbenches using coverage, assertions, object-oriented programming, and constrained random verification. The standard also provides application programming interfaces (APIs) to foreign programming languages."[1]

For starting, one of the first basic object-oriented programming principles is to combine code and data to obtain so-called classes, which are described in the following section.

[1] Language Reference Manual - Abstract from p. ii [IEEE, 2013]

3.1 Classes and Objects

Firstly, a class is the description of a user-defined data type that contains members for data as well as subroutines (tasks and functions) to operate on these data members. The data members a class contains are referred to as class properties and the subroutines are called methods; both are members of a class. The class properties and methods combined define contents and capabilities of an object when instantiated.

For instance, a *BaseTransaction*, (at first a class) is a transaction that will be instantiated to one or more objects of that type. Listing 3.1 depicts the class containing all information needed to describe a *BaseTransaction* in terms of properties such as unique identifier, an address, data, etc. Additionally, creational or comparing methods might be embedded into the *BaseTransaction* class, for example, the constructor that is needed to instantiate a **new** object of type *BaseTransaction* or a task to compare member values of type *BaseTransaction* with *AnotherBaseTransaction*.

Listing 3.1: Example BaseTransaction with compare()-function.

```
class BaseTransaction ;
  // BaseTransaction properties
  int UTID;
  bit [31:0] address;
  bit [31:0] data;
  // Constructor of BaseTransaction
  function new ();
      UID     =  8'b0;
      data    = 32'b0;
      address = 32'b0;
  endfunction : new
  // Method declarations
  function bit compare (A_Base_Transaction a_trans );
    if ( this.UTID != a_trans.UTID )
      // do something
```

```
    else if ( this.address != a_trans.address )
      // do something similar
    else if ( this.data != a_trans.data )
      // do something similar
  endfunction : compare
endclass : BaseTransaction
```

For creating or instantiating an object of any type, first a variable of the class type has to be declared as shown in Listing 3.2. This variable is then called an object handle and can be used to hold an object of that specific type. The object is created with the **new** constructor call and directly assigned to the handle *base_tr*.

Listing 3.2: Variable Declaration Object Creation and Handle Assignment.

```
// declare a variable of type BaseTransaction
BaseTransaction base_tr;
// create and assign object to handle
base_tr = new();
```

Nevertheless, there are numerous additional possibilities one can implement with classes and objects and their properties and methods. Therefore, it is recommended to read Chapter 5: Basic OOP in "SystemVerilog for Verification" by Spear [Spear, 2010] in order to ensure a good starting point and an excellent complementary work to the "IEEE Standard for SystemVerilog" by the IEEE [IEEE, 2013].

A more advanced OOP concept is the principle of inheritance, which is explained in the following section.

3.2 Inheritance

In the previous section, a class of type *BaseTransaction* was defined and illustrated to understand how to create and instantiate an object of that type. This class can be extended (keyword **extends**) to become a specialised form of a *BaseTransaction*, e.g. an *ErrorTransaction*. The following code example, presented in Listing 3.3, shows how to define a new subclass of type *ErrorTransaction* that **inherits** all members of the *BaseTransaction* class. This subclass then can further be extended with additional purpose specific members, e.g. an *error_flag*.

Consequently, *ErrorTransaction* holds all members of *BaseTransaction* plus the new added member *error_flag*. Since subclass objects are of the same type as their base class, they can be overridden to alter their definitions.

Listing 3.3: Inheritance Example.

```
class ErrorTransaction extends BaseTransaction;
    // newly added member of error Transaction
    bit error_flag;
    // Constructor of Special Transaction
    function new ();
        // Constructor chaining
        // calling new of super class BaseTransaction
        super.new();
        // init new member
        error_flag = 0;
    endfunction : new
endclass : ErrorTransaction
```

SystemVerilog only supports *single inheritance*, in other words, each specialised class derives from one single base class, as explained in the Language Reference Manual[2].

As the functionality of classes grows during development and by adopting the concept of in-

[2] Language Reference Manual p.144 [IEEE, 2013]

heritance, applying certain other techniques eases organising the structure. One technique, type conversion, is introduced in the following section.

3.3 Casting (Type Conversion)

There are certain aspects concerning casting that have to be considered since some class assignments are illegal, therefore, provoke errors.

> "It is always legal to assign an expression of a subclass type to a variable of a class type higher in the inheritence tree (a superclass or ancestor of the expression type)."[3]

Listing 3.4: Proper Handle Assignment.

```
BaseTransaction base_trans;
ErrorTransaction error_trans;
// Construction of error_trans
error_trans = new (...);
// LEGAL assignment
base_trans = error_trans;
// print the base UTID
$display( base_trans.UTID );
// calls function print of error_trans
error_trans.print();
```

If the assignment is implemented vice versa from the *base_trans* to the *error_trans*, this results in a compilation error because of a static check of the handle types done by the compiler, as mentioned by Spear [Spear, 2010]. Moreover, it is sometimes needed to assign a base handle to a derived handle, especially when the base handle points to a derived handle. When adopting the concept of channels (channels, see Section 4.3), using a *BaseTransaction* as a carrier of specialised transaction is needed.

[3] Language Reference Manual p.146 [IEEE, 2013]

However, for assigning the base transaction handle to a derived class handle, it is necessary to introduce the "Downcast" function since otherwise the assignment is illegal. Hence, *$cast* system task allows to safely assign a superclass handle to a subclass handle and therefore, makes all derived class members accessible again. According to Spear [Spear, 2010], this system task can be called either as task or function, see the following Listing 3.5:

Listing 3.5: Either called as Task or Function.

```
task $cast( destination , source );
or
function int $cast( destination , source );
```

It has to be noticed that if *$cast* is executed as a task, it results in a run-time error in case of type incompatibility.

In contrast, when called as function, the return value is either *true* or *false*, but does not result in an error if the types do not match. Consequently, *$cast* can be used to enable or disable certain parts and paths of and through the environment, depending on the transaction type.

The following section covers the concept of virtual methods and its connection to polymorphism, which is explained in more detail in Section 3.5.

3.4 Virtual Methods

Methods of a class can be declared as virtual methods by simply declaring the functions prototype with the keyword *virtual*.

> "Virtual methods are a basic polymorphic construct. A virtual method shall override a method in all of its base classes, whereas a non-virtual method shall only override a method in that class and its descendants."[4]

How to properly create a *deep_ copy* or so-called *clone* of an object in an optimised way is depicted in Listing 3.6. This is handled by the *deep_ copy()*-function, which duplicates

[4] Language Reference Manual p.148 [IEEE, 2013]

each existing member and assigns it to the corresponding member of the *clone*. In such a case, it is a derived class and therefore, *$cast* is used as explained in Section 3.3.

Listing 3.6: Optimised Way of Cloning an Object.

```
class ErrorTransaction extend Transaction;
  bit error_flag = 1'b0;

  virtual function void deep_copy(BaseTransaction _base);
    ErrorTransaction error_trans;
    // deep_copy of BaseTransaction
    super.deep_copy(_base);
    // type conversion
    // is the base of type error
    $cast(error_trans, _base);
    // copying the error flag
    error_trans.error_flag = error_flag;
  endfunction : deep_copy

  virtual function BaseTransaction clone();
    ErrorTransaction clone;
    clone = new();
    deep_copy(clone);
    return clone;
  endfunction : clone
endclass : ErrorTransaction
```

In contrast, a so-called *shallow copy* simply copies the references of members (e.g. memory addresses). When creating a new object and assigning it to a variable of the same object type, all members are linked to the new object members, thus, the new object merely points to the old object members. However, a *deep_ copy*, respectively, clone of that object is needed in order to access members.

When declaring a virtual method, it is essential to prepare precise preliminary planning of its particular purpose. Once a virtual method's signature is defined with all its arguments, it is not possible to alter it in any derived class. All derived classes have to use exactly the same signature defined in the base class as depicted by Spear [Spear, 2010]. Without this restriction, polymorphism, explained in the following chapter, in its concept no longer works.

The following section covers the concept of polymorphism based on the example regarding abstract classes and pure virtual methods. These techniques are used to create templates and add re-usability to the verification environment.

3.5 Abstract Classes, Pure Virtual Methods and Polymorphism

In this work, Doulos's class-based verification library template [Doulos, 2012] was used as a starting point for the development of the verification environment; it has a similar architecture compared to the one described by Spear [Spear, 2010]. The concept of abstract classes allows to define the verification environment's basic structure, which is elaborated below.

3.5.1 Abstract class

A class is only called an *abstract class* if the keyword **virtual** is written in front of the class prototype, similar to virtual methods. This means that the class shall serve as a template for various classes derived from that type. In Listing 3.7, the *BaseTransaction* defined in Section 3.1 is redefined to become an *abstract class*. By using *abstract classes*, a collection of classes may be created by deriving all specialised class types from a common base class, e.g. *BaseTransaction*. This abstract base class then defines the template and basic functionality of e.g. a transaction. Nevertheless, the template above is not complete and will not be created, as defined in the Language Reference Manual [IEEE, 2013].

Listing 3.7: Definition of an Abstract Class.

```
virtual class T_BaseTransaction;
  // static in all derived classes
  static int count;
  // unique transaction ID
  rand bit [7:0] UTID;
  // Constructor implementation extern
  extern function new(Component owner = null);
  // Pure Virtual Method declarations
  pure virtual function bit compare(T_BaseTransaction _other);
  pure virtual function void deep_copy(T_BaseTransaction _base);
  pure virtual function T_BaseTransaction clone();
endclass : T_BaseTransaction

// External Constructor of BaseTransaction
T_BaseTransaction::new(Component owner = null);
  // Implicit type conversion
  UTID = count++;
endfunction : new
```

In order to create a *Specialised_ Transaction*, e.g. a *Serial_ Transaction*, the BaseTransaction has to be extended and filled with additional lines of the code, according to its purpose. This *Serial_ Transcation* then can further be extended to e.g. an *Erroneous_ Serial_ Transaction.*

3.5.2 Pure Virtual Method

A method declared virtually inside an abstract class is called a *pure virtual* method and has to be indicated with the keyword **pure**, as defined in the standard. Extensions of this base class may provide implementations by overriding the pure virtual method. All methods declared as *pure virtual* shall have implementations in order to complete the class and to allow instantiation of an object.

Listing 3.8: Specialised Transaction including Complete Function.

```
class SerialTransaction extends T_BaseTransaction;
    ...
    virtual function int compare(T_BaseTransaction _base);
        // function body
        ...
    endfunction : compare2file
    ...
endclass : SerialTransaction
```

However, abstract classes and pure virtual methods are fundamental for the following concept of dynamic method lookup, referred to as polymorphism.

3.5.3 Polymorphism

The concept of polymorphism allows using a variable of the superclass type (*T_BaseTransaction*) to hold an object of any derived subclass type and to address the methods of those classes in a direct manner. As already shown in Listing 3.7, all virtual methods define the basic functionality of a transaction. Although the *T_BaseTransaction* is an abstract class, as mentioned in the Language Reference Manual, it can still be used to declare a handle, respectively, an array of handles:

Listing 3.9: Example Polymorphism.

```
T_BaseTransaction Transactions[10];

Serial_Transaction1 st1 = new;    // extends BaseTransaction
Serial_Transaction2 st2 = new;    // extends BaseTransaction
Err_Serial_Transaction est = new; // extends SerialTransaction
Transactions[0] = st1;
Transactions[1] = st2;
Transactions[2] = est;
```

Objects of various transaction classes of the same base type can be instantiated and combined into the same array, as shown in Listing 3.9: If the data types do not match, it is impossible to store all of them in a single array, however, with the concept of polymorphism, this can be achieved easily.

```
Transaction[0].compare(Transaction[1]);
```

For instance, *Transactions[0]* shall invoke the compare()-function associated with the *Serial_ Transaction1* on Transactions[1] to examine their members for equality. The system properly binds the function of the appropriate class at run-time, as defined in the standard.

In fact, this is a typical example for polymorphism - providing capabilities that are far more powerful than what is found in a procedural programming language.

In the following section, a brief introduction on interfaces is presented since interfaces are a common construct to connect the DUV and the test bench with a certain degree of re-usability added.

3.6 Interfaces

Interfaces can be seen as a packed bundle of interconnection points that simplifies the connectivity. To declare an interface it is beneficial to consider the re-usability and therefore, one has to plan separation of inputs and outputs in advance. When using an interface, it must be instantiated like a module is in Verilog. An interface behaves like a module although with the slight distinction that it can be connected to ports. To access members of this interface via the dot-operator, the module or class has to have an initialised handle to that interface. For more detailed information on how to handle interfaces the reader is advised to consult the "Verification Methodology Manual"[5].

Another construct concerning interfaces are *clocking* blocks. These are used for separating inputs and outputs as well as synchronising them to a common clock source.

[5] Chapter 4 Testbench Infrastructure [Bergeron et al., 2005]

3.6.1 Clocking Blocks

The *clocking* block construct is used to identify clocked signals and synchronises them accordingly to a common clock signal. The most appropriate place for a clocking block is inside an interface. The *SERIAL* interface, depicted in Listing 3.10, is defined with a common clock signal, which is used to synchronise the inputs and outputs accordingly. One can either declare the signals inside an interface of type **wire** as defined by "Rule 4-6 of the Verification Methodology Manual"[6] or of type **logic** as explained in "SystemVerilog for Verification"[7]. Subsequently, Spear focuses on the difference of using logic vs. wire by discussing examples. Nonetheless, it always depends on the factor one needs. For instance, if one recognises - in a later stage of the development - that multiple structural drivers for a certain signal are needed, the signal of type logic has to be altered into wire. Moreover, driving a wire requires some extra code to work properly. Inside a clocking block, specific input and output signal skews are defined with the #-operator followed by the time specified in either s, ms, etc. The input skew set with *#1step* defines the sampling of the signal to the last value directly before the corresponding clock edge, as defined in the standard. Inside the particular interface shown in Listing 3.10, two clocking blocks are defined, the first one for the *driver* and the second one for the *monitor*. Modports provide a controlled way of accessing the interface from the test bench and further provide access, for instance, on the cycles(N)-task defined. This task is used to mimic the behaviour of the cycle delay ##**N**[8], which is not allowed to be used in neither modules nor classes. With the import of this task, it becomes accessible by the test bench and can be used for cycle-based driving of signals inside classes.

Listing 3.10: Example Interface with Clocking Blocks embedded.

```
interface SERIALif (input bit clk);
    logic SDO, SI, SSY;
    // Clocking blocks to give access to the driver
```

[6] Verification Methodology Manual p.108 [Bergeron et al., 2005]

[7] SystemVerilog for Verification p.90 - 4.3.2 Logic vs. Wire in an Interface [Spear, 2010]

[8] Delaying the execution by N times the clock cycle or clocking event. Language Reference Manual p.308 [IEEE, 2013]

```
    clocking drv_cb @(posedge clk);
        output #1step SI;
        output #1step SSY;
    endclocking : drv_cb
    // Clocking blocks to give access to the monitor
    default clocking mon_cb @(posedge clk);
        input #1step SDO;
    endclocking : mon_cb
    // task to give testbench access to ##N behaviour
    task automatic cycles(int N);
        repeat (N) @(drv_cb);
    endtask : cycles
    // modports to provide controlled access from testbench
    modport drv_mp ( clocking drv_cb , import cycles );
    modport mon_mp ( clocking mon_cb , import cycles );
endinterface : SERIALif
```

Once an interface is created it can be defined to be virtual, explained in the following subsection.

3.6.2 Virtual Interfaces

A very useful construct provided by SystemVerilog are virtual interfaces that provide the possibility to separate the environment from the actual signals of the DUV, as defined in the Language Reference Manual. With the keyword **typedef**[9], a class, or in this particular case the interface, can be declared virtual and made visible before the definition of a certain class. Now the class is able to use the virtual interface called *SERIAL_drv_hook* to operate on the DUV, see Listing 3.11. The dot-operator is used to access the signals packed and offered by the virtual interface handle. This mechanism allows writing a reusable code that

[9] typedef provides a so-called forward declaration of the user-defined data type - Language Reference Manual p.76 [IEEE, 2013]

is able to operate on different interfaces and signals of the DUV.

Listing 3.11: Typedef of a Virtual Interface Hook.

```
typedef virtual SERIAL_if.drv_cb SERIAL_drv_hook;

class SerialDriver;
    // declare the handle for the virtual if hook
    SERIAL_drv_hook serial_hook;
    [...]
    function new (SERIAL_drv_hook _hook)
      serial_hook = _hook;
    endfunction : new
    [...]
endclass : SerialDriver
```

Finalising the basic concepts of object-oriented programming, the three most important features in terms of constrained random verification - SystemVerilog Assertions (SVA), SystemVerilog Functional Coverage (SFC) and Constraints - are explained in the following three sections.

3.7 SystemVerilog Assertions (SVA)

The SystemVerilog Assertion language is a discipline of its own and though it is part of the SV language, syntax and semantics are very different compared to SystemVerilog. An assertion defined in an understandable way by Mehta [Mehta, 2017] is a simple checker that ensures the design not to violate the specification. In case a failure occurs, one wants to be informed about this event and further needs to analyse what exactly has happened. In general, it has to be differentiated between three types of assertions:

- Immediate assertion
- Concurrent assertion
- Deferred assertion (immediate assertions with delay)

Immediate assertions are executed like a procedural statement in the code and therefore, can be used as an input sanity check. Another example is to combine the evaluation of *$cast*-task within an assertion to check for correct downcast evaluation and to report an error if occurred.

Concurrent assertions are used to verify signal transitions behaviour when combined with previously defined sequences and properties. This kind of assertion is edge sensitive and therefore requires a **posedge**[10] or **negedge**[10] signal transition to be triggered such as a clocking event or a different trigger signal connected to the DUV.

The third type, the deferred assertion type, was not used throughout this work. Its intention is to delay the evaluation to a certain time stamp to be in more control of glitches, as addressed in Mehta [Mehta, 2016].

Generally, SystemVerilog Assertions offer four different specifier statements: **cover**, **assert**, **assume** and **restrict**. These statements are defined depending on the verification purpose needed. In this thesis. cover and assert have been the only ones needed. The cover statement can be used to track the property evaluation and collect coverage information but does not interrupt the simulation. Whereas the statement assert is able to interrupt

[10] SystemVerilog keywords for sampling event - Language Reference Manual p.181 [IEEE, 2013]

the simulation in case of a failure and report it with, e.g., a defined error message.

3.7.1 Sequences and Properties

The *sequence* feature is often used to construct properties out of sequential behaviour. A sequence can be defined by simply ordering boolean expressions of the signals in linear time increasing order.

Basic Property

The meaning of the property in Listing 3.12 translates to the following: At the rising edge of a *valid_ trigger* signal, if and only if (**iff**) the reset is not active, check if either:
flag_ one is asserted and *flag_ two* is not,
or *flag_ two* is asserted and *flag_ one* is not
or none of the two flags are asserted at all.
Therefore, the property is not allowed to evaluate to *true* when both flags are active at the same time. In other words, the property is said to check mutual exclusion for these flags.

Listing 3.12: Example Mutual Exclusive Property.

```
property mutex_p1;
    (@(posedge valid_trigger iff ( !reset ))
        ( (flag_one && flag_two) ||
          (flag_two && !flag_one) ||
          (!flag_one && !flag_two)));
endproperty : mutual_exclusive_p1
mutex_assert1: assert property (mutex_p1);
```

The example above describes a method of checking two flags for their mutual exclusive assertion. If the outputs are registers, SystemVerilog offers built-in functions. According to the Language Reference Manual, these functions are named "sampled value functions"[11] and are explained in the following subsection.

[11] Language Reference Manual p.360 [IEEE, 2013]

3.7.2 Sampled Value Functions

The four functions **$stable**, its counterpart **$changed** as well as **$rose** and **$fell** were used to write part of the assertions for checking the proper DUV signal behaviour.

$stable and $changed

The meaning of the property in Listing 3.13 translates to the following:
At the rising edge of a *trigger_bus1* signal, **iff** currently not in *reset*, one has to check if: *bus_out1* has $changed in its value and *bus_out2* remained $stable like it was before the trigger had activated the check.

Listing 3.13: Example $stable and $changed in a Property.

```
property mutex_bus1_not_bus2;
    (@(posedge trigger_bus1 iff ( !reset ))
        $changed(bus_out1) && $stable(bus_out2)
    );
endproperty : mutex_update_bus1_not_bus2

mutex_bus_update_assert1: assert property(mutex_bus1_not_bus2);
```

$rose and $fell

The meaning of the property in Listing 3.14 translates to the following: At the rising edge of *CLK* used as a trigger signal, **iff** not in reset and *enable_toggling* is active, check if:
The toggle input $rose in its transition from 0 to 1 and the toggle output followed one cycle later or the toggle input $fell in its transition from 1 to 0 and the toggle output followed one cycle later.

If this property does not evaluate to *true*, the assertion is triggered, the simulation is interrupted and an error is displayed.

Listing 3.14: Example $rose and $fell in a Property.

```
property ex_toggle_p1;
    (@(posedge CLK iff ( !reset && enable_toggling))
        ( ($rose(toggle_i) |->##1 $rose(toggle_o)) ||
          ($fell(toggle_i) |->##1 $fell(toggle_o) ));
endproperty : wdog_toggle_p1

toggle_assert1: assert property (ex_toggle_p1);
```

Listing 3.14 shows a very rudimentary linear sequence. For more information on assertions, properties and sequences, especially sequence operators and repetition of sequences, it is recommended to consult the already cited books. In fact, to gain a better understanding of the entire theory, the Language Reference Manual is sufficient. Nevertheless, the book "SystemVerilog Assertions and Functional Coverage" by Mehta [Mehta, 2016] provides theory as well as practical examples.

The following section of this chapter highlights some SystemVerilog Functional Coverage (SFC) language features.

3.8 SystemVerilog Functional Coverage (SFC)

Besides the capability of Assertion-Based Verification (ABV), SystemVerilog offers the Functional Coverage language. (SFC) provides the possibility of creating **covergroup** constructs and embedding user-defined **coverpoints**, as can be seen in Listing 3.15. A coverpoint can contain multiple cover **bins** that are used to define and store the information on a hit. The covergroup needs a defined signature, e.g. *dwstr_conf_cg*, and a trigger for sampling the coverpoint automatically on the event of a certain expression evaluating to *true*. Furthermore, a covergroup offers the possibility to specify a couple of instance-specific options such as *name*, *per_instance*, *comment* and *at_least*.

Most parameters are self-explanatory in their meaning and detailed descriptions and various examples can be found in almost any book regarding SystemVerilog, Functional Coverage or Constrained Random Verification.

Listing 3.15: Downstream Functional Covergroup Example.

```
covergroup dwnstr_conf_cg @ (posedge frame_end );
  option.per_instance = 1;
  option.name = "Downstream Covergroup";
  option.comment = "Samples the frame lengths transmitted";

  cmd_length_cp:  coverpoint cmd_frame_len  {
      bins spec_cmd_len  = { 6'd15 };
      bins other_lengths = { [6'd2:6'd14],[6'd16:6'd63] };
      bins others        = default;
  }
  data_length_cp: coverpoint data_frame_len {
      bins spec_data_len  = { 6'd15 };
      bins other_lengths  = { [6'd2:6'd14],[6'd16:6'd63] };
      bins others         = default;
  }
```

```
  cmd_valid_cp:    coverpoint valid_cmd {
      bins cmd_valid      = { 1 };
      bins cmd_invalid    = { 0 };
  }
  data_valid_cp:  coverpoint valid_data {
      bins data_valid      = { 1 };
      bins data_invalid    = { 0 };
  }
  // Lenghts X valid
  cmd_length_X_valid: cross cmd_length_cp, cmd_valid_cp;
  data_length_X_valid: cross data_length_cp,data_valid_cp;
endgroup : msc_dwnstr_conf_cg

dwnstr_conf_cg dwnstr_conf_cg_inst = new;
```

With different coverpoints sampled in a covergroup, SFC allows to build **cross** coverage, which enables fine grain coverage analysis. It is essential to instantiate a covergroup with the new-operator since otherwise no coverage information is collected. A paper regarding the proper use of all possible options was published by Smith [Smith, 2009] at the DVClub Austin Conference 2009. The reader is advised to consult this paper before starting to build a coverage model.

The second way of sampling coverpoints is to be sampled manually by overriding the built-in sample()-function in order to meet specific needs, see Listing 3.16. Such a need may occur, for instance, in case of logging some data for passing transactions, hence, being able to reflect the correct transmission of data bits. The following example represents an embedded covergroup with a overridden sample function.

Listing 3.16: Example Embedded Covergroup with sample()-function.

```
covergroup updata_check_cg with function sample( Upstr_Trans mon );
  //emebedded covergroup
  option.per_instance = 1;
```

```
  option.name = "MSC Upstream Data Checker";

  msc_upstr_data_cp: coverpoint mon.data2TX {
    bins upstr_eat_cable  = { 63'h3A7CAB1EDEADBEEF };
    bins upstr_all_ones   = { 63'h7FFFFFFFFFFFFFFF };
  }
endgroup : msc_up_data_check_cg
```

As a last mandatory feature, a coverpoint has the capability to track transitions. This can be done by using the =>-operator in the coverpoints bin definition, see example in Listing 3.17.

Listing 3.17: Mode Transition Covergroup Example.

```
covergroup mode_cfg_cg @( mode_trigger );
  option.per_instance = 1;
  option.name = "Mode Transition Covergroup";

  mode_trans_cp: coverpoint mode_cfg iff{!reset}{
    bins msc_spi_dsra = ( msc_mode  => spi_mode => dsra_mode );
    bins dsra_spi_msc = ( dsra_mode => spi_mode => msc_mode );
    bins spi_msc_dsra = ( spi_mode  => msc_mode => dsra_mode );
    bins dsra_msc_spi = ( dsra_mode => msc_mode => spi_mode );
  }
endgroup : mode_conf_cg

mode_cfg_cg mode_cfg_cg_inst = new;
```

Finalising this chapter by providing an overview of the role of constraints in this highly sophisticated language standard, which already has become important in the field of verification, as mentioned in the introduction.

3.9 Constraints

SystemVerilog constraints are one of the three main interests in the subject of constrained random verification. As already described in Section 1.4.3, constraints allow the shaping and automatic generating of the input stimuli needed to automate such a verification environment and in addition, reaching interesting corner cases one has not explicitly thought of.

At first, a variable has to be indicated by the identifier **rand** or **randc** in order to become a random variable and and be solved by the constraint solver.

The constraint constructs in Listing 3.18 represents simple input constraints for a certain transaction. For instance, the *frame_length* is defined to be cyclic-random (**randc**[12]) and constrained inside the range of [1 : *'REG_WIDTH*[13]-1], depending on the same bit width the member *data* is sized and constrained with a random value according to the *frame_length*. This is guaranteed if one places a third constraint that fixes the order to solve the constraints. In this particular example the *frame_length* shall be solved before the *data*.

Listing 3.18: Example Possible Constraints Solving.

```
randc int frame_length;
constraint frame_length_range {
    frame_length inside {[1:'REG_WIDTH-1]};
}
rand bit data['REG_WIDTH:0];
constraint data_constrained_length {
    data inside {[1:(2**frame_length)-1]};
}
constraint solve_length_size {
    solve frame_length before data;
}
```

[12] cyclic-random means no value is repeated until all in a certain range have been picked once.
[13] *'REG_WIDTH* is defined via a compiler declaration

Chapter 4

Verification Environment

The base components of the environment, as mentioned in Chapter 1, originate from Doulos and appear to be a compact class library to provide a basic structure for a constrained random and coverage-driven verification environment. The template is based on principles, rules and recommendations of the ''Verification Methodology Manual''[1] and composed of the following four vital class templates:

- Virtual Component class
- Virtual Transaction class
- Channel class
- Objection class

The following four sections serve as a brief introduction to illustrate the purpose of each of these four templates as a recommended structure for the test bench.

4.1 Virtual Transaction Class

The virtual transaction class serves as the template for each specialised transaction to spread it across the environment where needed. A transaction, by definition, is an atomic

[1] [Bergeron et al., 2005]

operation on the DUV. Moreover, a transaction can be seen as a message that contains information regarding how to stimulate the design and simultaneously predict the response transaction for a certain input transaction.

- Virtual Class Transaction

  Is used to derive and implement specialised transactions.

  These are then randomised by a so-called Stimuli Generator and

  - sent to the driver in order to stimulate the design properly.
  - sent to the reference model in order to predict the design's response.

4.2 Virtual Component Class

Using the Doulos class-based verification library, the virtual class component serves as base class or base building block for each derived verification component such as stimuli generators, drivers, monitors, checkers and to assemble agents. The latter are complete verification components containing a driver and a monitor. The basic implementation of this virtual class is able to build up and maintain a hierarchy of components.

- Virtual Class Component

  Is used to derive and implement components such as stimuli generators, drivers, monitors and checkers.

  These are then used to compose agents, which are able to automatically:

  - stimulate the DUV,
  - collect and assemble response transactions,
  - send the responses to the checker.

4.3 Channel Class

The verification environment needs reusable verification components to be flexible, therefore, verification components must be decoupled. It is essential that components do not know anything about the surroundings they are connected to. This state is achieved by using channels to pass transactions between different components. Channels can be seen as a so-called first-in-first-out (FIFO) buffer, therefore, it is not needed to synchronise the threads for transaction distribution.

- Channel

  A FIFO channel is used to decouple components, which

  - increases the re-usability and independence.
  - hides components details.

Once decoupling is achieved, it is possible to exchange, for instance, a basic driver component with a more sophisticated one during the simulation execution of a test scenario.

4.4 Objection Class

The objection class is a simplified automatic end-of-test mechanism. This allows components to raise objections if they are busy with processing. Furthermore, it allows components running in a forever()-loop, such as the monitor, to be ended (see Listing 4.3). The objection class is implemented to automatically end finished components bottom-up the hierarchy. This reverse ordered ending of components maintains the verification environment's hierarchy. Once all components have dropped their objections, the test bench top holds the last objection active and by dropping it, the test bench can terminate safely.

- Objection

  simplified End-of-Test mechanism, allows components to

  - raise objections, if they are still busy with processing.

– drop objections, if they have finished processing.

One aspect to use this template is that the hierarchical creation and instantiation is guaranteed by the basic implementation of the class-based library as well as the synchronisation of multiple concurrent threads by using the concept of channels. Consequently, once it is developed it guarantees and maintains the hierarchy without the user's concern.

4.5 Recommended Test Bench Structure

In the course material offered by Doulos [Doulos, 2012] some recommendations regarding the test bench structure are provided. For instance, it is recommended to use a module for a test harness and interfaces to encapsulate the connectivity. Further recommendations are to include clocking blocks, modports as well as properties and sequences inside the interfaces. An optional advice is to create a top-level test bench module to include instances of interfaces, designs and test programs as well as the clock generator if it has not been inside a test harness yet.

For optional recommendations, the specific project has to be examined in order to consider if they are beneficial to utilise or not.

In the following sections the architecture and specialised verification components are explained in more detail.

4.6 Verification Components

The specialised components building the architecture of the environment are all derived from the virtual component class in order to guarantee a hierarchical structure.

Figure 4.1 presents the simplified block diagram of the developed constrained random verification environment with the most significant parts highlighted.

The most significant parts (in hierarchical order) are as follows. The *test_case* controls parts of the environment for configuration purpose. Inside the *test_case*, parts of the environment are defined and the transactions are configured. For instance, the *MSC_component*, *SPI_Component* and *APP_Component* encapsulate their specific implementations of the corresponding verification components such as a stimuli generator, driver and monitor.

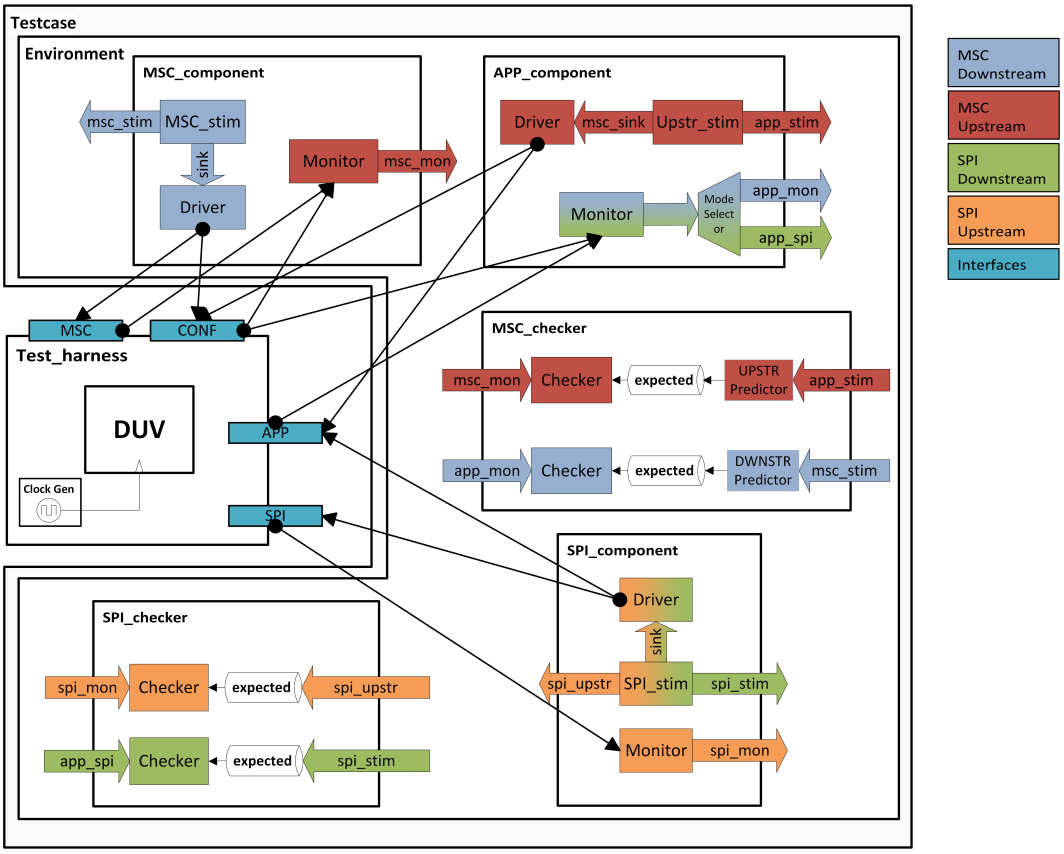

Figure 4.1: Basic Block Diagram of the Verification Environment.

Furthermore, channels are used to interconnect verification components and checker components (*MSC_Checker*, *SPI_Checker*). The four interfaces (*MSC*, *SPI*, *APP*, *CNF*) defined inside the *test_harness* are used to connect the instance of the DUV to drivers and monitors during run-time of the simulation. This entire structure serves as the basic architecture of the constrained random environment. The latter is composed of one test_harness, four driver classes, four monitor classes, two checker classes, three different transaction types as well as four different interface declarations.

The following sections explain each verification component regarding its purpose.

4.6.1 Stimuli Generator

The stimuli generator is in charge of creating, randomising and distributing transactions across the environment. The following code snippet, presented in Listing 4.1, demonstrates the use of Verilogs/SVs *repeat()* loop inside the stimuli generator to configure the number of transactions generated and distributed. This can be done from the test-level. The following line shows the randomisation procedure (simple function call inside an assertion statement), to check if there are any problems in randomising the transaction. After the randomisation, the transaction is put inside the two channels and sent to the driver as well as the checker.

Listing 4.1: Example Stimuli Generator.

```
task body();
    [...]
     repeat (num_trans) begin
          assert (template.randomize()) else
          $error("randomization failure in %s:\n\t\t%s",
                        get_hier_name(), template.psprint());
          template.static_ID ++;
          template.DWNSTR_ID = template.static_ID;
          // Push the transaction in the queue
          trans_queue.push_front(template.copy());
          // Sent the trans to the driver
          sink.put(template.copy());
          // Sent the trans to the checker
          analysis.put(template.copy());
     end
    [...]
endtask : body
```

4.6.2 Driver

The driver (see Listing 4.2) is the interface for translating transaction data into bit level information in order to apply proper design stimulation by transmitting a correct message or an erroneous one. This is a huge abstraction and beneficial for the rest of the test bench. One simply has to cope with signal transitions during the implementation of the specific drive(transaction)-task, the remaining environment is only confronted with specialised transaction data structures. In order to get access to the signals needed inside the driver, virtual interface handles are used to gather all needed signals, as explained in Subsection 3.6.2.

Listing 4.2: Example Driver.

```
task body();
    [...]
    forever begin
        [...]
        do begin
            assert ($cast(current, tr)) else
            $error("MSC_Driver␣%s␣got␣bad␣transaction:\n␣%s",
                        get_hier_name(), tr.psprint() );
            drive(current);
        end while (source.try_get(tr) && !all_trans_driven);
        all_trans_driven = 1'b1;
        [...]
    end
endtask
```

4.6.3 Monitor

The monitor (see Listing 4.3) assembles a response transactions at the output site of the DUV by collecting information regarding the responses of each single input transaction sent through the DUV. These response transactions are then sent to the checker in order to compare the actual output transaction with the one predicted by the checker, described in Section 4.7. Since the monitor is permanently running in a forever()-loop, a trigger from the DUV is needed. In the example of a serial downstream frame, the *frame_end* signal provided by the virtual interface of the monitor is used which indicates that the downstream transmission has ended.

Listing 4.3: Example Monitor.

```
task body();
    [...]
    forever @(posedge hook.tlm_cb.frame_end) begin
        objection_to_stop.raise();
        // Frame end trigger for MSC dwnstream
        if (hook.tlm_cb.dwstr_frame_end == 1'b1) begin
            // MSC MONITOR app_trans assembly function call
            if (cnf_hook.app_drv_cb.mode_cfg == msc_mode)
                body_msc(app_trans);

            // waiting for the completion of the Handshake
            wait4hsk();
            [...]
            app_trans.APP_ID ++;
        end
        [...]
    end
endtask: body
```

4.7 Checker

The checker (see Listing 4.4) predicts reference transactions that are assembled, depending on the information stored in the input transaction of the DUV. Whether it is a correct or an erroneous transaction, the checker is in charge of predicting the reference transaction and furthermore, receiving the response transaction of the monitor in order to compare both transactions with each other for correctness.

Listing 4.4: Example Checker.

```
task Checker_Component::service_msc_dwnstr();
    Transaction t_base;
    Dwnstr_Trans dwnstr_tr;
    forever begin
        // receive the transaction
        msc_stim_chan.get(t_base);
        // cast the transaction object
        $cast(dwnstr_tr, t_base);
        // predict the outcome based on the transaction
        predict_DUV_output(dwnstr_tr.copy());
    end // forever
endtask
```

4.8 Test Harness

The *test_harness* (see Listing 4.5) is a module which contains the DUV instance and connections to its port as well as the clock generator or other supporting structures such as a memory interface. Doulos [Doulos, 2012] recommends to place the clock generator inside a *test_harness* and never inside the environment. The *test_harness* should be used in a sophisticated way to place all needed interface declarations since these are the interfaces between the environment and the actual DUV.

By utilising a *test_harness*, it is possible to hide the verification environment's structure behind a collection of useful test implementation and even more important, the environment becomes separated from the design. Therefore, the purpose of the *test_harness* is to instantiate the interfaces, clock generation and furthermore, to connect everything accordingly between the DUV and the verification environment.

Listing 4.5: Example Test_Harness.

```
module test_harness();
    [...]
    //System clock generator for clock synch
    always #(sysclk_period/6) sysclk = ~sysclk;
    //MSC Clock Generator with enable possibility
    always #(sysclk_period/6) if(msc_en == 1'b1)
        msc_sysclk = ~msc_sysclk;

    //Interface instantiation
    SERIAL_intf SERif ( sysclk );
    CNF_intf CNFif ( sysclk );
    APP_intf APPif  ( sysclk );
    //Clock enabling/disabling structures
    [...]
    //Signal assignments shared by more than one interfaces
    [...]
    // DUV instanciation
    msc_top i_msc_top
    ( // Port connections
      ... );

endmodule : test_harness
```

4.9 Test Case

A *testcase* class (see Listing 4.6) can be seen as a control entity that is used to configure the needed parts of the environment as well as to define the transaction settings, see Listing 4.6. For example, it is possible to enable, respectively, disable certain constraints or to fix certain transaction members to specific values for analysis of corner cases or directed tests.

Listing 4.6: Example Constrained Random Test Case.

```
class testcase extends base_test;
   // Downstream Transaction Object
    SerialTransaction Serial_tr;
    // Constructor to create test object
    function new(string _name, Component _parent, serial_env _tb);
        super.new(_name, _parent, _tb);
        Dwnstr_tr = tb.SERIAL_comp.serial_stim_gen.template;
    endfunction
    // Transaction configuration
    task configure_test();
        // Assignment of the mode
        assign test_harness.mode_cfg = msc_mode;
        // Create and install a new template object
        tb.set_stim_template( Serial_tr );
        // Setting number of upstream transactions
        tb.set_num_trans(super.transactions);
        // enabling wrong length transmission
        Serial_tr.wrong_length.rand_mode(1);
        Serial_tr.wrong_length_prob.constraint_mode(1);
        // enabling randomisation of lengths
        Serial_tr.cmd_length.rand_mode(1);
        Serial_tr.data_length.rand_mode(1);
```

```
        // enabling full-range constraint
        Serial_tr.cmd_length_range_full.constraint_mode(1);
        Serial_tr.data_length_range_full.constraint_mode(1);
    endtask : configure_test

    // RUN TASK
    task body();
        configure_test();

        // Component setup needed
        fork
          tb.SERIAL_comp.serial_stim_gen.body();
          tb.SERIAL_comp.driver.body();
          tb.checker_comp.service_downstream();
          tb.APP_comp.monitor.body();
        join_none
    endtask : body
endclass : test001_msc_dwnstr_cont
```

In the example above, the **fork** construct is used to launch multiple concurrent environment parts inside threads. No additional synchronisation is needed when using multiple concurrent threads due to the concept of channels, as explained previously in Section 4.3.

In conclusion, to verify the given DUV, a total of 28 test cases have been developed. Ten test cases served the purpose of constrained random templates that have been used to create four more advanced scenarios. Another twelve test cases served the purpose of constrained random-directed tests and the last two test cases have not been used since they did not increase the coverage.

The following chapter presents a brief introduction concerning the DUV and parts of its functionality.

Chapter 5

MicroSecond Channel (MSC) Interface

This chapter refers to the MSC interface (highlighted in Figure 5.1), further on called DUV, which is independent concerning the application and used for integration into a peripheral slave device. The control logic embedded inside the MSC slave device is called *Slave Control* and handles application specific aspects (such as DUV configuration) and the handshake to acknowledge the communication. However, *Slave Control* and all surrounding parts referred as "Analog Parts" are not part of the DUV. Every statement concerning the *Slave Control* should be considered as a recommendation.

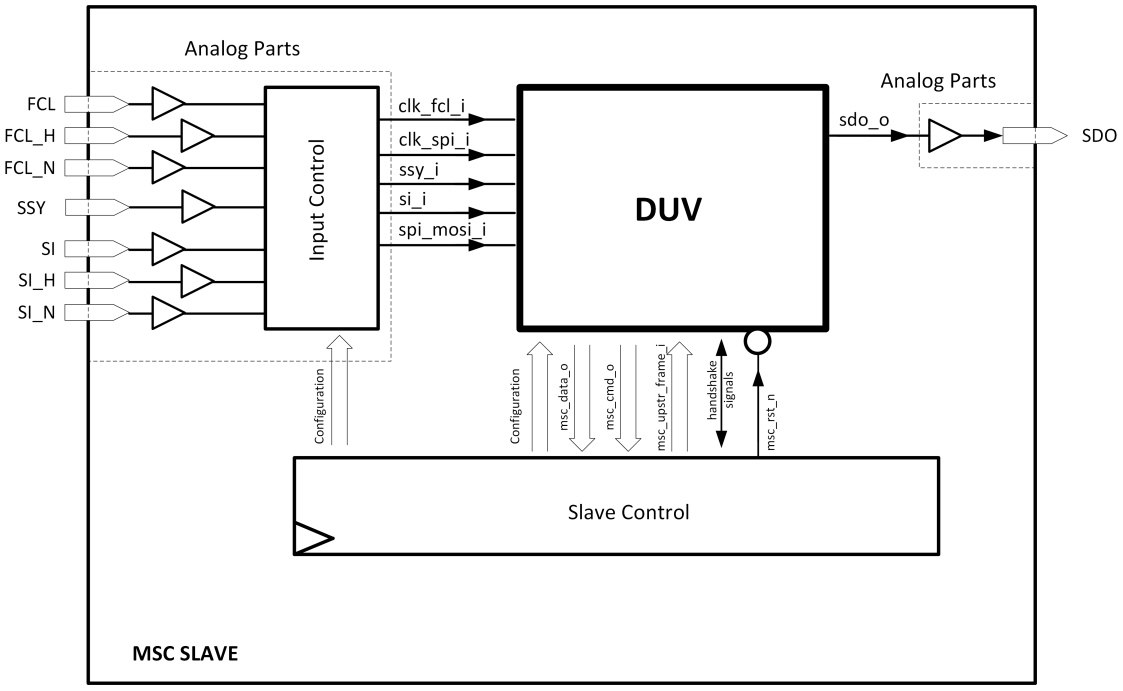

Figure 5.1: Example Integrated MSC Interface.

5.1 MicroSecond Channel Interface

The DUV is designed to perform the MSC standard communication and additionally, is able to perform SPI communication without adding any overhead in area. Since the communication between the DUV and the *Slave Control* has been considered asynchronous, any communication between the DUV and the *Slave Control* can be closed by handshake. The DUV includes all main MSC features described in Section 2.3. Moreover, further degrees of freedom in its functionality have been added in order to be more flexible and reusable accordingly to the application's needs. Part of the additional functionality is the DUV's compatibility with the SPI standard.

The following section explains the DUV regarding its way of receiving data.

5.2 Downstream Communication

The DUV receives a serial input on *si_ i* and in case of a valid communication, the frame is saved into one of two registers, accessible by the *Slave Control* through the outputs *cmd_frame_ o* and *data_frame_ o* (see Figure 5.1). These registers are used depending on whether a command or data frame is processed. If the DUV receives an invalid frame, the outputs previously mentioned are not updated and thus contain the values stored from the last valid transmission. The end of a downstream frame is always signalised by the DUV via the end of frame output (handshake signal in Figure 5.1), which then starts the handshake sequence. *Slave Control* is then able to close the handshake via the acknowledgement input. The entire handshake is handled independently from the clocking mode during the passive phase (see Section 2.3.2).

The following section explains the DUV regarding its way of transmitting data.

5.3 Upstream Communication

Slave Control must apply stable data to be sent to the DUV in order to initiate an upstream transmission. The DUV indicates a frame loaded signal when data has been captured and then starts transmitting data according to the MSC upstream standard (see Section 2.3.4).

The following section explains the clock feedback mechanism in order to enable the possibility of a clock watchdog implementation inside *Slave Control*.

5.4 Clock Feedback

The DUV offers an interface to observe the clock's activity; this scenario serves as the basis for a possible watchdog implementation inside *Slave Control*, refer to Figure 5.2. The DUV is designed to work without any additional clock besides *clk_fcl_ i*, which is independent from the application's clock. This mechanism delays the input signal for a configurable amount of clock cycles to an observation pin.

Therefore, *Slave Control* can configure the cycle delay via the input *wdog_ conf*. The cycle delay can be set to any value between 1 and 255. *Slave Control* has to set and keep *wdog_ i* high and observes the pin *wdog_ o*, which must follow after the configured delay cycles in case of clock activity. If the output does not change after the configured delay cycles, *Slave Control* is able to identify a missing clock signal.

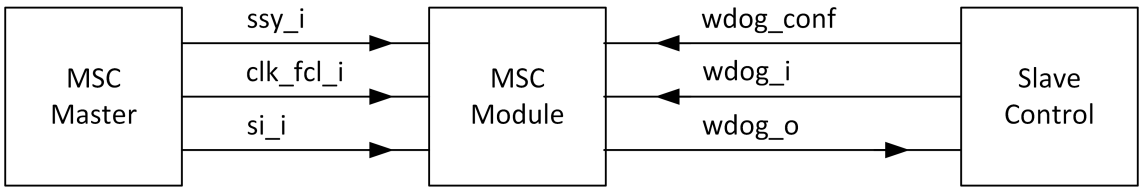

Figure 5.2: Example Blockdiagram Clock Timeout Feedback.

However, it is not possible to transmit data without an active clock signal in such a design.

The following chapter describes how to enable regression simulation and set up structured report generation using Incisive Enterprise Manager, a regression-capable tool designed by Cadence Design Systems.

Chapter 6

Incisive Enterprise Manager - A Regression Suite

The primary aim, besides to develop a constrained random environment, was to establish a sophisticated verification flow including the use of a regression mechanism and tool-supported report generation. This complete regression suite was achieved by integrating the environment into a regression management capable tool such as Incisive Enterprise Manager from Cadence Design Systems.

Regression testing is a technique that has its origin in software development. In case there is a design change, the regression suite can be utilised to quickly verify that a modification has no impact on already verified parts of the DUT. One can download manuals for all tools offered by Cadence Design Systems once registered on the website[1]. The Incisive Enterprise Manager enables to set up a verification plan and handle even big regression runs with ease. Generally, verification planning is a discipline of its own and unfortunately there is no common solution for it. Verification planning is always highly dependent on the specific project details and has to be manually planned and maintained by the verification manager. An advantageous approach is to gather all relevant requirements and translate them into corresponding metrics, respectively, coverage with the help of covergroups and assertions.

[1] Cadence website as ref

6.1 Verification Session Input File (.vsif)

The Verification Session Input File holds the configuration information of the current regression session, hence, various parameters can be specified. The basic structure is divided into session, group and test container that can hold specific parameters valid for either the entire session (all test cases), a group of tests or just a single test to be executed. Inside the session container, the parameter *top_dir* is used to specify the path to store all data produced, as shown in Listing 6.1. The *pre_session_script* is used to create an up-to-date snapshot of the DUT.

Listing 6.1: Example Session Container.

```
session multi_regression_tests {
  top_dir : $ENV(SIM_PATH)/sim4vman/regressions;
  pre_session_script : $ENV(SIM_SCRIPTS_PATH)/pre_session.sh;
};
```

Inside the group container, as presented in Listing 6.2, the parameter *run_script* is in charge of the proper simulation-tool invocation in order to start the elaboration and simulation phase. In case a test consumes an infinite amount of time because of any unforeseeable problem, this can be counteracted by simply specifying a *timeout* in seconds. If the *timeout* threshold is reached, the session is closed and all unfinished test cases are stopped and marked as failed. The filters needed to extract information from the console are specified with the *scan_script* parameter. These filters hold criteria defined for scanning the console output and searching specific string patterns to either indicate a passed, a failed or finished test case. Besides the number of runs per test defined via *count* and the seeds used for initialisation of the random number generator, there are numerous other parameters that can be specified. Detailed explanations regarding such parameters can be found in the corresponding tool documentation.

Listing 6.2: Example Group Container.

```
group group1 {
    run_script   : $ENV(SIM_SCRIPTS_PATH)/run_script.sh;
    timeout      : 7200;
    scan_script  : "vm_scan.pl ius.flt shell.flt \
                   $ENV(SIM_PATH)/vman_sim/pass_fail_filter.flt";
    sv_seed      : 123456789, 234567891, 345678912;
    count : 3;

    test test1
```

Inside the test container, the parameter *top_files* is used to point to the location with the environment's entry point, as shown in Listing 4.6.

Test_command is used to parse System Verilogs $plusargs arguments directly to the environment for configuration purpose. Here the $plusargs is used to specify the testname and the number of transactions to send through the DUV for this test.

```
test test001_msc_dwnstr_cont  {
    top_files     : $ENV(SOURCE_PATH)/sv_ortoff/tb/msc_tb.sv;
    test_command  : +TESTNAME=test001 +Trans=100;
};
```

The basic setup implemented and all parameters properly configured, IEM can now be launched to run a regression.

6.2 A Full Regression Run

Once IEM is launched, the toolbar displays various possibilities for how to proceed.

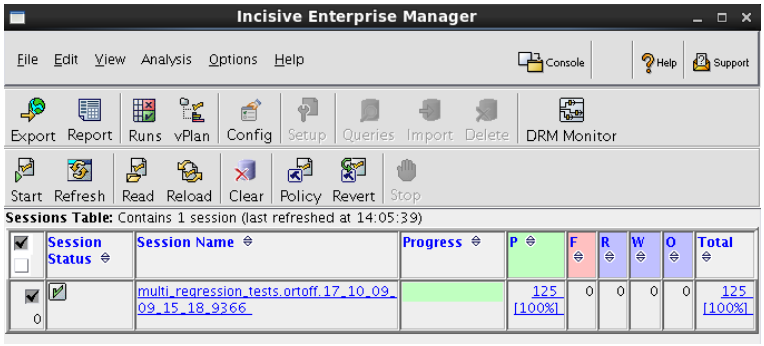

Figure 6.1: Example Finished Regression Run.

A finished regression run, as presented in Figure 6.1, displays an overview of the total number of tests as well as the passed, failed, currently running or waiting tests and even additional information, e.g. if a test has finished in time or not. After clicking the »Start« button, the Verification Session Input File (.vsif) can be loaded and regression starts automatically. During processing, the screen is updated every 5 minutes or can be updated manually by clicking the »Refresh« button. In case the session data of a previous run exists, this can be loaded by a click on the »Read« button. By clicking the »Clear« button with a session selected, the latter can be deleted from the view for clean-up purpose.

The »vPlan« button opens the Verification Plan Tree view, which by default, displays the relevant verification metrics with progress bars, including the percentage as shown in Figure 6.2.

Clicking the »New« button, Enterprise Planner opens a new editor window and the verification planning can begin, see Figure 6.3.

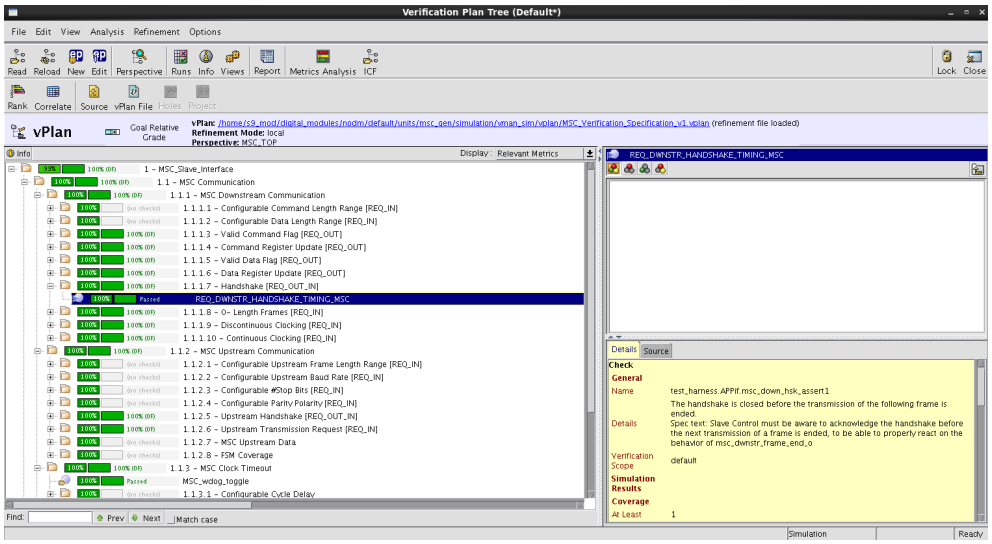

Figure 6.2: Final Report Structure including Progress and Linked Description.

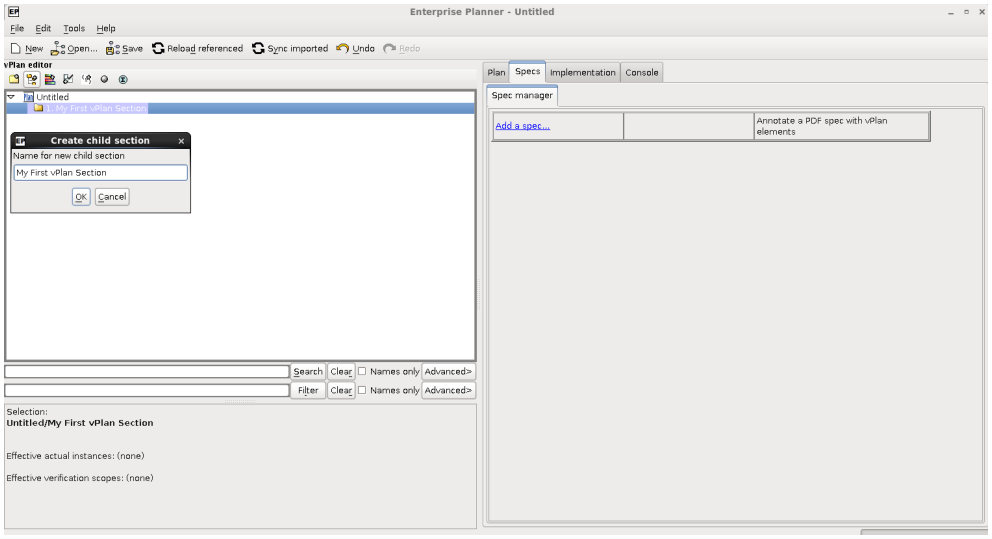

Figure 6.3: Initial Enterprise Planner Editor Window.

In case a plan already exists, it first has to be loaded in order to be edited by clicking the »Edit« button. The graphical user interface allows to set up a basic structure by adding sections and subsections. A specification documentation can be loaded under the tab »Specs« and is then used to annotate the items of the verification plan with details by simply highlighting the text sections and mapping them to corresponding requirements, as can be seen in Figure 6.4. Inside the sections, planned items of type coverage, checker

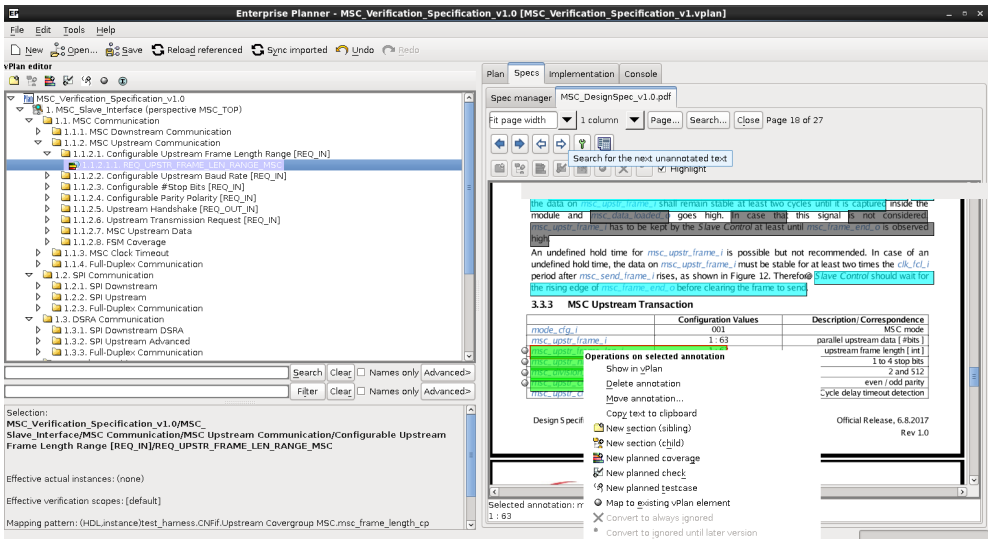

Figure 6.4: Mapping of Text to Plan Sections and Items.

and/or test case can be defined. It is possible to link the planned items against the corresponding verification metrics found under the tabs »Implementation« -> »Metrics«. For further details, it is recommended to consult the corresponding manual of Incisive Enterprise Planner.

After the plan is set up, it has to be loaded into the Verification Plan Tree view. For the sake of proper section numbering, a custom perspective can be created and activated.

Finally, after having all the coverage metrics collected and the plan completed, the verification report is ready to be generated, e.g. a full or summary report as well as custom reports. Therefore, it is recommended to open the Incisive Metrics Center, which will be briefly covered in the following section.

6.3 Incisive Metrics Center (IMC)

Incisive Metrics Center (IMC) is the built-in coverage analysis tool; when launched, it presents a top-down hierarchy including progress bars with percentages for all relevant metrics, see Figure 6.5. Inside IMC, it is possible to refine the coverage metrics. For instance, a case-default statement of a state machine shall never be reached, thus, it is marked as a hole inside IMC. These parts can then be excluded and a comment for documentation can be left. All the refinements made, they can be saved into a refinement file and later on loaded again when needed.

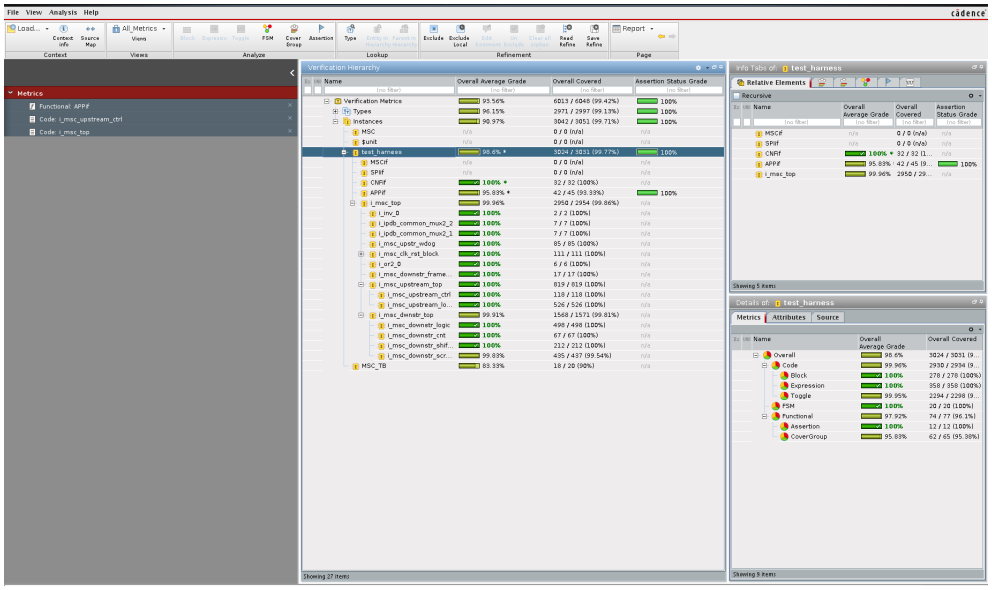

Figure 6.5: Incisive Metrics Center integrated Coverage Analysis Tool.

6.4 Generating the vReport

The »Report« dialogue refers to Figure 6.6 and simply is configured by specifying a name, a directory and the depth of detail. Furthermore, the filter criteria and metrics type to include has to be chosen according to the wanted result. The »OK« button starts the report's generation. The resulting report is an objective HTML website and includes a navigation menu.

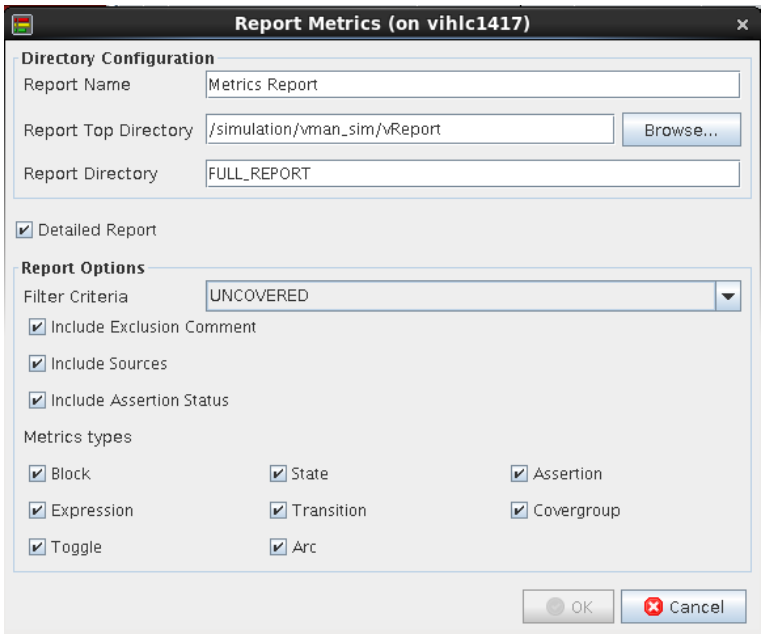

Figure 6.6: Options for Report Generation.

In conclusion, the 26 developed test cases used for the regression and metrics extraction yielded in a total functional coverage of **98.6%**. Code coverage reached 93.56%, whereas the four advanced scenarios mentioned in Section 4.9 mostly contributed to the overall coverage with 83.33%. Excluding certain lines of code regarding sanity checks and case-default statements refined code coverage to a final percentage of **99.96**. The missing coverage (functional 1.6%, code 0.04%) can be explained by differences between the MSC and the SPI protocol. The MSC is able to differentiate between command and data frames, as explained in the corresponding DUV chapter, whereas the SPI does not include such a feature.

Chapter 7

Conclusion

This work presents a common and modern way of developing a reusable constrained random verification environment in SystemVerilog. The resulting test bench and regression suite enable the functional hardware verification on a digital circuit, e.g. a protocol handling device. Since this device has not been verified yet this thesis cannot provide any direct comparison to the traditional approach.

However, the architecture of such a verification environment, as explained in Section 4.6, is designed to be reusable amongst similar projects. For instance, in case an Inter-Integrated Circuit slave device needs verification, first, the environment has to be adjusted according to the needs of the circuit and specifications. The setup of needed components does not change, therefore, the basic structure with stimuli generators, drivers, monitors, checkers, predictors and interfaces do not alter. The implementation will be a different one as explained in Section 3.5; so will be the coverage model.

Concerning this thesis, the acquisition of knowledge and skills has demanded great effort to enable discussing this vast topic of functional verification. In addition, gathering as well as combining all the know how to accomplish the goal of a pre-silicon verification flow and a stable regression suite have required a vast amount of time. In fact, it would be even more delicate if one wished to develop an entirely constrained random verification environment; the effort compared to the result is too high with the used class-based library. Nonetheless, the result is a flexible and reusable constrained random environment, especially when

considering regression testing and metrics extraction as well as reporting, as mentioned at the end of Chapter 6.

Developing an entirely random verification environment requires the additional support of a more sophisticated class library such as the Universal Verification Methodology. This methodology offers numerous possibilities for test bench configuration and execution, for instance, the use of design patterns, e.g. the factory pattern or a sophisticated console report generation.

The class library offered by Doulos [Doulos, 2012], which has been used for the practical work of this thesis, mainly holds the purpose to serve this very complex topic of functional hardware verification in a compact format, especially to young professionals and hardware designers. However, starting with the Doulos class-based approach helped to understand the fundamentals of functional hardware verification.

Since the Doulos class-based approach is designed for trainings and courses on design and verification, for further projects a transition to the Universal Verification Methodology standard should be considered.

Bibliography

[Bergeron, 2006] Bergeron, J. (2006). *Writing Testbenches using System Verilog*, volume 1. Springer.

[Bergeron et al., 2005] Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A. (2005). *Verification Methodology Manual for SystemVerilog*. Secaucus, NJ, USA.

[Doulos, 2012] Doulos (2012). *Comprehensive SystemVerilog - Course Material*.

[IEEE, 2013] IEEE, C. (2013). *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*.

[IPExtreme®, 2007] IPExtreme®, I. (2007). *Infineon MicroSecond Channel Interface*. 307 Orchard City Drive, M/S 202, Campbell, CA95008.

[Kelling et al., 2005] Kelling, N., Koenig, M., and McNair, K. M. (2005). Microsecond Bus (μSB): The New Open-Market Peripheral Serial Communication Standard. In *SAE Technical Paper*. SAE International.

[Knig, 2012] Knig, H. (2012). *Protocol Engineering*. Springer Publishing Company, Incorporated.

[Mehta, 2016] Mehta, A. B. (2016). *SystemVerilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications*. Springer Publishing Company, Incorporated.

[Mehta, 2017] Mehta, A. B. (2017). *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. Springer Publishing Company, Incorporated, 1st edition.

[Mentor Graphics, 2012] Mentor Graphics, V. M. T. (2012). *Coverage Cookbook*, volume 1. Mentor Graphics Corporation.

[Miller and Maloney, 1963] Miller, J. C. and Maloney, C. J. (1963). Systematic Mistake Analysis of Digital Computer Programs. *Commun. ACM*, pages 58–63.

[Smith, 2009] Smith, D. (2009). A Practical Look @ SystemVerilog Coverage - Practical Tips, Tricks, and Gottchas using Functional Coverage in SystemVerilog. In *Design and Verification Club Austin, DV Club Austin*. https://www.doulos.com/knowhow/sysverilog/DVClub_Austin_09/.

[Spear, 2010] Spear, C. B. (2010). *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd edition.