Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master's Degree in Computer Engineering

Automated UVM Testbench Generation Using EMF

Supervisor:
Prof. Daniele Vogrig

Company Tutor:
Ing. Enrico Vincenzi

Student:
Victor Isachi
2005836

Academic year 2021/2022

Abstract

The modern digital integrated circuit (IC) design flow requires a lot of verification effort. Furthermore, this effort is expected to grow in the future. The verification task is fundamentally a race against the clock as it needs to establish a good level of confidence in the design while keeping in mind the time to market. One of the most time consuming, and error prone, aspects of digital verification is the development of the necessary code infrastructure according to the Universal Verification Methodology (UVM).

This work is divided into two parts. In the first part we introduce a widely adopted digital verification toolchain. The latter is composed of SystemVerilog and UVM. To better understand SystemVerilog, Verilog is also briefly introduced. In the second part we introduce a common workflow for Model-Driven Engineering (MDE), a tool that allows us to automate code development. This workflow comprises the Eclipse Modeling Framework (EMF), Sirius and Acceleo. The latter are briefly introduced. Last, we discuss how these tools were used to develop a project for automating UVM testbench generation.

Abstract in Italiano

Il moderno flusso di progettazione di circuiti integrati (IC) digitali richiede un grande sforzo di verifica. Inoltre, ci si aspetta che questo sforzo cresca in futuro. L'attività di verifica è fondamentalmente una corsa contro il tempo in quanto deve stabilire un buon livello di fiducia nel design tenendo presente il time to market. Uno degli aspetti più dispendiosi in termini di tempo e soggetti ad errori della verifica digitale è lo sviluppo dell'infrastruttura di codice necessaria secondo l' Universal Verification Methodology (UVM).

Questo lavoro è diviso in due parti. Nella prima parte introduciamo una toolchain di verifica digitale ampiamente adottata. Quest'ultima è composta da SystemVerilog e UVM. Per comprendere meglio SystemVerilog, anche Verilog viene presentato brevemente. Nella seconda parte introduciamo un flusso di lavoro comune per il Model-Driven Engineering (MDE), uno strumento che ci consente di automatizzare lo sviluppo del codice. Questo flusso di lavoro comprende Eclipse Modeling Framework (EMF), Sirius ed Acceleo. Questi ultimi sono brevemente descritti. Infine, discutiamo di come questi strumenti sono stati utilizzati per sviluppare un progetto per automatizzare la generazione di testbench UVM.

Contents

1 Introduction **1**

I Primer on Digital Verification **5**

2 Verilog by Difference with VHDL **7**

2.1 Fundamental Construct: Module 8

2.2 Data Types 9

2.2.1 Scalar Types 9

2.2.2 Vectors and Arrays 12

2.3 Literals and Initialization 13

2.4 Operators 14

2.5 Continuous Assignments 16

2.6 Procedural Statements 17

2.6.1 Conditional Statements 20

2.6.2 Loops 22

2.7 Structure and Hierarchy 24

2.7.1 Functions and Tasks 25

2.7.2 Generate Blocks 28

2.8 Modeling Building Blocks of Digital Design 29

2.9 Verilog for Verification 33

2.9.1 Fundamental Verilog Tools 33

2.9.2 Useful System Tasks and Functions 36

2.9.3 Testbenches 41

3 SystemVerilog **45**

3.1 Data Types 46

3.1.1 Arrays 46

3.1.2 Queues, Associative Arrays and User Defined Data Types . 48

3.2 Operators 49
3.3 Procedural Statements and Blocks 49
3.4 Tasks and Functions 51
3.5 OOP in SystemVerilog 53
3.5.1 Encapsulation 55
3.5.2 Inheritance 56
3.5.3 Polymorphism 58
3.6 Interfaces 61
3.6.1 Ports and Parameters 61
3.6.2 Subroutines and `modports` 63
3.6.3 Virtual Interfaces 65
3.7 Clocking Blocks 66
3.8 Randomized Stimulus 69
3.8.1 Randomizing Variables 69
3.8.2 Randomizing Properties 71
3.8.3 Constraint Blocks 73
3.9 Coverage 75
3.9.1 The Basic Constructs 75
3.9.2 `covergroups` in Classes and `covergroup` Methods and Options 78
3.10 Assertions 79
3.10.1 Immediate Assertions 80
3.10.2 Concurrent Assertions 81
3.11 Threads and Interprocess Synchronization 83
3.11.1 `semaphores` 84
3.11.2 `mailboxes` 86
3.12 Further Topics 88

4 Universal Verification Methodology (UVM) 91
4.1 Data Modeling 92
4.1.1 Sequences 94
4.2 Simulation Cycle 96
4.2.1 End of Simulation 97
4.3 Interface UVCs 97
4.3.1 Agents 99
4.3.2 Monitors 100
4.3.3 Drivers 101

4.3.4 Sequencers 102
4.4 Configuration 103
4.5 Factory and Type Overrides 104
4.6 Module UVCs 105
4.6.1 Scoreboards 107
4.7 Testbenches 109
4.7.1 Tests 111
4.7.2 Multichannel Sequences 113
4.7.3 Logging 114

II Automated Testbench Generation 117

5 Tools for automatic code generation 119
5.1 Metamodels 119
5.1.1 Modeling 120
5.1.2 Metamodeling 120
5.2 Eclipse Modeling Framework 122
5.2.1 Defining an Ecore Domain Model 122
5.2.2 Use of Metamodeling Tools 124
5.3 Sirius 124
5.3.1 Overview 125
5.4 Acceleo 126
5.4.1 Overview 126
5.4.2 Useful Tips when Developing Acceleo Projects 126

6 evigen 129
6.1 `vipgen` 129
6.1.1 Automated UVM Testbench Generation Framework 130
6.1.2 Interfacing with an Existing Testbench Generator 130
6.2 Interfacing with the Operating System 133
6.2.1 Command Execution 133
6.2.2 Resource Management 134
6.2.3 Reading and Writing to/from Files 136
6.3 Customizing the Sirius Layout Algorithm 138
6.3.1 Notification Listener 139
6.3.2 Graphical Structure 143
6.3.3 Modifying Semantics 144

6.4 Customizing the Eclipse User Interface 145
6.4.1 Custom Eclipse Perspective 145
6.4.2 EMF Wizards 148
6.5 Exporting the Project as a Plug-in 149

Conclusions **151**

Bibliography **153**

Sitography **155**

Chapter 1

Introduction

In the early '70s, at the dawn of the microprocessor, the design flow of digital integrated circuits (ICs) required a single person working both on the logical design, physical implementation and technology, all at the same time. Many innovations were made in this field. For example, to manage the design of circuits with tens of thousands of logic gates and to aid the compatibility of the vendor specific tools with the design, the standard cell (semi-custom) approach was developed. Throughout much of this early period, verifying that the design was compliant with the desired specifications was done post-silicon, i.e. once the chip was fabricated. Thus, chip design was an iterative process that required several design revisions to iron out the existing bugs. As digital ICs grew in complexity, the need for Electronic Design Automation (EDA) software was becoming apparent. With the introduction of Hardware Description Languages (HDLs), such as VHDL and Verilog, and later on in the '90s their added capability to synthesize gate netlists, i.e. translate a digital design described at a higher level of abstraction into a network of logic gates and basic sequential circuits, verification started to also be done before the chip was made. This new kind of verification was dubbed pre-silicon. Designers describe the functionality of the circuit at the level of abstraction of the Register Transfer Model (RTL). Pre-silicon verification can check if the RTL design complies with the specification. Furthermore, once the RTL is translated into a gate netlist, it is possible to check RTL and netlist equivalence.

Design and verification have evolved throughout the years. One of the innovations was the introduction, at the beginning of the new millennium, of a new type of language. A Hardware Description and Verification Language (HDVL), namely SystemVerilog. This new language provides both the capabilities of a HDL, in fact SystemVerilog is a super set of Verilog, as well as some features not found in HDLs, such as Object Oriented Programming (OOP), coverage constructs, interfaces, etc., that are useful for verification. A common modern development flow consists in using SystemVerilog for both design and verification. With the latter being done in accordance with the Universal Verification Methodology (UVM). UVM, introduced in 2011, condenses many of the industry best practices and aids verification by providing a way of developing extensible, partially automated, reusable and flexible testbenches and verification components. In order to develop these, however, a significant amount of code needs to be written.

This work gives an overview of UVM, SystemVerilog and Verilog, from the perspective of a reader that is familiar with VHDL, in chapter 4, chapter 3 and chapter 2 respectively.

As state previously, writing UVM verification components (dubbed UVCs) and testbenches requires a non-trivial amount of code. Luckily, there is a software engineering framework, Model-Driven Engineering (MDE), that can help us partially automate code development. This framework is based on a very successful idea of computer science, abstraction. As we will later see, UVM has standard architectures for the components that populate its testbenches. We can abstract out these architectures. This allows us to provide the user with a way of defining the desired architecture, not the associated code. The former is then automatically translated into code with the use of a code generator. This can clearly help us automate the development of the code infrastructure that is needed to properly set up and use UVM.

A common and open-source set of tools that allows us to work with MDE is based on the Eclipse IDE. This set is composed on the Eclipse Modeling Framework (EMF), Sirius and Acceleo. EMF allows us to define metamodels, i.e. abstractions of the entities we model with software. These metamodels represent the possible models that we can develop. Once we have developed metamodels for the common UVM architectures, we can use these to allow the user to specify any valid UVM architecture. Acceleo allows us to query the models defined by the user. That information can then be used, together with specific language constructs that are designed for automatic code generation, in code templates that are translated to the final code files. The last tool, Sirius, allows us to develop a Graphical User Interface (GUI) for model specification. With it we can allow the user to work with an intuitive GUI when defining the UVM architecture, instead of a more cryptic menu based default provided by EMF.

Metamodeling, EMF and the rest of the tools are briefly covered in chapter 5. This chapter contains many references to online tutorials that can get you started with MDE.

The last part of this work, chapter 6, is dedicated to giving an overview of how the presented MDE tools have been used to develop a tool for automatic UVM testbench and UVC generation. You can also find some indications on the problems that were faced during development and the adopted solutions.

This dissertation contains many code examples that are meant to illustrate the described topics.

References to most of the online material that was used during thesis development are included. There is no possibility to guarantee that the referenced links will not be broken in the future.

Part I

Primer on Digital Verification

Chapter 2

Verilog by Difference with VHDL

VHDL and Verilog are the two original, independently developed, HDLs that permeated (and still do) the modern digital IC design flow started in the '80. Many of the features of one language carry over to the other. However, there are some key differences between the two, both in terms of the provided constructs, syntax, as well as design philosophy. In this chapter we will explore the Verilog HDL from the perspective of a VHDL digital designer, as such, it is assumed that the reader is familiar with the latter. Both the differences between the two HDLs as well as some of the Verilog specific features will be presented. This guide does not aim at exhaustively covering Verilog, rather, it hopes to aid the transition from VHDL.

There are plenty of resources that cover the Verilog HDL, without assuming prior knowledge of VHDL. One such resource is [1] that provides a clear and concise introduction to the language. For an online introduction consult [2] instead. [3] provides a more in depth overview of Verilog. Finally, some excellent courses can be found on Cadence's website [4].[1] To fully appreciate the above cited material, it is advised the reader be familiar with digital ICs and digital design (if that is not the case consult [5]).

As stated before, the reader is assumed to have working knowledge of VHDL. Nevertheless, I will leave some references to books that cover this topic. [6] is a textbook covering both VHDL and digital design.[7] provide a more advance look at how to optimize RTL models in VHDL.

[1]The courses might not be available for free. They are usually available to employees of companies that have agreed on a Cadence license.

2.1 Fundamental Construct: Module

The latest, 2005, Verilog standard (IEEE 1394-2005) and practices will be used throughout this chapter.

Verilog is a **case sensitive**[2] and **loosely typed**[3] language. Its syntax is similar to that of the *C Programming Language* [8][9]. For example, Verilog's comment syntax is identical to C's:

- `//this is a single line comment`
- `/*and this is a multiline comment*/`

The fundamental building block of Verilog is the **`module`**. It takes the place of VHDL's `entity-architecture` pair in describing the individual components of a design. Like in VHDL's case, each `module` is usually contained in a separate file, which in Verilog's case have the `.v` suffix. Unlike in VHDL, each `module` has one, and only one, associated architecture. An example that illustrates `module`'s syntax is presented below (Code 2.1).

```verilog
module add
  #(parameter integer N = 8)
   (input wire         en,
    input wire [N-1:0] a, b,
    output reg [N-1:0] s,
    output reg         c);
     ...
endmodule
```

Code 2.1: Verilog

```vhdl
entity add is
  generic(N: integer := 8);
  port(en  : in std_logic;
    a, b: in  signed(N-1 downto 0);
    s   : out signed(N-1 downto 0);
    c   : out std_logic);
end add;

architecture rtl of add is
   ...
begin
   ...
end rtl;
```

Code 2.2: VHDL

We can see that Verilog appears to be less verbose than VHDL, this holds true in general.

Note that the Verilog **`parameter`** keyword allows us to achieve the same behaviour as VHDL's `generic` keyword.

[2]Case sensitivity implies that `ab`, `Ab`, `aB` and `AB` are all distinct variables.

[3]Being loosely typed implies the existence of language mechanisms that allow the execution of operations not defined for that specific data type (e.g. performing the logical and of two floating-point numbers).

Code details will become clear in the next section. For now, I will anticipate that **`wire`** and **`reg`** are Verilog data types, and can be omitted when declaring a `module`. If we do so, the **`input`**, **`inout`**, and **`output`** ports' data types will be set to `wire`.[4] Verilog requires `input` and `inout` ports to always be of type `wire`.[5] The `output` port, on the other hand, can be specified as either `wire` or `reg`. The syntax for defining vectors (`[W-1:0]`) and arrays will also be discussed in the next section. An important note is that unlike VHDL, Verilog uses 4-state data types (`0`, `1`, `z`, `x`) so we do not need to include external libraries (e.g. VHDL's `std_logic_1164` and `numeric_std`) for this feature.

The description of the module's functionality is carried out between the `module` and **`endmodule`** keywords, after parameter and port declarations. Both of these, as in VHDL's case, can be omitted. The constructs for describing modules will be presented in the next sections.

2.2 Data Types

Each Verilog definition, declaration or assignment is semicolon (`;`) terminated. Thus, line wraps can be used within these.

As was said in the last section, Verilog's data types are usually 4-state: `1` and `0` are the two logical values, while `z` and `x` represent the high-impedance and unknown values respectively.

2.2.1 Scalar Types

The first category of data types is the **`net`**. These types are used to model interconnections between components. As such, a `net` data type must be driven at all times (e.g. through a **`continuous assignment`**), and cannot be driven with **`procedural assignment`**s (i.e. within **`procedure`**s).[6] The most common `net` types are:[7]

- **`wire`**: the default way of representing an interconnection.

[4]In other words, `wire` is the default data type for all the ports.

[5]We can thus omit explicit data type specification for these port types when declaring a `module`.

[6]Both `continuous assignments` and `procedures` will be discussed in the next sections. For now, I will anticipate that `continuous assignments` are similar to VHDL's assignments outside of a process, while `procedural assignments` are similar to VHDL's assignments within a process.

[7]There are more complex `net` types, e.g. for modelling Wired-OR or Wired-AND connections. The full list can easily be found consulting the literature.

- **`tri`**: virtually identical to `wire`, as a convention, used to represent `net`s driven by more than one driver.
- **`supply1`**: represents a `net` connected to V_{DD}.
- **`supply0`**: represents a `net` connected to GND or V_{SS}.
- **`uwire`**: a `wire` that can have only one driver.

As a side note, in Verilog each driver has an associated **`strength`**. The `net` will be driven to the value of the driver with the highest strength (the other drivers will be ignored). If there are drivers, with the highest strength, that try to drive the `net` to different values, the resulting `net` value will be `x`.[8] Verilog will assign default driver strength values (7 for `supply1` and `supply0`, 6 for all the others) so we don't need to do so ourselves manually, unless necessary. Consult the literature for more details on this topic.

The second category of data types is the **`variable`** (or sometimes called **`register`**). These types are used to model abstract storage elements. They will hold the value assigned to them until the next assignment, and can only be driven inside `procedure`s. The `variable` types are:

- **`reg`**: unsigned 1 bit storage (can also be made signed with an arbitrary number of bits).
- **`integer`**: signed (at least) 32 bit storage.
- **`time`**: unsigned (at least) 64 bit storage.
- **`real`**: double-precision floating-point value (IEEE 754-1985).
- **`realtime`**: same as `real`, as a convention, used to represent time.

Now that the basic data types have been introduced, we can discuss some of their details. Both `wire` and `reg` can be made signed by adding the **`signed`** keyword after type declaration (i.e. `wire signed` and `reg signed`). Furthermore, only the `wire`, `reg` and `integer` data types can be directly synthesized. We have previously seen that there are some rules to be followed when declaring a port's type, let's quickly review them:

[8]This holds true for `wire` and `tri`, however, for `net`s that model Wired-ORs or Wired-ANDs, the behaviour is the one you would expect. Consult the literature for more details.

- `input` ports can only be of type `wire`.[9]
- `output` ports can be of type `wire` or `reg`.
- `inout` ports can only be of type `wire`.

There are also rules on what types can drive what ports:

- `input` ports can be driven by a `wire` or `reg` type.
- `output` ports can only be driven by a `wire` type.
- `inout` ports can only be driven by a `wire` type.

Last, let's briefly introduce `parameters`.[10] We have already encountered them in the previous section, and have seen that they represents constant variables. The syntax to declaring a `parameter` (different from the syntax used in the `module` port declarations) is the following:

`parameter {<type>} NAME;`

Where `<type>` is optional and can be either `integer`, `time`, `real` or `realtime`.

See some examples of data types and `parameters` below (Code 2.3).

```
module dtypes_param_example
     ...
    //these are wire declarations
    wire w1, w2;

    //and these are signed wires
    wire signed sw1, sw2;

    //these are reg declarations
    reg r1, r2;

    //and these are signed regs
    reg signed sr1, sr2;

```

[9]To be precise, when we talk about the allowed type of a port or the allowed type of a port driver, I should be using `net` and `variable` instead of `wire` and `reg`. For example, not only `wire` but any `net` type is allowed as the `input` port's type.

[10]There is more to `parameters` than what is discussed here. In fact, they represent a third fundamental Verilog data type. There are ways to redefine `parameters` (e.g. with `defparam`), and there are also different types of `parameters` (e.g. `localparam` and `specparam`). More detail can be found in the literature.

```verilog
    //these are parameter with no explicit type declarations (their type will
    be inferred)
    parameter INTEGER_P = 3,
              REAL_P = 3.1415;

    //and these are parameter declarations with an explicit type
    parameter integer P_OFFSET = 1023,
                      N_OFFSET = -1024;
     ...
endmodule
```

Code 2.3: Verilog Scalar Data Type and Parameter Examples

2.2.2 Vectors and Arrays

In Verilog, vectors represent entities defined by a collection of simple elements of a specific type. As such, they are always one-dimensional. For example, a 4-bit port is a vector. A 32-bit `reg` is also a vector. Arrays, on the other hand, are usually composed of more complex elements. For example, a collection of 4096 32-bit `reg`s. Note that in this example the array represents two-dimensional data. The syntax for both vector and array declarations is the following:

`<type> [MSB_INDEX:LSB_INDEX] NAME;` for a vector.

`<type> [MSB_INDEX:LSB_INDEX] NAME [START_INDEX:END_INDEX];` or

`<type> NAME [START_INDEX:END_INDEX];` for an array.

An example can be found below (Code 2.4).

```verilog
module vector_array_example
     ...
    //these are vector declarations
    wire signed [7:0] h1, h2;

    //this is the second most significant bit of h1
    wire signed smsb = h2[6];

    //this is an array
    reg [31:0] ram [0:4095];

    //this is also an array
    realtime samples [0:1024];

    //this is the second 32-bit reg of ram
    reg [31:0] sram = ram[1];

    //and this is the least significant bit of the last 32-bit reg of ram
    reg llram = ram[4095][0];
     ...
endmodule
```

Code 2.4: Verilog Vector and Array Examples

2.3 Literals and Initialization

Verilog numerical literals are specified using the following syntax:

{<bit_size>}'<base>VALUE

`<bit_size>` is the number of bits used to store the literal. It can be omitted, in which case a default size of (at least) 32 will be set. `<base>` can be one of: **b**, **o**, **d**, **h**, **sb**, **so**, **sd** or **sh**. The meaning of these is obvious but, as an example, `so` stands for signed octal and `d` for decimal. `VALUE` is the number that we want to represent and can also contain the `z` and `x` digits if the base is not decimal. The underscore (_) character can also be used for readability, as long as it is not the first character. We can also prefix the literal with a minus (-) indicating that it is a negative (two's complement) number. Examples of literals and initialization can be found below (Code 2.5).

```
module literal_init_example
     ...
    reg [7:0] r1 = 8'hAA;              //1010 1010
    reg [7:0] r2 = 8'b0101_0101;       //0101 0101
    reg [7:0] r3 = 8'b10;              //0000 0010
    reg [7:0] r4 = 8'd10;              //0000 1010
    reg signed [7:0] r5 = -8'd10;      //1111 0110

    //the literal is truncated after the 3 least significant bits
    reg [7:0] r6 = 3'hFF;              //0000 0111

    //the most significant bit after truncation is interpreted as a sign bit
    and is extended
    reg [7:0] r7 = 3'shFF;            //1111 1111

    //allowed syntax: the literal is interpreted as a signed integer
    reg [7:0] r8 = 18;                //0001 0010
     ...
endmodule
```

Code 2.5: Verilog Literal and Initialization Examples

Explanations of how the minus (-), `<base>` sign, truncation and implicit conversions work are omitted. Details can be found in the literature.

2.4 Operators

Verilog Operators follow a syntax that is very similar to that of C. For this reason I will first introduce the operators familiar to the reader, that is assumed to have basic knowledge of C. Subsequently, I will discuss operators with a Verilog specific syntax.

Verilog shares the following with C:

- Bit-wise operators: `~`, `&`, `|`, `^`.
- Logical operators: `!`, `&&`, `||`.
- Relational operators: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Ternary operator: `?:`.
- Arithmetic operators: `+`, `-`, `*`, `/`, `%`.

Now let's look at Verilog specific operators:

- Bit-wise operators: `~^`.
- Reduction operators: `&`, `|`, `^`, `~&`, `~|`, `~^`.
- Relational operators: `===`, `!==`.
- Arithmetic operator: `**`.
- Shift operators: `<<`, `>>`, `<<<`, `>>>`.
- Concatenation and repetition operators: `{}`, `{{}}`.

The logical operators transform their operands into Boolean values: true (`1'b1`) if the operand has at least one `1` bit, unknown (`1'bx`) if the operand has at least one `z` or `x` bit and not a single `1` bit, and false (`1'b0`) if all the bits of the operand are `0`. The logical operation is then carried out on the Boolean values of the operands, the result itself is a Boolean value. Bit-wise operators, on the other hand, perform the logical operations on the single bits of their operands. They can be unary (as in the case of not (`~`)) or binary (as in the case of and (`&`), or (`|`) and xor (`^`)). The reduction operations are performed on a single operand. It results in the application of the operation on the set of its bits. As such, the obtained result will always be a Boolean value. For example, `~^a` will produce a single bit that is the xnor of all the bits of `a`.

Both `==`, `===` and `!=`, `!==` are used to compare the equality of the operands. The difference is that the `==` and `!=` operators can return unknown (`1'bx`) if either operand contains a `z` or `x` bit. `===` and `!==`, on the other hand, will always compare the bits of the operands (regardless if they are `z` or `x`) and return true (`1'b1`) if they match and false (`1'b0`) otherwise. As an example, both `4'b01zx != 4'b01zx` and `4'b0001 == 4'b000z` will return unknown, while `4'b01zx !== 4'b01zx` and `4'b0001 === 4'b000z` will return false.

Double asterisk (`**`) represents exponentiation, while the two (`<<`, `>>`) and three (`<<<`, `>>>`) angle bracket operators represent the logical and arithmetic shifts respectively. The `{sized_expr_1, sized_expr_2, sized_expr_3, ...}` operator is used to concatenate the comma separated sized expressions inside of it. The `{const_expr{sized_expr}}` will concatenate `sized_expr`, `const_expr` number of times.[11] Consult Code 2.6 for examples on the use of operators.

```
module operator_example
     ...
    reg [3:0] a = 4'hA;           //1010
    reg [3:0] b = 4'h5;           //0101
    reg c;
    reg [19:0] d;
     ...
      a = a ^ b;                  //1111
      c = !a || ~^b;              //1 (!a is 0, ~^b is 1)
      b = c ? b : a;              //0101
```

[11]Sized expressions can be either literals, `nets`, `variables`, vectors or slices of vectors. The literal size must be declared, e.g. `'hFF` is not allowed. Constant expressions must be either literals or `parameters`.

```verilog
     d = {{2{a[3:2], b}}, a};    //0000_11_0101_11_0101_1111 [12]
    ...
endmodule
```

Code 2.6: Verilog Operator Examples

2.5 Continuous Assignments

Verilog has a very similar mechanism to VHDL's assignments for defining concurrent blocks. It is called *continuous assignment.* Just as in VHDL, continuous assignments are defined outside of procedural blocks and can have three characteristics: namely being *simple*, *conditional* or *delayed.* Examples of their syntax can be found below (Code 2.7).

```verilog
`timescale 1ns/1ps

module ca_example
    ...
    reg [7:0] a, b;
    wire [7:0] c;
    reg en, s;

    assign    c = ~(a & b);
    assign #5 c = ~(a & b);
    assign c = (en == 0) ? 8'hZZ :
               (s  == 1) ? a :
               b;
    ...
endmodule
```

Code 2.7: Verilog

```vhdl
entity ca_example is
    ...
end ca_example;

architecture rtl of ca_example is
    ...
  signal a, b, c: std_logic_vector
    (7 downto 0);
  signal en, s: std_logic;
    ...
begin
    ...
  c <= a nand b;
  c <= a nand b after 5 ns;
  c <= (others => 'Z') when en =
    '0' else
       a when s = '1' else
       b;
    ...
end rtl;
```

Code 2.8: VHDL

As can be seen, assignments have a different syntax but function basically the same in both languages. Verilog's simple assignment syntax is:

[12] `a[3:2]` is `11`, `b` is `0101`. `{a[3:2], b}` is then `11_0101` and `{2{a[3:2], b}}` is `11_0101_11_0101`. We then add `a` at the end obtaining `11_0101_11_0101_1111`. Finally, d is a 20-bit number and gets padded with 0s at the beginning.

```
assign LHS = RHS;
```

Where `LHS` must be a `net` type. The syntax for delayed assignments is instead:

```
assign #N LHS = RHS;
```

`N` is an integer that represents the number of time units of the delay.[13] The time unit can be specified with the **`` `timescale unit/precision ``** directive.[14] If the user does not specify one, the default time unit will be used. To obtain conditional assignments a sequence of nested ternary (sometimes appropriately called conditional) operators is used.

2.6 Procedural Statements

Verilog also provides a mechanism that takes place of VHDL's process. This mechanism is called *procedural block*. The latter is a richer construct, compared to the VHDL counterpart, that provides the user with a larger set of features. These will be discussed later in this section, but first, let's look at an example (Code 2.9).

Verilog has two procedural blocks, namely the **`initial`** and the **`always`** blocks. The `initial` block will be executed only once, at the beginning of the simulation. It is not used to model synthesizable behaviour but rather in simulations (e.g. setting up the testbench). The `always` block is executed the same way as VHDL's process (i.e. cyclic execution). In fact, it is Verilog's equivalent of the VHDL process and is used to model synthesizable logic. The execution of an `always` block can be triggered in different ways. The most common one is through the use of an event control. The latter is represented with the at (**`@`**) symbol. The event control precedes the specification of the event(s) that will trigger the `always` block. There are multiple ways of describing events, but here, I will present some of the simplest ones useful to specify the sensitivity list of a procedural block:

`@(A, B, ...)`: the `always` block will be triggered when any of the signals (`nets` or `variables`) appearing between parentheses (i.e. *`A`*, *`B`*, etc.) changes.[15]

[13]There are more advance types of delays (e.g. different falling and rising delay times). For more information consult the literature.

[14]Verilog directives are very similar to C preprocessor directives. The major difference being that C's `#directive_name` has been replaces by Verilog's `` `directive_name ``.

[15]There exists an alternative syntax: `@(A or B or ...)`.

@(*) : the `always` block will be triggered when any of the signals appearing on the right hand side of a procedural assignment, or conditional statement, changes.[16]

@(posedge CLK) : the `always` block will be triggered on the positive edge of the *CLK* signal.[17]

On a side note, it is important to remember that the left hand side of a procedural assignment (i.e. assignment within a procedural block) must be of `variable` type. Below (Code 2.11) you can find some examples of the different procedural blocks and sensitivity lists previously discussed.

```verilog
module proc_example
    ...
    wire [7:0] a, b;
    reg [7:0] c;
    reg d;

    always @(a, b) begin
      c = ~(a & b);
      if (a < b)
        d = 1'b0;
      else
        d = 1'b1;
    end
    ...
endmodule
```

Code 2.9: Verilog

```vhdl
entity proc_example is
   ...
end proc_example;

architecture rtl of proc_example is
   ...
  signal a, b, c: std_logic_vector
    (7 downto 0);
  signal d: std_logic;
   ...
begin
   ...
  process(a, b) begin
    c <= a nand b;
    if a < b then
      d <= '0';
    else
      d <= '1';
    end if;
  end process;
   ...
end rtl;
```

Code 2.10: VHDL

Let's now introduce one of the most important aspects of procedural blocks. So far we have completely ignored it, event though it is crucial in properly modelling sequential and combinational blocks. In VHDL we have only one type of

[16]This should be the default syntax for specifying the sensitivity list of an `always` block. It allows us to avoid many mistakes related to unintentional memory element declaration.

[17]We can use `negedge CLK` to trigger the `always` block on the negative edge instead.

assignment. Verilog, however, has two types of procedural assignments. The first type is called procedural *nonblocking assignment*. It is represented with **<=** (i.e. `LHS <= RHS;`) and is very similar to VHDL's assignment, which follows the same syntax. As in VHDL, a nonblocking assignment is not executed immediately, instead it is scheduled for execution in δ time. As such, the order in which nonblocking assignments appear in a procedural block does not change its behaviour. This type of assignment must be used to model **sequential** blocks. This way we can execute multiple procedural blocks, as well as nonblocking assignments, concurrently without warring about race conditions. The second type of assignment is called procedural *blocking assignment* and is represented with **=** (i.e. `LHS = RHS;`). As the name suggests, the blocking assignment halts the execution flow of the procedural block until the assignment has completed. Afterwards the execution resumes. This type of assignment is much more similar to what you typically find in a general purpose programming language (e.g. C). The closest VHDL equivalence is the assignment to variable (`:=`). The blocking assignment can be used when modelling **combinational** circuits.[18] The designer must be careful, as the order in which the assignments appear does change the block behaviour.

```verilog
module procedural_blocks_example
    ...
    reg [3:0] a, b;
    reg clk, en;

    //will only be executed once, at the very start of the simulation
    initial begin
      a = 4'hA;
      b = 4'h5;
    end

    //MISTAKE: forgot to include en into the sensitivity list
    always @(a, b) begin
      if (en == 0)
        a = b;
      else
        b = a;
    end
```

[18]Sometimes blocking assignments are also used inside sequential blocks (e.g. for temporary variables, i.e. variables used only inside the procedural block), but their use should be deliberate and reasoned. Combinational blocks can use both types of assignments.

```

    //equivalent to @(a, b, en)
    always @(*) begin
      if (en == 0)
        a = b;
      else
        b = a;
    end

    //Verilog model of a register
    always @(posedge clk)
      a <= b;   [19]
     ...
endmodule
```

Code 2.11: Verilog Procedural Block Examples

2.6.1 Conditional Statements

Just as in VHDL, Verilog has the **`if-else`** and **`case`** constructs. The two HDL versions are basically the same, with slight syntactic differences. I will illustrate these with some examples below (Code 2.12 for the `if-else` construct, Code 2.14 for the `case` construct). Verilog has also the **`casez`** and **`casex`** constructs that allow the user to specify "don't care" bit positions. For example, "`2'b?1 : d = 1'b0;`" in Code 2.14 would assign `0` to `d` whenever `s` was equal to `01` or `11`. For more information on these constructs consult the literature.

[19]Do not worry about the meaning of `<=`. It will be discussed later in this section. For now just think of it as an assignment.

```verilog
module if_example
     ...
    wire [7:0] a, b;
    reg [7:0] c;
    reg d;

    always @(*) begin
      if (d == 0)
        c = a;
      else if (d == 1)
        c = b;
      else
        c = 8'hZZ;
    end
     ...
endmodule
```

Code 2.12: Verilog

```vhdl
entity if_example is
   ...
end if_example;

architecture rtl of if_example is
   ...
  signal a, b, c: std_logic_vector
    (7 downto 0);
  signal d: std_logic;
   ...
begin
   ...
  process(a, b, d) begin
    if d = '0' then
      c <= a;
    elsif d = '1' then
      c <= b;
    else
      c <= (others => 'Z');
    end if;
  end process;
   ...
end rtl;
```

Code 2.13: VHDL

```verilog
module case_example
    ...
    reg [1:0] s;
    reg d;

    always @(*) begin
      case (s)
        2'b00 : d = 1'b1;
        2'b01, 2'b10, 2'b11 : d = 1'b0;
        default : d = 1'bX;
      endcase
    end
    ...
endmodule
```

Code 2.14: Verilog

```vhdl
entity case_example is
  ...
end case_example;

architecture rtl of case_example is
  ...
  signal s: std_logic_vector(1 downto 0);
  signal d: std_logic;
  ...
begin
  ...
  process(s) begin
    case s is
      when "00" => d <= '1';
      when "01"|"10"|"11" => d <= '0';
      when others => d <= 'X';
    end case;
  end process;
  ...
end rtl;
```

Code 2.15: VHDL

2.6.2 Loops

Verilog has a variety of constructs to define loops. The basic two are the **`while`** and **`for`** loops. I will not spend time discussing them since both their syntax and functionality are basically identical to the C's counterpart.[20] There are two more loop types. The **`repeat(n)`** loop executes its statements `n` times.[21] The **`forever`** loop repeats its statements indefinitely.[22] As usual, code examples can be found below (Code 2.16). Last, let's briefly digress and introduce some of Verilog's synchronization mechanisms. We can suspend execution of a procedural statement with:

`@(<event>)`: will halt the execution flow of the procedural block until the

[20]The major differences being that in Verilog we cannot use the increment (`++`) and decrement (`-`) operators. Furthermore, the constructs require prior declaration of any variables used in expressions.

[21]It is equivalent to the `for(i = 1; i <= n; i = i + 1)` loop, where `i` in an `integer` defined before the loop, and not used within it.

[22]It is equivalent to the `while(1)` loop.

<event> happens.

wait(<expr>): will suspend the execution flow of the procedural block until *<expr>* becomes true.

#DELAY: will halt the execution flow of the procedural block for *DELAY* time units.

```
module loops_example
     ...
    reg [3:0] a, b;
    reg clk, en;
    integer i;

     ...
      //increment b for each 1 bit of a
      while (a) begin
        if (a[0])
          b = b + 1;
        a = a >> 1;
      end
     ...
      //barrel shift
      for (i = 0; i < 4; i = i + 1)
        a[i] = b[(i + 1) % 4];
     ...
      //generate two pulses on en
      repeat (2) begin
        #10 en = 1;
        #10 en = 0;
      end
     ...
      //make a clock signal of period 20
      forever
        #10 clk = !clk;
     ...
endmodule
```

Code 2.16: Verilog Loop Examples

2.7 Structure and Hierarchy

As in VHDL, one of the most important ways of dealing with the complexity of the design is through hierarchies. As such, the Verilog HDL offers mechanisms for instantiating `module`s designed elsewhere. The way this mechanism works is pretty much identical (except for the syntax) in both languages. Let's take a look at an example to illustrate this point (Code 2.17).

```verilog
module struct_example
     ...
    reg [1:0] s;
    reg d;
    reg [7:0] c;

    cu CU_I (.cu_s(s), .cu_d(d));
    proc #(.N(8)) PROC_I
      (.proc_output(c));
     ...
endmodule
```

Code 2.17: Verilog

```vhdl
entity struct_example is
   ...
end struct_example;

architecture str of struct_example
    is
   ...
  signal s: std_logic_vector(1
    downto 0);
  signal d: std_logic;
  signal c: signed(7 downto 0);
   ...
begin
   ...
  CU_I: entity work.cu(rtl) port
    map (cu_s => s, cu_d => d);
  PROC_I: entity work.proc(rtl)
    generic map (N => 8)
    port map (proc_output => c);
   ...
end str;
```

Code 2.18: VHDL

As you can see, the way that components are instantiated follows the syntax:

`module_name {INST_ID} (.PORT(SIGNAL) {, .PORT(SIGNAL)})`

where `module_name` is the name of the `module` that needs to be instantiated. `INST_ID` is an optional unique name that identifies that particular instance of the `module`. The `.PORT(SIGNAL)` syntax is pretty self explanatory looking at the side by side code.

Verilog, similarly to VHDL, has also a positional way of connecting ports to signals (e.g. in Code 2.17 "`cu CU_I (s, d);`"). It is however less clear and more error prone so should be avoided.

An interesting feature of the Verilog HDL is the presence of **gate-level primitives**. These act as modules defined elsewhere, and available globally, that model the behaviour of basic logic gates. With these, the designer can directly describe the structure of the logic circuit as a gate netlist. The primitives are: **`not`**, **`and`**, **`nand`**, **`or`**, **`nor`**, **`xor`**, **`xnor`**. They accept an arbitrary number of inputs and produce a single output. Examples might be: `"nand NAND_INST1 (out, in1, in2);"`, `"nand NAND_INST2 (out, in1, in2, in3, in4);"`, `"xor XOR_INST1 (out, in1, in2);"`, etc. Furthermore, Verilog allows the users to define their own primitives by specifying their truth table. This mechanism is called *user-defined primitives* (**UDP**). Details can be fount in the literature.

2.7.1 Functions and Tasks

Just as in VHDL, Verilog has constructs for grouping and reusing frequently needed code segments. Verilog's **`function`**s and **`task`**s are similar to VHDL's functions and procedures respectively. There are however some key differences that are now going to be descibed.

A Verilog `function` contains statements that execute in sequence. It has one or more inputs, and returns a single value. Furthermore, `function`s are invoked as **expression** terms, and must be declared within a `module`. You can do so following the syntax:

```
function {automatic} {signed} {<range_or_type>} FUNCTION_ID
(<function_port_list>);
{<block_item_declaration>}
<function_statement>
endfunction
```

The easiest way to understand the above is through examples, see Code 2.19. By default, Verilog `function`s are **`static`**, i.e. there is only one copy of the `function`'s variables that are accessed by `function` calls. This means that `function` calls are not recursive by default, and that different `function` calls can interfere with each other. The optional **`automatic`** keyword makes `function`s non-static. Thus, each `function` call to an `automatic function` has its own local variables.[23] The

[23]On a side note, static `functions` can have their internal variables accessed via *hierarchical references* (e.g. if we have a `module` named `my_module`, and a `function` `my_function` with local

return type of the function is a set to a single bit. You can change that by specifying `<range_or_type>`. The latter indicates the return type of the function, it can be `integer`, `real`, `realtime`, `time`, or a vector range (`[S_INDEX:E_INDEX]`). A vector is by default not signed. You can declare it signed by including the `signed` keyword. A `function` must have at least one `input` port and must have no `output` and `inout` ports. Only blocking assignments are allowed within `functions`, which must not consume any simulation time (i.e. they are not allowed to call the scheduler). Each `function` has an associated variable, with the same name (`FUNCTION_ID`) and same type as the return type. This variable can be manipulated and is automatically return when the function ends.

```
module function_task_example
    ...
    wire [7:0] x, y_1, y_2;
    reg [3:0] u, v_1, v_2;

    //function declaration
    function [3:0] count_f (input [7:0] a);
      integer i;
      begin
      count_f = 0;
      for (i = 0; i < 8; i = i + 1)
        if (!a[i])
          count_f = count_f + 1;
      end
    endfunction

    //task declaration (automatic, otherwise the two enables below would use
    the same parameter set)
    task automatic count_t (input [7:0] a, output [3:0] c);
      integer i;
      begin
      c = 0;
      for (i = 0; i < 8; i = i + 1)
        if (!a[i])
          c = c + 1;
      #5;  //delay the output (mimics latency)
      end
```

variable `my_variable`, we can access `my_variable` from within `my_module` using the syntax: `my_function.my_variable`. There is more to hierarchical references. Seek more details in the literature.). External code cannot however access the internal variables of an `automatic function`.

```
    endfunction

    //function call
    assign x = count_f(u);

    //task enables [24]
    always @(negedge y_1, posedge y_2)
      count_t(y_1, v_1);
      count_t(y_2, v_2);
     ...
endmodule
```

Code 2.19: Verilog Function and Task Examples

Verilog `task`s contain statements that are executed in sequence. Unlike `function`s, they have zero or more inputs and outputs and are invoked as **procedural** statements. `task`s must be declared within a `module` using the syntax:

```
task {automatic} TASK_ID

(<task_port_list>);

{<block_item_declaration>}

<task_statement>

endtask
```

Once again, refer to Code 2.19 for examples. The `automatic` keyword works basically the same way in both `task`s and `function`s, so there is no point in repeating what was already said previously in this subsection. Aside from this similarity, there are a couple further ways in which `task`s and `function`s differ. The first one is that the former can have any number of `input`, `output` an `inout` ports. The second is that `task`s are allowed to invoke the scheduler. This means that both blocking and nonblocking assignments are permitted. In addition to that, `task`s can use delay mechanisms. In fact, you can rewrite as a `function` any `task` that does not invoke the scheduler.

A few things need to be kept in mind when dealing with `function`s and `task`s. First, we have seen that by default these are static. Care must be taken in their design. We must consider whether the `function`/`task` needs a separate set of variables for each call or the ability to call itself recursively. If that is the

[24]`task` calls are usually referred to as `task` *enables*.

case, we must remember to use the `automatic` keyword. Second, it must be noted that `function`/`task` arguments are passed `by value`. This means that when a `function`/`task` is called/enabled, a sample of the input arguments is fed to the `function`/`task`. If we enable a `task` and during it's execution one of the inputs changes, the `task` has no way of knowing that.

2.7.2 Generate Blocks

Verilog has a **`generate`** mechanism very similar to VHDL's counterpart. Used outside procedural blocks, it allows the repeated execution of statements and can be used in conjunction with loops and conditional blocks. Specifically, it can be used with `if-else`, `case` and `for` constructs. Furthermore, a `generate` block can declare a **`genvar`** which is a positive integer value used only inside the block. The `generate` block is wrapped by the `generate`-**`endgenerate`** (optional, but recommended) keywords. Examples follow below (Code 2.20).

```
module generate_example
     ...
    parameter WIDTH = 8;
    reg [WIDTH-1:0] a, b, c, d;
    reg clk, en;

    //instantiate the adder based on the WIDTH parameter that might be
    modified
    generate
      if(WIDTH < 8)
        add_seq #(WIDTH) ADDER (.IN1(a), .IN2(b), .OUT(c));
      else
        add_par #(WIDTH) ADDER (.IN1(a), .IN2(b), .OUT(c));
    endgenerate
     ...
    //xor a and the reverse of b
    genvar i;
    generate
      for(i = 0; i < WIDTH; i = i + 1)
        assign d[i] = a[i] ^ b[WIDTH-1-i];
    endgenerate
     ...
endmodule
```

Code 2.20: Verilog Generate Examples

2.8 Modeling Building Blocks of Digital Design

Now that we have introduced the basics of the Verilog HDL, we can discuss its use in modelling digital electronic circuits. I will do so by presenting some examples of commonly used blocks, namely flip-flops and FSMs. These are presented below (Code 2.21 to Code 2.26), side by side the VHDL counterpart.

There are several features of Verilog that were omitted in this brief introduction. One example is the rich set of **directives** that the language provides. This feature is somewhat advanced and will not be covered. I will use and explain some directives, as necessary, in the next section.[25]

```verilog
module flip_flop_example
  #(parameter integer N = 8)
   (input wire R, CLK,
    input wire [N-1:0] D,
    output reg [N-1:0] Q);

    always @(posedge CLK) begin
      if (R == 0)
        Q <= 0;
      else
        Q <= D;
    end
endmodule
```

Code 2.21: Verilog

```vhdl
library IEEE;
use IEEE.STD_LOGIC;

entity flip_flop_example is
  generic(N: natural := 8);
  port(R, CLK: in std_logic;
         D: in std_logic_vector(N-1
    downto 0);
         Q: out std_logic_vector(N-1
    downto 0));
end flip_flop_example;

architecture bhv of
    flip_flop_example is
begin
  process(CLK) begin
    if CLK'event and CLK = '1' then
      if R = '0' then
        Q <= (others => '0');
      else
        Q <= D;
      end if;
    end if;
  end process;
end bhv;
```

Code 2.22: VHDL

Below you will see the use of `localparam` for the first time. The latter defines a constant value. It works just like `parameter`, except for one property.

[25]Maybe the most important directive that should be known is `` `include ``. The latter works as C's `#include`, copying the code available in another file.

`localparam`s cannot be redefined (e.g. in a hierarchically higher level module) and are thus the preferred mechanism for defining FSM states.

```verilog
module fsm_example
  (input wire R, CLK,
   input wire FSM_IN,
   output reg FSM_OUT);

   localparam A = 1'b0, B = 1'b1;
   reg [1:0] CURR_STATE, NEXT_STATE;
   reg Y;

   always @(CURR_STATE, FSM_IN) begin: COMB
     case (CURR_STATE)
       A: begin
         Y = 1'b0;
         NEXT_STATE = (FSM_IN == 0) ? A : B;
       end
       B: begin
         if (FSM_IN == 0) begin
           Y = 1'b0;
           NEXT_STATE = A;
         end else begin
           Y = 1'b1;
           NEXT_STATE = B;
         end
       end
     endcase
   end
   always @(posedge CLK): SEQ
     if (!R) begin
       CURR_STATE <= A;
       FSM_OUT <= 0;
     end else begin
       CURR_STATE <= NEXT_STATE;
       FSM_OUT <= Y;
     end
endmodule
```

Code 2.23: Verilog

```vhdl
library IEEE;
use IEEE.STD_LOGIC;

entity fsm_example is
  port(R, CLK: in std_logic;
       FSM_IN: in std_logic;
       FSM_OUT: out std_logic);
end fsm_example;

architecture rtl of fsm_example is
  type fsm_state_t is (A, B);
  signal CURR_STATE, NEXT_STATE: fsm_state_t;
  signal Y: std_logic;
begin
  COMB: process(CURR_STATE, FSM_IN) begin
    case CURR_STATE is
      when A =>
        Y <= '0';
        if FSM_IN = '0' then
          NEXT_STATE <= A;
        else
          NEXT_STATE <= B;
        end if;
      when B =>
        if FSM_IN = '0' then
          Y <= '0';
          NEXT_STATE <= A;
        else
          Y <= '1';
          NEXT_STATE <= B;
        end if;
    end case;
  end process COMB;
  SEQ: process (CLK) begin
    if CLK'event and CLK = '1' then
      if R = '0' then
        CURR_STATE <= A;
        FSM_OUT <= '0';
      else
        CURR_STATE <= NEXT_STATE;
        FSM_OUT <= Y;
      end if;
    end if;
  end process SEQ;
end rtl;
```

Code 2.24: VHDL

```verilog
module comp_example
  #(parameter integer N = 8)
   (input wire R, CLK,
    input wire A, B,
    output wire SUCC,
    output reg [N-1:0] COUNT);

    wire C;

    assign C = A ^ B;

    fsm_example FSM (.R(R), .CLK(
    CLK), .FSM_IN(C), .FSM_OUT(SUCC
    ));

    always @(C)
      if (C)
        COUNT = COUNT + 1;

    always @(posedge CLK)
      if (!R)
        COUNT <= 0;
endmodule
```

Code 2.25: Verilog

```vhdl
library IEEE;
use IEEE.STD_LOGIC;
use IEEE.NUMERIC_STD;

entity comp_example is
  generic(N: natural := 8);
  port(R, CLK: in std_logic;
       A, B: in std_logic;
       SUCC: out std_logic;
       COUNT: out unsigned(N-1
    downto 0));
end comp_example;

architecture rtl of comp_example is
  signal C: std_logic;

  component fsm_example
    port(R, CLK: in std_logic;
       FSM_IN: in std_logic;
       FSM_OUT: out std_logic);
  end component;
begin
  C <= A xor B;

  FSM: fsm_example port map(R => R,
     CLK => CLK, FSM_IN => C,
    FSM_OUT => SUCC);

  process(C) begin
    if C = '1' then
      COUNT <= COUNT + 1;
    end if;
  end process;

  process (CLK) begin
    if CLK'event and CLK = '1' then
      if R = '0' then
        COUNT <= (others => '0');
      end if;
    end if;
  end process;
end rtl;
```

Code 2.26: VHDL

2.9 Verilog for Verification

Up to now we have discussed Verilog in general terms. The main constructs and features were introduced, without going into the details of how to actually used them in the most effective way to describe digital systems. A deep dive into these topics would require some time and effort, and are beyond the scope of this brief introduction. That being said, in this section I will discuss in more depth the use of Verilog for verification. Many constructs presented below have been encountered previously. Here, however, I will give a more verification centric overview.

2.9.1 Fundamental Verilog Tools

Verilog provides a variety of mechanisms that are very useful when verifying a design. Let's start with the **`case equality`** (===) and **`case inequality`** (`!==`) operators. These are quite common, since they allow us to also deal with the `z` and `x` bit values. As an example, comparing the values returned by the DUT and the reference model should be done using `case equality/inequality`. This way, event if the return values contain `z` and `x` bits we can compare them. See section 2.4 for a reminder of how these operators work. Next, let's introduce the somewhat rare **`force`** and **`release`** keywords. The former is used to perform a continuous assignment within a procedural block. For this reason it is called **procedural continuous assignment**. Its syntax is *`force LHS = RHS;`*. `LHS` can be either a `net` or a `variable`.[26] The procedural continuous assignment overrides any other assignment until it is released or another procedural continuous assignment is made. To release such an assignment the `release` keyword is used. A `net` subject to a continuous assignment instantly resumes its value. A `variable` retains the `force`d value until its subsequent procedural assignment.

When an identifier is used in a continuous assignment without being previously declared, a `net` is implicitly assumed. This can be an unwanted feature since misspelling a `net` name does not result into an error or warning, rather in the declaration of a new `net`. We can solve this with the **`` `default_nettype ``** directive. In particular, setting it to `none` will disable this implicit declaration allowing us to more easily spot the error.

Testbenches often use events since the latter don't have an associated binary

[26]This is quite distinct from the continuous assignment (can only be done on a `net`) and the procedural assignment (can only be done on a `variable`).

value and don't need to be scheduled. This makes them more efficient than `nets` or `variables`. To declare one use the ***event** EVENT_NAME;* syntax. An event can be triggered with `->` (i.e. `-> EVENT_NAME`) and caught with `@` (i.e. `@ (EVENT_NAME)`). Events are not synthesizable.

A statement that we have previously encountered and is very useful for verification is **`wait`**`(<expr>)`. If `<expr>` is true the `wait` will be ignored as if it wasn't there, otherwise the execution flow will be blocked until `<expr>` becomes true. Another one is the `initial` block which is executed only once at the very beginning of the simulation. It is generally the block that defines the testbench execution flow.

Verilog provides a mechanism for concurrency. The **`fork-join`** block defines a set of statements, each subject to its own timing controls, that are executed concurrently. When the execution flow reaches this block, each statement within it start executing. The execution of the block completes when all the forked statements have completed.

The last statement that I will introduce in this subsection is **`disable`**. The latter can be used as a C style break to terminate the execution of named blocks or `tasks`.[27]

Since this subsection is fairly dense I have provided examples (Code 2.27) to illustrate the above.

```verilog
`default_nettype none

module verification_tools_example;
  reg a, b;
  wire c_dut, c_ref;
  event mismatch;

  xor DUT(c_dut, a, b);

  /* assign c_raf = a ^ b;
   * ERROR: misspelled c_ref, detected with `default_nettype set to none
   * without this directive this statement is equivalent to:
   * wire c_raf;
   * assign c_raf = a ^ b;
   */
```

[27]Named blocks are simply blocks that have an associated label. To associate a label to a block simply write ": LABEL_NAME" after the `begin` keyword of the block (or after where the keyword would be, if the block does not have one).

```

assign c_ref = a ^ b;

initial begin
  fork
    //stimulus generating block, executed in parallel with...
    begin
      #40;
      a = 1’b0;
      b = 1’b0;
      wait(c_dut == 0);  //wait for the signal to settle
      #10;
      a = 1’b0;
      b = 1’b1;
      wait(c_dut == 1);
      #10;
      a = 1’b1;
      b = 1’b0;
      wait(c_dut == 1);
      #10;
      a = 1’b1;
      b = 1’b1;
      wait(c_dut == 0);
      #10;
    end
    //...this named block (with name check_mismatch) that forces a mismatch
    begin: check_mismatch
      force c_dut = 0;  //continuous assignments to wires
      force c_ref = 1;  //within a procedural block
    end
  join
end

/* truth table for the inequalities (equalities are easily derived)
 * | !=  c_dut: 0  1  x  z | !==  c_dut: 0  1  x  z |
 * | c_ref:                | c_ref:                 |
 * |     0      0  1  x  x |     0       0  1  1  1 |
 * |     1      1  0  x  x |     1       1  0  1  1 |
 * |     x      x  x  x  x |     x       1  1  0  1 |
 * |     z      x  x  x  x |     z       1  1  1  0 |
 */
always @(c_dut, c_ref)
  if (c_dut !== c_ref)
    -> mismatch;    //trigger the event
```

```verilog

  //catch the event; another named block (with name display_mismatch)
  always @(mismatch) begin: display_mismatch
    $display("Mismatch: t=%f, DUT=%b, Reference=%b", $realtime, c_dut, c_ref);
    disable check_mismatch;  //stop generating the mismatch
    release c_dut;  //release the value so that they can be driven
    release c_ref;  //by the dut and reference model
  end
endmodule
```

Code 2.27: Verilog Fundamental Verification Tools Example

2.9.2 Useful System Tasks and Functions

In the previous example (Code 2.27) you can see that I have used `$display()` without really explaining it. The latter is a system `task`. This section will go into some detail about these.

The first family of system `tasks` is composed of the following: **`$display()`**, **`$write()`**, **`$strobe()`** and **`$monitor()`**. All of these allow the user to print information to the standard output. The difference between them will be discussed below. There are multiple versions of these `tasks` and multiple ways/stiles of using them.[28] One such style, that should be quite familiar and I suggest using, is the C's `printf()` way of displaying information. For example, you can see in Code 2.27 that the `$display() task` will print `"Mismatch: t=<RT_F>, DUT=<C_DUT_B>, Reference=<C_REF_B>"`, where `<RT_F>` is the floating-point representation of `$realtime`, while `<C_DUT_B>` and `<C_REF_B>` are the binary representations of `c_dut` and `c_ref` respectively.[29] Here is the list of all the formats:

- `%b` or `%B` for binary. E.g. `$display("%b", 4'b1010);` will print `"1010"`.
- `%o` or `%O` for octal. E.g. `$display("%o", 4'b1010);` will print `"12"`.
- `%d` or `%D` for decimal. E.g. `$display("%d", 4'b1010);` will print `"10"`.[30]

[28]Each of these `tasks` is by default associated with the decimal format. This means that, when printing signals, they will appear as decimal numbers. There are also versions that have different default formats (e.g. `$displayb()`, `$displayo()` and `$displayh()` will display signals in the binary, octal and hexadecimal form respectively). More information on these can be found at [1]. There are other caveats to these `tasks` that are rarely encountered and are beyond the scope of this subsection.

[29]`$realtime` is a system `function` (with no arguments) that returns the current simulation time.

[30]To eliminate the leading spaces use `%0d` instead of `%d`.

- `%h` or `%H` for hexadecimal. E.g. `$display("%h", 4'b1010);` will print `"a"`.
- `%f` or `%F` for float. E.g. `$display("%f", 3.14);` will print `"3.14"`.
- `%e` or `%E` for exponential. E.g. `$display("%e", 3.14);` will print `"3.14e+00"`.
- `%c` or `%C` for ASCII character. E.g. `$display("%c", 86);` will print `"V"`.
- `%s` or `%S` for strings. E.g. `$display("%s", "hello");` will print `"hello"`.[31]
- `%t` or `%T` for time. E.g. `$display("%t", $time);` will print `"3.6 ns"`.[32]

There are some further ways to use the above system `tasks`. For example, `%l` or `%L` can be used to print the library binding of the `module`. Analogously, `%m` or `%M` can be used to print the hierarchical name. These are somewhat advanced features and are left here mainly for reference. See also `%v`, `%V`, `%u`, `%U`, `%z`, `%Z` in the literature if interested in more details.

Verilog supports a set of escape sequences that are very similar to C's. They are:

- `\n` for a new line.
- `\t` for a tab.
- `\\` for a \ character.
- `\"` for a " character.
- `%%` for a % character.

Last, before moving to a different family of system `tasks`, let's discuss the difference between `$display()`, `$write()`, `$strobe()` and `$monitor()`. `$write()` works just as C's `printf()`, displaying its contents to standard output. `$display()` will also append a new line character at the end. Unlike these two, `$strobe()` is not executed immediately. It is instead scheduled to display at the end of the current time step. `$monitor()` is similar to `$strobe()`, in that it is executed at

[31] Verilog allows the definition of string literals and their assignment to `regs`. As an example, we need 5 bytes to store `"hello"` since each character is an ASCII encoded byte. Thus, `reg[4:0] string = "hello";` can be used to define and store a string for later use (e.g. in a `$display()` enable).

[32] `$time` is a system `function` that returns the current simulation time (as a `time` data type). In this example the simulation time is assumed to be 3.6 ns. Note that unlike `$time`, `$realtime` returns the current time as a real. You can use the `$timeformat` system `task` to set the format in which `%t` is displayed.

the end of the time step. Unlike `$strobe()` however, once `$monitor()` has been invoked, it constantly monitors its arguments. If it detects a change in one of them, except for `$time`, it acts just as a `$strobe()` enable. Only one `$monitor() task` is active at any given time. Thus, each time a new `$monitor()` is invoked, the previous one ceases its execution. You can disable and enable monitoring with the `$monitoroff` and `$monitoron` system `tasks`.

The second family of system `tasks` is related to file I/O. Some of these take inspiration from C's standard library functions. You can open a file with:

> *`$fopen(FILE_NAME)`* (e.g. `$fopen("samples.txt");`), or
>
> *`$fopen(FILE_NAME, <mode>)`* (e.g. `$fopen("samples.txt", "w");`).

Using the first syntax, the `function` returns a 32-bit unsigned integer called multi-channel descriptor or **MCD**. The most significant bit is reserved and always set to `0`. Each of the other 31 bits represents a unique flag, and is associated to a particular opened file. The MCD with value 1 is reserved for standard output. Thus, in total, we can have 30 files open simultaneously this way. The system `function` returns 0 if it is not able to open the file. To close a file use:

> *`$fclose(MCD_NAME);`* (e.g. `file = $fopen("samples.txt"); $fclose(file);`)

You can write to files, via MCDs, using the **`$fwrite()`**, **`$fdisplay()`**, **`$fstrobe()`** and **`$fmonitor()`** `tasks`. These are analogous to their counterpart without a leading `f`, except that they require an MCD as their first argument (e.g. `$fdisplay(file, "Time = %t", $time);` instead of `$display("Time = %t", $time);`). You can simultaneously write to multiple files by writing to the bit-wise or of their MCDs (e.g. `$fdisplay(file1 | file2, "Time = %t", $time);`)

You can read via MCDs by using:

> *`$readmem<f>(FILE_NAME, ARRAY_NAME, [START_ADDRESS, [END_ADDRESS]]);`*
> (e.g. `$readmemb("samples.txt", samples_array, 5, 20);`)

`<f>` represents the format and can be either `b` (binary) or `h` (hexadecimal). `FILE_NAME` and `ARRAY_NAME` specify the name of the file and the name of the array where the file contents will be stored. `START_ADDRESS` is an optional argument that indicates the starting index of the array where file data should be stored. If it is specified, you can also indicate the end index of the chunk to be read with `END_ADDRESS`. You can add comments (using the `//` or `/**/ syntax`) inside the file (e.g. inside `sample.txt`) that will be ignored by `$readmem<f>`. Furthermore, you can specify the array index where the next line of the file should be stored with `@HEX_INDEX`. For example, let `sample.txt` contain

```
...

//this comment will be ignored

@FF

0110_1001

...
```

then `$readmemb("samples.txt", samples_array);` will store `01101001` at index `FF` (255 in decimal) of the `samples_array` array.

The other way of opening files (`$fopen(FILE_NAME, <mode>)`) takes inspiration from C. In fact `<mode>` works in the same way. With `"r"`, `"w"`, `"a"`, `"r+"`, etc. indicating whether the file should be read-only, overwritten, appended or read and written. When calling this `function` a 32-bit unsigned integer, called file descriptor or **FD**, is returned. The most significant bit being reserved and always set to `1`. Each combination of the remaining 31 bits uniquely represents a file. Thus, we can have 2^{31} distinct files open simultaneously. The FDs 0, 1 and 2 are reserved for standard input, output and error respectively, and are pre-opened. The above `functions`/`tasks` for closing, writing and reading work analogously for files represented by FDs. Besides these, file descriptors allow other ways to read/write to files that are very similar to C's. For example, Verilog provides the **`$fgetc(FD_NAME)`**, **`$fgets(STRING, FD_NAME)`**, **`$fflush([FD_NAME])`**, **`$fscanf(FD_NAME, <format>, <args>)`**, **`$feof(FD_NAME)`**, etc. `tasks`.[33] If not familiar with these or if a refresher is needed, consult the literature.

Other useful system `functions`/`tasks` are:

- **`$realtobits(REAL)`** : will convert the `real` value `REAL` into a binary vector.
- **`$bitstoreal(BITS)`** : will convert the bit vector `BITS` into a `real`.
- **`$rtoi(REAL)`** : will convert the `real` value `REAL` into an `integer`.

[33] `<format>` and `<args>` follow the same syntax as the writing `tasks` (e.g. `$fscanf(file, "Input_1 = %b, Input_2 = %b", a, b);`)

- **`$itor(INT)`** : will convert the `integer` value `INT` into a `real`.
- **`$dumpfile(FILE_NAME)`** : will store the value-change data (**VCD**), indicating the signal definitions and their changes, into the file named `FILE_NAME`.[34] This **`task`** will only dump the VCD of signals that have been selected, to select signals use...
- **`$dumpvars`** (no arguments): indicates that the VCD of all the signals should be dumped.[35]
- **`$stop`** (no arguments): stops the simulation and enters interactive mode.
- **`$finish`** (no arguments): terminates the simulation.

As usual, to clarify the presented notions, consult the examples (Code 2.28).

```verilog
module sys_funcs_tasks_example;
  $dumpvars;
  $dumpfile("simulation_data.vcd");

  reg a, b;
  wire c_ref, c_dut;
  integer simulation_file, stimulus_file;

  and DUT(c_dut, a, b);

  assign c_ref = a & b;

  initial begin
    stimulus_file = $fopen("stimulus.txt", "r");
    /* suppose the stimulus file (stimulus.txt) to simply contain:
     * 0 0
     * 0 1
     * 1 0
     * 1 1
     */

    simulation_file = $fopen("simulation_report.txt", "w");
    $fmonitor(simulation_file, "Time: %t\n-%b AND %b = %b", $time, a, b, c);
```

[34]See more ways to manipulate VCD dumps in the literature. Furthermore, you could look into the extended value-change data (EVCD).

[35]There are ways of selecting just an arbitrary subset of signals. See the literature if interested.

```
    /* will print something like:
     * Time: 0 ns
     * -0 AND 0 = 0
     * Time: 10 ns
     * -0 AND 1 = 0
     * ...
     */

    $display("Beginning stimulus retrieval");
    while(!$feof(stimulus_file)) begin
      $fscanf(stimulus_file, "%b %b", a, b);
      #10;
    end
    $display("TEST FINISHED");

    $fclose(simulation_file);
  end

  always @(c_ref, c_dut)
    if (c_ref !== c_dut) begin
      $fdisplay(2, "ERROR: mismatch");  //print the mismatch to stderr
      $stop;    //enter the interactive mode if a mismatch has been detected
    end

endmodule
```

Code 2.28: Verilog System Functions and Tasks Example

2.9.3 Testbenches

Testbenches can have different degrees of complexity. A simple one will generally just stimulate the DUT in a deterministic user specified way. It will not interact with the design, nor simulate the operational environment. Furthermore, DUT output is analyzed post simulation. A sophisticated testbench on the other hand will:

- Model and simulate the operational environment of the DUT.
- Check and analyze DUT output during simulation.
- React dynamically to the design, based on its response, and communicate with it.

- Generate complex randomized stimuli.

Quite often testbenches require the definition of clocks to sync up the different DUTs. Code 2.29 provides examples of how one might go about doing so.

```
module clock_example
  ...
  localparam PERIOD = 10;
  localparam HALF_PERIOD = 7;
  localparam DUTY_CYCLE = 6/10;
  localparam DELAY = 2;
  reg clk_1, clk_2;
  ...
  //waits 2 time units before oscillating, is 1 60% of the time
  initial: clocking_block_1
    #(DELAY) forever begin
      #(PERIOD*DUTY_CYCLE) clk_1 = 1;
      #(PERIOD*(1-DUTY_CYCLE)) clk_1 = 0;
    end

  //simple clocking block
  //always does not have an event control (@) thus will always trigger
  always: clocking_block_2
    #(HALF_PERIOD) clk_2 = ~clk_2;
  ...
endmodule
```

Code 2.29: Verilog System Clock Examples

Another useful tool in testbenches is the Verilog (pseudo) random number generator. It is a `task` that takes in input and outputs (via the same `inout` port) an optional 32-bit integer called **seed**. The seed sequence allows us to track and re-generate the same (pseudo) random numbers. Each `task` enable returns a (pseudo) random 32-bit signed integer. The syntax is:

`$random[(seed)]`.

This `task` will return one of the possible 32-bit signed integers, picked uniformly at random. You can change the distribution of the returned (pseudo) random numbers by using one among the following instead:

- **`$dist_uniform(seed, start, end)`** for the uniform distribution.
- **`$dist_normal(seed, mean, std)`** for the Gaussian distribution.

- **`$dist_poisson(seed, mean)`** for the Poisson distribution.
- **`$dist_exponential(seed, mean)`** for the exponential distribution.
- **`$dist_erlang(seed, k ,mean)`** for the Erlang distribution.
- **`$dist_chi_square(seed, dgf)`** for the χ^2 distribution.
- **`$dist_t(seed, dgf)`** for the Student's t-distribution.

We will look more into randomized stimuli generation in the next chapter dedicated to SystemVerilog. This is due to the fact that the latter has much more powerful tools for this task. The topic of randomized stimuli generation is quite an important one. Oftentimes randomized inputs can detect errors in one of the most critical parts of a design: edge cases.

This concludes our brief introduction to the Verilog HDL. Before moving on to SystemVerilog however, I would like to leave some additional references to more advance topics that were not covered. Furthermore, I will mention some terminology and concepts frequently used in verification. Additional topics:

- Getting command line values with the **`$test$plusargs()`** and **`$value$plusargs()`** system `functions`.
- **Hierarchical names**.
- **Test configuration methods**.
- Programming Language Interface (**PLI**).
- **Configurations** and **libraries**.

Some useful verification notions:

- Technological advancements increase the complexity of the design more and more. Higher complexity requires higher verification effort which, to be the most effective, requires the design and verification teams to be distinct (this property is sometimes called **verification integrity**). To achieve the best verification results a plan must be devised.
- The **Design Verification Plan** is a crucial step in the verification process that ensure the best results. It specifies the used technologies, how to test in accordance with the functional specifications, the scheduling of the design and verification to ensure synchronicity, and more.

- The goal of the Design Verification Plan is **progressive testing**. It usually starts with high-level system design verification to endure the correctness of the design before implementation. Afterward, the component and subsystem implementations, as well as the hardware/software interaction, are verified.
- We can asses the state of the verification process through coverage metrics. **Code coverage** indicates the amount of "code lines" that has been verified (e.g. what code blocks, FSM states and transitions, expression terms, etc. have been encountered during simulation). **Functional coverage**, on the other hand, indicates what specific values, ranges of values and transitions between values have occurred (**data oriented coverage**) or the sequence of control signals that have occurred (**control oriented coverage**). SystemVerilog has some fairly powerful tools to deal with functional coverage where as Verilog has none.
- A "**sweep**" test is a test where the same stimulation patterns are used in multiple runs. What changes from run to run is instead the relative timing these patterns arrive at.
- A **regression test** is usually a fully automated test (e.g. through `tcl` scripts) that verifies the integrity of the design after the addition or modification of some features.

Chapter 3

SystemVerilog

SystemVerilog (SV) is an extension of the Verilog HDL. In fact, it is a super set. As such, any Verilog code is also fully compatible with SV. First introduced in 2002 and later standardized by IEEE as IEEE 1800-2005, SystemVerilog is a HDVL (Hardware Description and Verification Language) as it combines both features of a HDL as well as a HVL (Hardware Verification Language) in a single language. The latest standard is the IEEE 1800-2017. On the HDL level SystemVerilog introduces many improvements to Verilog. Enhanced data types, procedural blocks, constructs, etc. allow for more readable, less error prone and more efficiently synthesizable RTL that can be developed at a higher level of abstraction with less code. On the HVL side, SystemVerilog has extensive features for developing layered testbenches (based on OOP), for coverage-driven verification, constrained random verification and assertion-based verification.[36] SV is a truly vast language and its full description is beyond the scope and capability of this work. Here I will give an overview of the language, presenting both HDL and HVL constructs, with an emphasis on the latter. Many of the features will be presented at a higher level, without spending too much time on technicalities, since they are borrowed from other languages (e.g. Java) that the reader is assumed to be familiar with. For an online introduction to SystemVerilog consult [10] [11]. For a more thorough dive into the HDL and HVL sides of SystemVerilog

[36]Layered testbenches are testbenches that allow verification at different levels of abstraction. They usually have some common structure and will be explored in the next chapter when we introduce UVM. Constrained random verification and coverage-driven verification go hand in hand. In the former DUT stimulus is generated in a random way according to some constraints. In the latter the verification progress is monitored based on functional coverage and stimulus is generated to try and improve it. Last, assertion-based verification (ABV) uses assertions to improve the verification process. Randomized stimulus, coverage and assertions will be covered in later sections.

see [12] and [13] respectively. Cadence [4] also offers excellent material on this topic.

3.1 Data Types

The first key improvement that SystemVerilog makes over Verilog is the introduction of a new data type that replaces its `net`s and `register`s. This new data type is **`logic`** and can be used in place of the other two, making it similar to VHDL's `signal`. You no longer have to remember the sometimes confusing rules for declaring ports and signals as `net`s or `register`s. The only limitation of this new data type is that it can be driven by at most one source. If you need a data type that can have multiple drivers you need to use Verilog's `net`s.

SystemVerilog introduces 2-state (1 and 0) variables, namely: **`bit`**, **`byte`**, **`short`**, **`int`** and **`longint`**. These are represented with 1, 8, 16, 32 and 64 bits respectively. The 2-state data types are not very useful for design but, since they require less memory and are more efficient to work with, are extensively used in verification. Some care must be taken when converting from 2-state to 4-state data types and vice versa. When converting from 4-state to 2-state the `X`s and `Z`s will be converted to a 2-state 0 and information will be lost. A 2-state 0 clearly gets converted into a 4-state 0.

3.1.1 Arrays

In SystemVerilog Arrays have two main attributes. They can be **packed** or **unpacked**, **fixed size** or **dynamic**. Packed arrays can be interpreted as both a single scalar as well as a set of value. For example, if we want to have a data element that we can treat both as a byte and a set of 8 bits, we could use packed arrays. This way we could assign it values directly, or work on the bit level. These types of arrays are specified by providing the array range before the array name. Unpacked arrays, on the other hand, are seen only as a collection of data and not as a single entity. As such, you cannot assign a value directly to an unpacked array. These types of arrays are specified by providing the array range after the array name. You can declare arrays to have both packed and unpacked dimensions, furthermore, you can specify the array range using the **`[size]`** syntax with is equivalent to `[0:size-1]`. Fixed size arrays are, as the name suggests, arrays whose size was declared once and cannot change. Dynamic arrays, on the other hand, are unpacked arrays that do not have a defined size as the user can change it during run time. For an illustration of these types of arrays consult the code below (Code 3.1). Find out more technicalities in the literature.

```
module arrays_example
   ...
  //this is a packed array
  bit [7:0] my_byte_packed;

  //this is an unpacked array
  bit my_byte_unpacked [8];

  //this array has both packed and unpacked dimensions
  bit [3:0][7:0] ram [1024][32];

  //this is a dynamic array
  bit bit_dynamic[];

     ...
    my_byte_packed[3] = 1'b0; //legal
    my_byte_packed = 8'hFA;   //legal

    my_byte_unpacked[3] = 1'b0; //legal
    my_byte_unpacked = 8'hFA;   //illegal

    ram[1023][31][3] = 8'hAA;   //note the proper dimension order

    bit_dynamic = new[8];   //new array of size 8
    foreach(bit_dynamic[i])
      bit_dynamic[i] = i;

    //copy the old 8 bit array into a new 16 bit array
    bit_dynamic = new[16](bit_dynamic);
     ...

   ...
endmodule : arrays_example
```

Code 3.1: SystemVerilog Array Examples

3.1.2 Queues, Associative Arrays and User Defined Data Types

SystemVerilog provides some build in data structures of common use. Queues are doubly linked-lists where, thanks to references to all the elements of the list, you can access any element in constant time without the need to sequentially scan the entire list. Adding elements at the head or tail of the list is computationally efficient, however, adding elements close to the middle takes linear time. Associative arrays are maps, often used to model large memories that are accessed sparsely. See Code 3.2

```
module qs_and_aas_example
   ...
  int i;

  //this is a queue
  int q[$];

  //this is an associative array with key int and value byte
  byte aa[int];

     ...
      q.push_front(1);  //add element to the front of the queue
      i = q.pop_front;  //remove element from the front of the queue

      //initialized the associative array
      do begin
        aa[i] = i;
        i <<= 1;
      end while (i > 0);

      //iterate over the elements of the associative array
      foreach(aa[i])
        aa[i]--;
     ...
   ...
endmodule : qs_and_aas_example
```

Code 3.2: SystemVerilog Queue and Associative Array Examples

SystemVerilog carries over many of the constructs of the C programming languages for defining custom data types. **`typedef`**, **`struct`**, **`union`** and **`enum`** are all common to both C and SV. These work basically the same with marginal differences. Later on we will see that SV also supports defining custom data types using the OOP principles, i.e. through classes. Data Types in SystemVerilog is an extensive topic, here I have provided but the mention of the existence of these language features. Still, the working principle of most of these should be fairly familiar to a reader with prior knowledge of C. For the technicalities, such as the syntax, consult the references provided at the beginning of this chapter.

3.2 Operators

SV adopts many of the modern short-cut operators from other programming languages. For example, `+=`, `^=`, etc. which have the familiar meaning. The increment (`++`) and decrement (`--`) operators find their way into SystemVerilog, both in their pre and post forms.

New operators, specific to HDLs get introduced. For example the **`==?`** and **`!=?`** treat the `Z` and `X` values of the RHS of the operators as don't care conditions. This results in, for example, `4'b1010 ==? 4'b1XZ0` being true. Another one is the **`inside`** operator which follows the syntax:

`expression inside {range {, range}}`

with *`range`* being either an expression or a range of type `[start:end]`. The value of this operator is true if the expression is contained inside the range and false otherwise. For example, "`my_signal inside {0, [2:4], 7, [9:11]}`" is true only when `my_signal` takes the values 0, 2, 3, 4, 7, 9, 10, 11.

3.3 Procedural Statements and Blocks

Unlike in Verilog, SV allows you to put the named `begin...end` block's label also at the end of block. Furthermore, you can add labels on keyword that identify the end of a standard block (e.g. `endmodule`) for improved readability. See the code below (Code 3.3) for an example.

You can declare the `for` loop's variable directly in the loop (as is done for example in C) without needing to declare it in a preceding named block. In fact, SystemVerilog allows the user to define variables even inside unnamed blocks. The conclusion is that in SystemVerilog the familiar syntax *`for (int i = 0; ...; ...)`* is now legal and works pretty much the same way as it does in other programming languages. Additionally, SystemVerilog supports the **`foreach`** loop that can be found in languages such as Java. An important note is that if we want to iterate over multiple variables of an array, the correct syntax is *foreach(array[i, j])* and not *foreach(array[i][j])*. Further loop constructs, found in other programming languages, that make their way to SV are the **do...while** loops and the **break** and **continue** statements.

```
module named_blocks_example
  ...
  begin : my_block
    ...
    begin : my_block2
      ...
    end : my_block 2
    ...
  end : my_block
  ...
endmodule : named_blocks_example
```

Code 3.3: SystemVerilog Named Block Examples

SystemVerilog adds the **priority** and **unique** modifiers for the if-else and case statements. The former modifier generates a simulation time warning if no matching branch is found, the latter if a number of matching branches different than one is found. Note that using default and else in case and if-else statements makes the priority modifier redundant as there will always be at least one matching branch.[37]

SystemVerilog introduces constructs that allow for a more clear and less error prone modeling of combinational logic, registers and latches. These are based on Verilog's always block and are: **always_comb**, **always_ff** and **always_latch**. These should always be preferred to Verilog's alternative. Find out more about these HDL features in the literature.

Below you can find a code example (Code 3.4) that uses some of the introduced constructs.

```
module procedural_constructs_example
  ...
  bit [7:0] mat [8][8];
  logic [1:0] sel;
```

[37]You can find out more about these modifiers in the literature and, in particular, how they map to Verilog constructs.

```
  logic [31:0] a;
   ...
  foreach(mat[i, j])
    Mat[i][j] = 10*i + j;
   ...
  unique casez (sel)
    2’b1?: a = ’1;
    2’b?1: a = ’0;
  endcase
   ...
endmodule : procedural_constructs_example
```

Code 3.4: SystemVerilog Procedural Constructs Examples

3.4 Tasks and Functions

There are a few changes to `tasks` and `functions` that improve their usability. First, to reduce verbosity, remember you do need to use the `begin...end` keywords to delimit `functions` and `tasks` as in Verilog. Instead you can simply use the `function...endfunction` and `task...endtask` keywords to delimit these. This fact was already mentioned in a previous section, together with the fact that you can put the block label at its end and SV will check for consistency. That being said, the first major addition of SystemVerilog to `tasks` and `functions` are the **`void functions`**. These are specified by declaring `void` as their return type. As the name suggests, these `functions` are not expected to return any value directly.[38] You can specify to SV that the return value of a function should be ignored by casting it to `void`. See details in the code example below (Code 3.5). Even though `function` and `task` arguments have a default type and direction, you should never rely on that. Always explicitly declare both the direction, as well as the type, of an argument. Failing to do so can often lead to hard to debug errors.

SystemVerilog allows you to specify the direction of a `function` argument to be `input`, `output` or `inout`. This extends Verilog’s `function`, making this construct a general-purpose, synthesizable routine without timing commands.

SystemVerilog also allows you to define a default value for `function` and `task` arguments. See Code 3.5 for an example. Furthermore, you can use the **`return`** statement as in other programming languages. This allows you to have more

[38]They might still return values through output ports as we will see later.

control over when the `functions` and `tasks` terminate their execution and the value they return.

Remember that Verilog passes argument by value. In SystemVerilog you can specify that you want to pass an argument by reference instead. This features works similarly to C++'s passing by reference. In particular, any argument passed by reference will be transparent to that `function` or `task`. This means that, if a passed by reference argument is modified outside the `function` or `task`, this change will be visible immediately inside the subroutine. Analogously, changing an argument, that was passed by reference, inside the `function` or `task` body, will make the change detectable immediately outside. This contrasts Verilog where any changes to the output arguments was visible only at the end of the subroutine. You can pass argument by reference only to `automatic function` and `task`. Additionally, only `variables` and not `nets` can be passed this way. To specify that an argument is passed by reference use the **`ref`** keyword in place of the argument direction. If you do not want the passed by reference argument to be modified use **`const ref`** instead. See Code 3.5 for an example.

```
module functions_tasks_example
   ...
  int i;

  //the function returns void but can modify the out argument
  //default value for in is 1
  function void my_void_function(input int in = 1, output int out);
     ...
    return;
     ...
  endfunction : my_void_function

  function int my_int_function(input int in);
     ...
    return 1;
     ...
  endfunction : my_int_function

  //both argument passed by reference, the first one (in) cannot be modified
  task automatic my_task(const ref int in, ref int out);
     ...
  endtask : my_task
   ...

  //ignoring the return value by casting to void
  void'(my_int_function(.in(i)));

  //equivalent to my_void_function(1, .out(i));
  my_void_function(, .out(i));
   ...
endmodule : functions_tasks_example
```

Code 3.5: SystemVerilog Functions and Tasks Examples

3.5 OOP in SystemVerilog

SystemVerilog supports the Object Oriented Programming (OOP) paradigm. It's implementation takes inspiration from different programming languages such as C++ and Java. This feature is particularly useful for building testbenches that allow verification at different levels of abstraction. Many of the topics presented here should already be familiar to the reader, that is assumed to have experience in the aforementioned C++ and Java. As such, many technicalities will be omitted.[39]

As in other programming languages that support OOP, the central construct that allows this paradigm in SV is the **`class`**. The latter can contain data and methods and can be declared in `module`s, `interface`s, `package`s or other design blocks. It is defined, similarly to other blocks in SV, using the **`class-endclass`** keyword pair. For an example see Code 3.6. `class`es can contain one, and only one, constructor method. This method's name is always **`new()`** and is a function without a return type. The constructor supports default argument values.[40] When instantiating an object of a given class, use the constructor method `new()`. This is quite different from other languages, such as C++, where the constructor has the `class` name and `new` is a keyword for memory management. On the topic of memory management, SV has automatic garbage collection, similarly to Java, so the user does not need to worry about manually allocating and deallocating memory.

When you define an object handle, without initializing it, SV automatically sets the handle to `null`.

[39] Also, some more advanced features of SV, like generic programming through parameterized classes, are omitted. You can find these topics in the provided reference material.

[40] SystemVerilog does not support method overloading in general but, as we have seen, supports default argument values.

Subroutines can be declared **`external`**, which means that the `class` contains only the subroutine declaration, the definition will be found outside the `class`. When defining an `external` subroutine you must match the declaration signature, but also include the `class` name with the scoping operator (::). This means that, for example, an `external` method called `my_method` declared in `my_class` should be implemented using the name `my_class::my_method`. See the example below (Code 3.6).

SystemVerilog `class`es also support `static` variables and methods. They work similarly to other programming languages, with `static` variables being common to all (0 or more) instances of the class. `static` methods, similarly, are not specific to an object, but rather to all (0 or more) objects of that class. As such, `static` methods can only refer to `static` variables and other `static` methods.

```
module oop_example
  ...
 //class definition
 class my_class;
   logic [15:0] data;

   //all of the objects manipulate this same variable since its static
   static int num_objects;
    ...
   //constructor: always function with 'new' as name
   function new(data = 16'h0000);
      ...
   endfunction
    ...
   task t1( ...);
      ...
   endtask
    ...
   static task t2( ...);
      ...
   endtask
    ...
   //external function declaration, will be defined outside the class
   external function int f1( ...);
    ...
 endclass
  ...
 //definition of an external function
 function int my_class::f1( ...);
    ...
 endfunction
  ...
 //creating an object handle: initialized to null
 my_class object;

 //creating new object, equivalent to: object = new(16'h0000);
 object = new();

 //calling a class method
 object.t1( ...);

 //calling a static method
 my_class::t2( ...);

 //directly accessing a non-encapsulated class variable (bad practice)
 object.data = 16'bFFFF;
  ...
endmodule : oop_example
```

Code 3.6: SystemVerilog Class Examples

3.5.1 Encapsulation

SystemVerilog implements encapsulation and information hiding analogously to most of the other programming languages that support this feature. The main difference is that SV does not have the public keyword found in many of the other languages. SystemVerilog `class` data and methods are public by default. If we want to make data or methods private (private methods and data in other languages such as C++ or Java) we use the **`local`** keyword. Protected data and methods use the **`protected`** keyword.[41] See Code 3.7 for a brief example of the encapsulation syntax.

```
module encapsulation_example
   ...
```

[41]As a quick reminder, public data and methods (SV default, no keyword required) are accessible/visible from anywhere. Protected data and methods (`protected` keyword) are accessible/visible only in sub-`class`es. Private data and methods (`local` keyword) are not accessible/visible anywhere but in the class they are defined in.

```
  class my_class;
    //public variable
    int a;

    //protected variable
    protected int b;

    //private variable
    local int c;
     ...
    //public subroutine
    task t1( ...);
       ...
    endtask

    //protected subroutine
    protected task t2( ...);
       ...
    endtask

    //private subroutine
    local task t3( ...);
       ...
    endtask
     ...
  endclass
   ...
endmodule : encapsulation_example
```

Code 3.7: SystemVerilog Encapsulation Examples

3.5.2 Inheritance

SystemVerilog inheritance is pretty straight forward. It is basically identical to Java's implementation, so only *singe inheritance* is supported.[42] Even the syntax is identical to Java's, with the keyword **`extends`** to indicate a sub-`class` and **`super`** to call the parent `class`'s data and methods.

There is one thing we need to pay attention to however, the sub-`class`'s constructor. There are two important things to know about this topic: *(i)* the constructor is not inherited from the parent `class`, unlike the other methods. *(ii)*

[42]Single inheritance means that a `class` can extend only another `class`. It is not possible to have one `class` extending multiple parent `class`es like in C++.

the sub-`class`'s constructor will always invoke `super.new();` as the first statement. Even if you have not specified it. And even if the `new()` function is not compatible with the parent `class`. For example, if the parent defines the constructor to be `new(int a)`, without a default value, the sub-`class`'s constructor will still invokes `new()`. This will lead to an error since an `int` argument is expected. It is then a good practice, that is highly recommended, to always define constructors, even if their behaviour is the default one. Furthermore, begin the sub-`class`'s constructor with an explicit `super.new(...)` call that is compatible with the parent constructor. This call will overwrite the default parent constructor call. See Code 3.8 for an example.

```
module inheritance_example
   ...
  //parent class
  class my_class_a;
     ...
    int a;
     ...
    function new(int a);
       ...
      this.a = a;
       ...
    endfunction
     ...
  endclass
   ...
  //sub-class that extends the parent class
  class my_class_b extends my_class_a;
     ...
    int b;
     ...
    function new(int a, int b);
      //note that the first statement is a call to the parent constructor
      //if we had not included this call,
      //SV would have automatically called super.new();
      //since the parent constructor expects an int argument,
      //an error would have occurred
      super.new(a);
       ...
      this.b = b;
       ...
    endfunction
     ...
  endclass
   ...
endmodule : inheritance_example
```

Code 3.8: SystemVerilog Inheritance Examples

3.5.3 Polymorphism

While SystemVerilog inheritance is virtually identical to Java's implementation, polymorphism is very much C++ like. By default method virtuality is not enabled, just like in C++. Given an object handle that contains an instance of a sub-`class`, the methods that are going to be called are determined based on the handle type and not the actual object type. In other words, suppose we have a `class my_class_a`, a sub-`class my_class_b` that `extends my_class_a`, and suppose that both of these `class`es implement the `my_method()` method. If we now declare an object of child type (`my_class_b object = new(...);`) and a handle of the parent type that is assigned that object (`my_class_a my_handle = object;`), what happens when we call `my_method()` on that handle? The answer is that `my_method()` defined in `my_class_a` is called because the handle is of that type, even though the the object is actually of type `my_class_b`. To call the method based on the object type and not the handle type, we need to declare that method **`virtual`**. See an example below (Code 3.9).

Even though once we have declared a method to be `virtual`, all of the methods that overwrite it in sub-`class`es are `virtual` as well without needing to explicitly specify it with the `virtual` keyword, it is good practice to always include the `virtual` keyword for readability and documentation.

Another important tool for polymorphism is the **`$case(dst, src)`** subroutine.[43] This subroutine allows us to assign the contents of a parent `class` handle to a sub-`class` handle. The operation is successful only if the contents are of the sub-`class` handle's type.

[43]There are two versions of this subroutine. A `function` and a `task`. Both of them will check if the content of the `src` handle is of the same type as the `dst` handle. If that is the case the content of `src` will be copied into `dst`. The `task` throws a run-time error if the content of `src` does not match the `dst` handle and does not complete the cast. The `function`, on the other hand, return a 0 in this case and 1 if the cast was successful. Since the `function` allows us to recover from failed casts, it is preferred.

```
module polymorphism_example
   ...
  //parent class
  class my_class_a;
     ...
    function int f1( ...);
       ...
    endfunction

    virtual function int f2( ...);
       ...
    endfunction
     ...
  endclass
   ...
  //sub-class that extends the parent class
  class my_class_b extends my_class_a;
     ...
    function int f1( ...);
       ...
    endfunction

    //the virtual keyword is optional since the parent method is virtual
    //it is highly recommended to include it nonetheless
    virtual function int f2( ...);
       ...
    endfunction
     ...
  endclass
   ...
  my_class_b b = new( ...);
  my_class_a a = b;

  //the method in my_class_a will be called
  a.f1( ...);

  //the method in my_class_b will be called
  a.f2( ...);

  //try assigning to b the contents of a
  //since the contents of the a handle are actually of type my_class_b
  //the cast will succeed
  if($cast(b, a))
```

```
    b.f1( ...); //the method in my_class_b will be called
   ...
endmodule : polymorphism_example
```

Code 3.9: SystemVerilog Polymorphism Examples

Finally, I will mention that SV has a mechanism for defining **`virtual`** (or abstract) **`class`**es. These cannot be instantiated. They are instead used to define abstract entities from which we can extend `class`es that represent objects that can be instantiated. `virtual class`es can declare **`pure virtual`** methods. These are not implemented directly in the `virtual class` (only declared) but must be defined in any sub-`class` that `extends` the `virtual class`. See Code 3.10.

```
module virtual_classes_example
   ...
  //virtual class
  virtual class my_class_a;
     ...
    //pure virtual subroutine, note that it lacks the task body
    pure virtual task t1( ...);

    virtual task t2( ...);
       ...
    endtask

    task f3( ...);
       ...
    endtask
     ...
  endclass
   ...
  //extends the virtual class
  class my_class_b extends my_class_a;
     ...
    //we can omit defining t2 and/or t3
    //but we cannot omit t1, it must be defined since it is pure virtual
    virtual task t1( ...);
       ...
    endtask
     ...
  endclass
   ...
endmodule : virtual_classes_example
```

Code 3.10: SystemVerilog Virtual Class Examples

3.6 Interfaces

SystemVerilog introduces a mechanism that allows users to work at a higher level of abstraction when dealing with module interconnections. This mechanism is called an `interface`. At its core an `interface` is just a set of signals. These can have associated constructs, e.g. subroutines, as we will see later. What `interface`s allow you to do is working with a single entity that represents the entire module interconnection, instead of working with the single `nets` or `variables`.

An `interface` is similar to a `module` in that it needs to be declared as a separate entity (e.g. in a separate file) and instantiated when we want to use it. An `interface` block is delimited by the `interface...endinterface` keywords and contains, at the very least, a set of `variable`s and/or `net`s. See Code 3.11 for a more concrete example of how to declare and instantiate an `interface`. When you declare a `module`, you can declare an `interface` for its the `port` list. Now, instead of declaring the `port` list of that `module` with all the necessary signals, you can simply pass the newly declared `interface` as you would any other `port` argument, without specifying the direction (see Code 3.11).[44] Inside the `module` you will have access to all the signals defined in the `interface` using the dot notation (`interface_name.interface_signal`).

3.6.1 Ports and Parameters

Besides containing a set of signals, `interface`s support other functionality. One of these is the ability to interact with signals not directly defined inside the `interface`. This is done with **`interface port`**s. An `interface port` is simply a `port` list for the `interface`, and is defined in the same way as for `module`s. When instantiating the `interface`, we connect the external signal to it the same way we do for `module`s. See Code 3.11. Another way in which `module`s and `interface`s are similar is their support for parameterization. In particular, they

[44]You can also pass a **`generic interface`** that does not have any restriction on signals. Then, when you instantiate the `module` and pass an `interface` instance, you must make sure that the latter defines the signals you have used in the `module`. See details in the literature after you have finished this section.

are parameterized using the same syntax and rules (e.g. using `parameter`).

```
module interfaces_example
  ...
  logic res, clk;
  ...

  //declaration of an interface named my_if
  //interface port for the clk signal
  interface my_if(input logic clk);
    parameter SIZE = 32;
    logic en;
    logic [7:0] in;
    logic [7:0] out;
    ...
  endinterface : my_if

  //declaring my_if in the port list will make all the signals defined in the
    interface visible
  //this way there is no need to specify every single signal of the port (en,
    in, out, ...)
  module a (my_if bus, input logic res);
    ...
    //accessing signals defined in the interface via dot notation
    val = bus.in;
    ...
  endmodule : a

  module b (my_if bus, input logic res);
    ...
    //accessing signals defined in the interface via dot notation
    bus.in = val;
    ...
  endmodule : b
  ...

  //instantiation of an interface of type my_if named if_inst
  //connect the external signal clk to the interface port
  my_if #(.SIZE(64)) if_inst(.clk(clk));

  //we have connected module instances a_inst and b_inst with the same
    interface instance
  //they are now connected via the nets and variables defined in the interface
  a a_inst(.bus(if_inst), .res(res));
  b b_inst(.bus(if_inst), .res(res));
   ...
endmodule : interfaces_example
```

Code 3.11: SystemVerilog Interface Examples

3.6.2 Subroutines and `modport`s

The `interface` construct we have discussed up to now allows all of the `modules` connected with it to access its signals. This makes the `interface` connection symmetric. In practice, however, we often want to model asymmetric connections between `modules`. For example when modeling a master-slave relation. SV has an `interface` construct that allows us to model asymmetry, the **`modport`**. The latter allows us to specify which of the `interface` signals are visible to the `module` and what is their direction. The syntax for specifying a `modport` is:

`modport MODPORT_NAME(<direction> NAME {, <direction> NAME})`

where *`direction`* is the signal direction and *`NAME`* is the signal name.

When you want to specify that a `module` has a specific view of the `interface` defined in a `modport`, you can follow two approaches. The first one consists in specifying the `modport` directly in the declaration of the `module` and then, when instantiating the `module`, passing the `interface`. In the second one, during `module` declaration you only specify the `interface` and then, when you instantiate the `module`, pass the `modport` via the dot notation (`if_inst_name.modport_name`). See Code 3.12 for a clarifying example.

Another capability of `interfaces` is containing subroutines, dubbed **`interface methods`**. This is very useful for defining `functions` and `tasks` that are specific to the `interface`, as there would be a single place where these need to be written and maintained. Without this capability, each `module` would have to have its own copy of these subroutines. Writing subroutine for `interfaces` is identical to writing subroutines for `modules`. `interface method` are accessed via the dot notation (`if_name.method_name`). By default, no subroutine defined in the `interface` is visible via `modports`. To make a method visible, you must **`import`** it as shown in the code below (Code 3.12).

```
module modport_example
   ...
```

```
interface my_if;
  logic en;
  logic [7:0] in;
  logic [7:0] out;
  logic [15:0] comm;
   ...

  function read( ...);
     ...
  endfunction : read

  task write( ...);
     ...
  endtask : write
   ...

  //declaration of two modports to model the different views of the connection a master and slave have
  //note that slave does not see the comm signal and write task
  modport master(input in, output out, en, inout comm, import read, import write);
  modport slave(input out, en, output in, import read);
   ...
endinterface : my_if

//specifying the modport in the declaration
module a (my_if.master bus);
   ...
  //accessing intarface method
  bus.write( ...);
   ...
endmodule : a

//not specifying the modport in the declaration
module b (my_if bus);
   ...
  //accessing intarface method
  but.read( ...);
   ...
endmodule : b
 ...

my_if if_inst();

```

```
  //passing the interface
  a a_inst(.bus(if_inst));

  //passing the modport
  b b_inst(.bus(if_inst.slave));
   ...
endmodule : modport_example
```

Code 3.12: SystemVerilog Interface Modport and Subroutine Examples

3.6.3 Virtual Interfaces

So far the `interfaces` we have seen cannot be directly assigned to or referenced. It is however very useful in practice to be able to pass `interface` references around, especially in testbenches that follow UVM, as we will later see. We need some sort of `interface` handle, just like there are object handles. This `interface` handle (or, more appropriately, reference) is called in SV a **`virtual interface`**. When you declare the latter, you specify that the variable of that type will hold a reference to the `interface` and not the `interface` itself. This way, we can assign to a `virtual interface` an `interface` instance, passed as an argument, in a subroutine. This is not possible without `virtual interfaces`. For an example see Code 3.13.

```
module virtual_interfaces_example
   ...
  interface my_if;
     ...
  endinterface
   ...
  class my_class;
     ...
    //declaring a virtual interface (works like an interface reference)
    //initialized to null
    virtual interface my_if bus;
     ...
    function new(virtual interface my_if if_inst);
       ...
      //this assignment would have not been possible
      //cannot assign to an interface variable another interface variable
      bus = if_inst;
       ...
    endfunction
  endclass
   ...
  //interface instance
  my_if bus();

  //the interface instance is passed to the class instance
  //now the object has access to the interface instance
  my_class my_object = new(.if_inst(bus));
   ...
endmodule : virtual_interfaces_example
```

Code 3.13: SystemVerilog Virtual Interface Examples

3.7 Clocking Blocks

Unlike VHDL, Verilog can have race conditions between its clock and data signals. SystemVerilog introduces a mechanism to prevent this from happening and to allow the user to model signal skews during simulation/verification. This mechanism is the **`clocking` block** which, always from the perspective of a testbench that drives a DUT, can define a delay on the outgoing signals with respect to a reference signal (usually the clock). Furthermore, `clocking` blocks allow the user to define a set delay for testbench response sampling. That is, given a testbench that samples the DUT output at specific events (e.g. positive edges of the clock) we can specify that the signal should be sampled a fixed amount of time before that event.[45] `clocking` blocks can be defined in `modules` or `interfaces`. They defined the *clocking event* to which all the signals in the `clocking` block will be synchronized, the signals that must be synchronized, with their directions and the input and output delays.[46] Delays can be specified for individual signals or for all the outgoing and incoming signals. These two types of delay specifications can be mixed with individual delays (delay on single signals) having precedence over group delays (delays on all the input signals and output signals). For an

[45]This means that the sampled signal can change between sampling and the triggering event. For example, suppose we sample the DUT on each positive edge of the clock and that this sampling is done 10% of the clock period before the positive edge. We sample the signal after 90% of the clock period has elapsed but in the remaining 10%, before the next positive edge of the clock, the sampled signal could have changed.

[46]Note that a `clocking` block does not define new `nets` or `variables` but only declares the ones that need to be synchronized to the clocking event. On top of that, it specifies whether the signal should be considered as going from the testbench to the DUT (`output` signals) or coming from the DUT to the testbench (`input` signals)

example see Code 3.14. When driving or sampling signals defined in a `clocking` block, you can synchronize to events. For example, you could wait for the clocking event before interacting with the signals defined in the `clocking` block as in Code 3.14. Driving `clocking` block signals should always be done with non-blocking assignments. Furthermore, try to avoid mixing driving signals both with `clocking` blocks and without them.

```
module clocking_blocks_example
   ...
  logic clk;
  logic a, b, c, d, e, f, g, h, i;
   ...

  //note that the clocking block does not reference signals h and i
  //these signals are not part of the clocking block
  clocking my_cb @(posedge clk);
    //specifies default delays for inputs and outputs
    //inputs will be sampled 1ns before the rising edge of the clock
    //outputs will be driven to the DUT 2ns after the rising edge of the clock
    default input #1ns output #2ns;

    //the default 1ns delay will be applied to a and b
    input a, b;

    //a delay of 3ns will be applied to c
    input #3ns c;

    //the default 2ns delay will be applied to d and e
    output d, e;

    //a delay of 1ns will be applied to f
    output #1ns f;

    //same as:
    //input g;
    //output g;
    //the default 1ns delay will be applied when sampling g
    //the default 2ns delay will be applied when driving g
    inout g;
  endclocking : my_cb
   ...

  //synchronize the following statements to the clocking event
  @(my_cb);

  //drive the signal through the clocking block
  my_cb.a <= 1'b1;

  //sample the signal through the clocking block
  h <= my_cb.d;
   ...
endmodule : clocking_blocks_example
```

Code 3.14: SystemVerilog Clocking Block Examples

You can define multiple `clocking` blocks in the same `module` or `interface` to model different delays, group different signal or, for example, synchronize to different signals (e.g. `clocking` blocks that have different clock signals as their clocking event). One, and only one, of these `clocking` blocks can be defined as the default (see how in Code 3.15). Default `clocking` blocks allow you to work cycle delays, see how in the literature.

```
module default_clocking_blocks_example
   ...
  logic clk1, clk2;
   ...
  clocking my_cb1 @(posedge clk1);
     ...
  endclocking

  clocking my_cb2 @(posedge clk2);
     ...
  endclocking

  //specifying that my_cb2 is the default clocking
  default clocking my_cb2;
   ...
endmodule : default_clocking_blocks_example
```

Code 3.15: SystemVerilog Default Clocking Block Examples

3.8 Randomized Stimulus

Verifying a design often involves sending stimulus to the DUT and checking the response to make sure that we obtained the desired behaviour. There are different strategies for generating stimulus. One of the simplest involves looping through the possible inputs in a predefined way (e.g. writing loops that iterate over all the inputs). This strategy might cover all possible inputs, however, it has a significant drawback. Since the stimulus was generated by iterating over the possible inputs an a deterministic way, with each input combination being generated only once, we have covered very few state transitions. For example, consider a module that takes in input two 8-bit numbers and returns their sum. We might stimulate this module by writing an external loop that iterates over all the possible 8-bit numbers and an internal loop that does the same. The generated stimulus will then be the sequence: $(00, 00), (00, 01), \ldots, (00, FF), (01, 00), \ldots, (01, FF), \ldots$. Even though with this sequence we might cover all the possible pairs of two 8-bit numbers, we will never see, for example, the input sequence $(00, 00), (FF, FF)$. Writing directed tests that cover all the possible state transitions is much more involved. Furthermore, the number of transitions grows exponentially (or even worse) with the number of possible inputs.

Constrained random verification (CRV) keeps in mind these limitations of writing directed tests and tries to improve the verification process. The aim is that of allowing verification of as many state transitions as possible, with the minimal amount of code needed. The idea is generating stimulus in a pseudo-random way that conform to some user specified constrains. The latter are used to make sure that the stimulus is legal and respects the verification needs. Since the state transitions are random with this approach, we are likely to see many state transitions that were not included in the directed tests.

3.8.1 Randomizing Variables

The first tool used in CRV is randomization of variables. This is done with the **`randomize()`** system function. This function takes in input a list of `variables` and randomizes them according to the constraints. If the constraints do not contradict each other and randomization was successful, the function returns 1, 0 otherwise. See an example below (Code 3.16). You can specify the random number generator's seed using the **`process::self.srandom()`** method. Additionally, the random seed can be set from the command line argument with the

+svseed=SEED, where SEED is a positive integer number.

We can add constraints to randomization in various ways. The simplest one is by specifying the constraints directly after the **randomize()** call. This can be done using the **with** clause followed by brackets ({}) that contain the constraints. Constraints are semicolon (;) separated. They can be relational expressions (e.g. `a >= b;`), contain value ranges using the `inside` keyword (e.g. `a inside {[0:100]};`) or define distributions over the possible values. The latter is achieved by using the **dist** keyword. This keyword can be thought of as an extension of `inside` that also allows the user to specify the weight of each element that can be selected. The probability of an element to be selected is then equal to its weight divided by the sum of the weights of all selectable elements (e.g. `a dist {[0:50]:=2, [51:100]:=1};` has probability of generating 0 equal to 2/(2*51 + 1*50), while the probability of generating 100 is 1/(2*51 + 1*50)).[47] Last, I will mention that it is also possible to use `if-else` constructs when defining constraints. For a clarifying example of the above discussion see Code 3.16. You can find out more about writing constraints in the literature.

```
module randomizing_variables_example
   ...
  typedef enum bit[1:0] {IDLE, START, S1, S2} fsm_state_e;
  logic [15:0] data;
  fsm_state_e state;
  int ok;
     ...
    //set random seed to 1
    process::self.srandom(1);

    //if randomization fails report it
    if (!randomize(data, state))
      $display("randomization failed");
     ...
    //constraints that use relational expressions
    ok = randomize(data) with {data >= 100;};
     ...
    //constraints that use ranges
    ok = randomize(data) with {data inside {[100:130], 135, [140:170]};};
     ...
    //constraints that use dist
```

[47]Note that if you specify all the weights to be 1 you will obtain the same result you would with `inside`.

```
    ok = randomize(data) with {data dist {[100:130]:=5, 135:= 15,
    [145:170]:=3};};
     ...
    //constraints that use if-else
    ok = randomize(data) with {if (state == IDLE) data <=200;
                               else if (state == START) data >= 200;};
     ...
    //combination of different constraints
    ok = randomize(data) with {if (state == S1) {data >= 64; data <= 128;}
                               else if (state == S2) data dist {[0:10]:=1,
    [20:30]:=2};};
     ...
endmodule : randomizing_variables_example
```

Code 3.16: SystemVerilog Variable Randomization Examples

3.8.2 Randomizing Properties

For `class`-based verification, that is verification where both the testbench as well as the stimulus is represented through `class`es, it is important to be able to randomize `class` properties, in particular data. SystemVerilog has mechanisms specifically dedicated to that. Integral data items in `class`es can be declared **`rand`** or **`randc`**. Then, using the familiar `randomize() function` we can randomize all the properties that include these keywords. `rand` variables can be randomized to any legal value (picked uniformly at random). `randc` variables are obtained by cycling through the possible values in a random order, a specific value can appear only once per cycle. In other words, if a `randc` variable has been randomized to a specific value, that value can only be seen again once all the other legal values have been seen.

When you call the `randomize() function` two additional `void functions` are implicitly called. **`pre_randomize()`** just before the `randomize()` call and **`post_randomize()`** just after. By default these `function`s do not perform any action, but you can overwrite them (an only them, you cannot overwrite `randomize()`) inside the `class` to obtain some desired behaviour.

If you have an aggregate `class`, randomization will not propagate to the contained `class` instances.[48] To make the randomizable data items of the contained `class` instances react to `randomize()` you must declare the contained `class`

[48]As a reminder, an aggregate `class` is a `class` that contains other `class` objects as data items.

instance `rand`. See an example below (Code 3.17).

If you want to only randomize a subset of the randomizable data items of an object, pass those data items as arguments to the `randomize() function` call.

Last, it useful to know that it is possible to enable/disable randomization for individual `class` data items. This is done with the **`rand_mode()`** subroutine. The `task` version of this subroutine expects as argument either a 1 (indicating that randomization should be enable) or a 0 (indicating that randomization should be disabled). The `function` version does not expect any argument and instead returns an `int` indicating the randomization status. This subroutine can be called on the entire object or on single data items. For a clarifying example see Code 3.17.

```
module randomizing_properties_example
   ...
  class my_class_a;
    rand bit [7:0] var1;
    bit [7:0] var2;
  endclass

  class my_class_b;
    //will not be randomized
    int a;

    //will be randomized
    rand int b;

    //will be randomized
    //with each value appearing exactly once per randomization cycle
    randc int c;

    //without including the rand keyword the var1 variable of obj
    //would not be randomized by the randomize() call
    rand my_class_a obj;
     ...
    //after randomization, make sure that a is always equal to b+c
    function void post_randomize()
      a = b+c;
    endfunction
     ...
  endclass
   ...
  my_class_b my_object = new();
   ...
  //disables randomization for the variable c
  my_object.c.rand_mode(0);

  //enables randomization for all the variables (will affect only c)
  my_object.rand_mode(1);

  //check if c can be randomized
  int status = my_object.c.rand_mode();
  if (status)
    $display("c can be randomized");
   ...
  //will randomize var1, b and c
  if (!my_object.randomize())
    $display("randomization failed");

  //will only randomize b
  if (my_object.randomize(b))
    $display("randomization failed");
   ...
endmodule : randomizing_properties_example
```

Code 3.17: SystemVerilog Class Randomization Examples

3.8.3 Constraint Blocks

Until now we only have see how to apply constraints to variable randomization. We still need to discuss how constraints are declared when randomizing `class` instances. This is done by declaring **`constraint`** blocks inside the `class`. These contain semicolon separated constraint defined as described previously. For an example of the syntax see Code 3.18. `constraint` blocks are inherited and can be overwritten, just like methods. You can also enable/disable constraints using the **`constraint_mode()`** subroutine the same way you use `rand_mode()` for randomizable data items.

It is important to understand how randomization and constraints work. When you call `randomize()`, all the `randc` data items are randomized simultaneously. Then, all the `rand` data items are also randomized simultaneously. At this point the randomized data items are checked to see if all constraints are respected. If that is the case, randomization succeeds and return. Otherwise, a new set of value is randomly generated in the same way and checked against the constraints.

This continues until either a valid random combination is found (success) or the randomization space has been exhausted (randomization failure). Upon randomization failure a warning message will be generated. If randomization fails, even for one variable, all variables are not changed. What this randomization process implies is that there is no order in which random variables of the same type (i.e. `randc` or `rand`) are generated. All of them are generated simultaneously. So, if we have a desired order in which data items should be randomized, as in the case where one variable depends on another, we need to explicitly indicate that. This is done by including in the `constraint` block a statement that indicates a precedence. This statement is **`solve VAR1 before VAR2;`** and indicates that `VAR1` should be randomized before `VAR2`. See Code 3.18 for an example. More details can be found in the literature.

```
module constraint_blocks_example
  ...
  class my_class;
    rand bit [1:0] a;
    rand bit [1:0] b;

    //constraint block
    constraint ab_diff {a != b;}

    //constraint block
    constraint const_1 {a > 2'b10; b > 2'b01;}

    //a will be >= 2'b10 1/8 of the time
    //since only 2/16 possible combinations of a and b satisfy the constraint
    //a=00,b=00 | a=00,b=01 | a=00,b=10 | a=00,b=11 | -> invalid
    //a=01,b=00 | a=01,b=01 | a=01,b=10 | a=01,b=11 | -> invalid
    //a=10,b=00 -> valid
    //a=10,b=01 | a=10,b=10 | a=10,b=11 | -> invalid
    //a=11,b=00 -> valid
    //a=11,b=01 | a=11,b=10 | a=11,b=11 | -> invalid
    constraint const_bug {if (a >= 2'b10) b = 2'b00;}

    //a will be >= 2'b10 1/2 of the time
    //since a is randomized before b
    constraint const_fix {if (a >= 2'b10) b = 2'b00; solve a before b;}
     ...
  endclass
  ...
  my_class my_object = new();

  //disable the const_1 and const_bug constraints
  my_object.const_1.constraint_mode(0);
  my_object.const_bug.constraint_mode(0);

  //the randomized values will respect the ab_diff and const_fix constraints
  void'(my_object.randomize());
   ...
endmodule : constraint_blocks_example
```

Code 3.18: SystemVerilog Constraint Block Examples

3.9 Coverage

In CRV stimulus is not directly specified by the verification engineer. Instead, it is randomly generated according to some constraints. Since test are randomized, we need a way of checking whether the generated stimulus verifies some functionality of the circuit. The number of features verified by the stimulus is called functional coverage. In directed test there was no need for features that support it since the checked functionality was implicit in the test. In randomized tests we might have certain runs that verify some specific functionality and others that do not. Coverage is an extensive topic, here I present some basics of *data-oriented functional coverage*. For a more exhaustive dive into SystemVerilog coverage consult, for example, [14].

3.9.1 The Basic Constructs

The basic building block of functional coverage is the **`covergroup`** block. The latter can identify a *sampling event*, which specifies the instance when coverage should be updated, and a list of **`coverpoint`**s, which are signals to be tracked. A `covergroup` is enclosed in the `covergroup`-**`endgroup`** delimiters and is an object-like construct. You can define it in `module`s, `class`es, `interface`s, etc. Once you declared a `covergroup`, you need to instantiate it. This is done similarly to objects by calling the `new()` function. As stated before, `covergroups` contain `coverpoints`. These are expressions that can optionally contain a condition that guards their sampling. Each `coverpoint` has a number of associated **`bins`**, which are counters for subsets of the values the expression can take. By default, if you specify a `coverpoint` without specifying the associated `bins`, one bin will be generated for each possible value of the expression.[49] If the user manually defines the `bins`, the default ones will not be generated. See an example of these constructs below (Code 3.19). Each time the sampling event occurs, the values of the `coverpoints` are checked. If there is a bin that contains that value it is incremented. In the end, we can check the count for each bin.

It is also possible, as shown below (Code 3.19), to define more complex `bins`. **`illegal_bins`** defines the values that are illegal for the expression. **`ignore_bins`** defines the values that should be ignored. Illegal `bins` have precedence over ignored `bins`. This means that, if a value is part of `illegal_bins`, it will not be ignored even if it is also part of `ignore_bins`. You can also define vector `bins`. These can be of constrained or unconstrained sizes. Vector `bins` of unconstrained size will generate a separate bin for each unique value. Those of constrained size will generated a number of `bins` equal to the specified size and populated them with the values that you indicate (duplicates will be retained).[50] Last, if you define a bin as **`default`**, all the values not part of other `bins` will be part of that bin.

```
module coverage_constructs_example
   ...
  logic clk,
  logic [8:0] var1;
  logic [15:0] var2;

  //definition of a covergroup named my_cove_g
  //the coverpoints will be sampled at each rising edge of the clk,
  //which is the sampling event
  covergroup my_cov_g @(posedge clk);
    //coverpoint for var1,
    //indicates that the expression (signal) should be tracked by coverage
    //a default bin generated for each of the possible 256 values of var1
    va1_cp: coverpoint var1;

    //a more complex coverpoint
    //the expression to be tracked is var1+var2
    //this expression should be sampled only if the guarding condition (var1 >
     0 && var2 > 0) is met
    sum_cp: coverpoint (var1+var2) iff (var1 > 0 && var2 > 0){
      //this bin will be hit if var1+var2 is equal to 0 or 1 or ... or 64 or
```

[49]The number of individual `bins` for each value has a limit. It can be changed by the user.
[50]To see how duplicates, illegal and ignored values are removed from `bins` see the literature.

```
  128
   bins low = {[0:64], 128};

   //this bin will be hit if var1+var2 is equal to 129 or 130 or ... up to the maximum possible value
   bins high = {[129:$]};

   //note that since no bin contains the values 65, 66, ..., 127
   //these values will be ignored
  }

  //a coverpoint that illustrates some of the possible bins
  var2_cp: coverpoint var2{
    //a single bin that contains values 0, 1, ..., 10
    bins b1 = {[0:10]};

    //vector bins of unconstrained size
    //a different bin for each value 20, 21, ..., 30
    bins b2[] = {[20:30]};

    //vector bins of constrained size
    //6 different bins
    //first contains 30 and 31, second contains 32 and 33, ..., sixth contains 39 and 40
    bins b3[6] = {[40:50]};

    //a single bin that specifies the illegal values
    illegal_bins b4 = {0, 20, 40};

    //a single bin that specifies the values to be ignored
    ignore_bins b5 = {10, 30, 50};

    //a single bin containing all the other values
    bins b6 = default;
  }
 endgroup : my_cov_g

 //instantiating a covergroup
 //each time the sampling event occurs for this covergroup
 //the values of the tracked expressions will be checked
 //if they match any bins, those will be incremented
 my_cov_g cov_inst = new();
  ...
endmodule : coverage_constructs_example
```

Code 3.19: SystemVerilog Coverage Constructs Examples

You can also have even more complex `bins`. Ones that keep track of the intersection of other `bins`. These are called cover `cross`es. You can also have `bins` that do not keep track of the specific values an expression has taken, rather, of the transition between values that expression takes. Once the reader has familiarized with the simple coverage `bins` presented here, it is suggested to consult the literature to find out more about these more advanced constructs.

3.9.2 `covergroup`s in Classes and `covergroup` Methods and Options

As stated previously, it is also possible to define `covergroup`s in `class`es. This way we can define the coverage model for that specific `class`. The syntax for this type of `covergroup` is slightly different from what we have seen the previous example (Code 3.19). In particular, each declaration creates an anonymous `covergroup` instance. This instance has access to all the variables of that `class`, even the `local` and `protected` ones.

You can also call methods on `covergroup`s, `coverpoint`s or `bins` that perform some specific action. Those can, for example, specify that a `covergroup` should be sampled at this moment of time or that the count of certain `bins` should be `return`ed. `covergroup`s, `coverpoint`s and `bins` are also parameterized. These parameters are called *options* and can specify, for example, the name or relative weight of `bins`. The user can modify the options to customize the coverage constructs. For an exhaustive list of all the options and methods consult the literature. For an example of `covergroup`s in `class`es and the use of options and methods see Code 3.20.

```
module coverage_in_classes_example
   ...
  class my_class;
    logic [7:0] var1;
    logic [15:0] var2;
     ...
    //anonymous covergroup instance
    covergroup my_cg;
      var1_cp: coverpoint var1;
      var2_cp: coverpoint var2{
        bins b1 = {[0:10], 12};
        bins b2[] = {[20:30], 32};
      }
    endgroup
     ...
    function new()
      //note that when we instantiate the covergroup we do not specify the
    name since it is anonymous
      //as a consequence only one instance can exist per covergroup type
      my_cg = new();
    endfunction
  endclass
   ...
  my_class my_object = new();
   ...
  //option that specifies that the weight of the b1 bin should be 12 (the
    other bins have weight 1, which is the default)
  my_object.my_cg.b1.option.weight = 12;

  //method that indicates that the covergroup should be sampled at this
    instance
  my_object.my_cg.sample();

  //display the coverage that is obtained by calling the get_inst_coverage()
    method
  $display("my_cg coverage: %0.2f %%", my_object.my_cg.get_inst_coverage());
endmodule : coverage_in_classes_example
```

Code 3.20: SystemVerilog Coverage Constructs with Classes Examples

3.10 Assertions

An important language tool are ***assertions***, especially for verification engineers. These constructs are usually ignored by the synthesis tool and, if used systematically and with care, can be the basis for a verification strategy. This approach to verification is called Assertion-Based Verification (ABV). There are different types of assertions that fulfill some specific purposes. The basic idea of assertions is simple: evaluate a condition, if it is true pass, otherwise fail. Usually we are interested in fails as they might indicate an error in our design. In its most basic form an assertion is similar to an `if-else` statement. Assertions are another extensive topic that requires some time and effort. Here I will be presenting the basics. For a more in depth coverage consult [15] [14].

3.10.1 Immediate Assertions

The simplest kind of `assertion` is the *immediate assertion*. The latter evaluates a boolean expression. The assertion passes if the expression evaluates to `1` and fails for any other value (`0`, `X`, `Z`). Each time an assertion fails and error message is generated. Labeling your assertions with meaningful names is highly recommended as upon failure the error message will contain the assertion label. With carefully chosen names debugging and error comprehension can be much easier. If you do not specify assertion labels default one will be generated and are usually harder to understand.

You can specify actions to be executed on assertion pass and/or fail (usually only fails are of interest). The syntax for doing so is very similar to that of an **`if-else`** statement but uses the **`assert`** keyword. See Code 3.21 for an example. The set of statements that is executed upon pass or fail is called an *action block*. The latter has access to the assertion label via the `%m` format specifier.[51] If you need to print something in the action block of an assertion fail do not use the `$display()` subroutine. Instead, use **`$info()`**, **`$warning()`**, **`$error()`** and **`$fatal()`**. These follow the same syntax as `$display()` but also allow you to specify the *severity level* of the fail. If severity level is `fatal` the simulation will be terminated. Note that the printed message will follow the error message of the assertion fail and not overwrite it.

```
module immediate_assertions_example
   ...
  logic en_low, en_high;
   ...

  //immediate assertion labeled BOTH_EN_NOT_ON
  BOTH_EN_NOT_ON: assert (!(en_low && en_high)) begin
    $display("%m passed");  //executed upon pass
  end else begin
    $error("assertion failed");   //executed upon fail
  end
   ...
```

[51]Note that upon failure the assertion label will be printed automatically so `%m` is primarily useful in the action block of the pass.

```
endmodule : immediate_assertions_example
```

Code 3.21: SystemVerilog Immediate Assertion Examples

3.10.2 Concurrent Assertions

Often we need to monitor behaviour that spans multiple clock cycles. With the immediate assertions we have just seen we cannot achieve this. Luckily, SV has another type of assertions that allows us to do just that. This type of assertions are called *concurrent assertions*. They verify a **`property`**, which is composed of a *temporal condition* and a boolean expression. The temporal condition specifies the event that triggers the evaluation of the boolean expression. This expression can span multiple clock cycles. It can also be any single cycle expression usable as the condition of an `if-else` statement. Later we will see how to specify expressions that span over time. The `property` is specified, usually in a `module` or `interface`, with the `property-endproperty` block and can be labeled. See an example below (Code 3.22). To then `assert` the `property` we must use the `assert` keyword in conjunction with the `property` keyword as in the example of Code 3.22. It is important to remember to label your assertions.

Let's now see some constructs for defining expressions that are evaluated over time. I will present them first and then we will go over their meaning. The constructs are:

- *`expr1 ##N expr2`*: cycle delay.
- *`expr[*N]`*: repeat.
- *`expr[*n:N]`*: range repeat.
- *`expr1 |-> expr2`*: same cycle implication.
- *`expr1 |=> expr2`*: next cycle implication.

The cycle delay `expr1 ##N expr2` specifies that `expr1` should be evaluated, then, after `N` (with $N \in \mathbb{N}$) clock cycles, `expr2` should also be evaluated. This sequence passes when both evaluations pass, and fails in any other case. `expr1 ##0 expr2` is also legal and indicates that the two conditions should be evaluated in the same clock cycle.

`expr[*N]` is equivalent to evaluating `N` (with $N \in \mathbb{N}$) times `expr`, with a cycle delay of one between each evaluation. For example, `expr[*3]` is equivalent to `expr ##1 expr ##1 expr`.

`expr[*n:N]` (with `n`, `N` $\in \mathbb{N}$) is equivalent to $\cup_{i=\texttt{n}}^{\texttt{N}}$`expr[*`$i$`]`. For example, `expr[*1:3]` will pass if either `expr`, `expr ##1 expr` or `expr ##1 expr ##1 expr` passes. In other words, this expression indicates that we expect a sequence composed of a minimum of 1 to a maximum of 3 `expr`s. It is also possible to specify `n` as 0, indicating that the sequence may never occur.

`expr1 |-> expr2` indicates that if `expr1` is true, then `expr2` must also be true in the same clock cycle. `expr1 |-> expr2` will pass if either `expr1` is false, or both `expr1` and `expr2` are true in the same clock cycle.

`expr1 |=> expr2` indicates that if `expr1` is true, then `expr2` must also be true in the next clock cycle.

Below (Code 3.22) you can see examples of properties that use the constructs presented above to define expressions that span multiple clock cycles.

```
module concurrent_assertions_example
   ...
  logic clk, en, rst;
  logic a, b, c;
   ...
  //definition of a simple property named MY_PROPERTY
  //the event that will trigger the evaluation of the assertion is the posedge
    of clk
  //the expression that is evaluated is that en and rst should not be both 1
  //note that this is not an expression that spans multiple clock cycles
  property MY_PROPERTY;
    @(posedge clk) !(en && rst);
  endproperty

  //asserting MY_PROPERTY
  MY_ASSERTION: assert property (MY_PROPERTY);
   ...
  //this is a more complex assertion that uses sequences, let's break it down
  //the assertion will be evaluated at the posedge of the clk, but only if en
    is 1
  //if during the assertion evaluation rst is 1, the assertion will not be
    evaluated and any sequence that was in the process of evaluation will be
    aborted
  //the sequence that needs to be asserted is:
  //if c is 1 than in the next clock cycle a should be 1 and after two
    additional clock cycle we expect b to be 1 for between 0 to 5 clock cycles
  //once the sequence of b=1's finishes, if the sequence up to this point is
    correct, after 5 additional clock cycles c should be 1 for 2 clock cycles
  property MY_PROPERTY_2;
    @(posedge clk iff (en)) disable iff (rst)
      c |=> a ##2 b[*0:5] |-> ##5 c[*2];
  endproperty

  MY_ASSERTION2: assert property (MY_PROPERTY_2)
   ...
endmodule : concurrent_assertions_example
```

Code 3.22: SystemVerilog Concurrent Assertion Examples

3.11 Threads and Interprocess Synchronization

SystemVerilog introduces several modifications to enhance Verilog's support and capabilities for multi-threaded programming. First, it introduces two new constructs that team up with `fork-join` in defining concurrent blocks. These are **`fork-join_any`** and **`fork-join_none`**. As a reminder, when a `fork-join` block is encountered, the main execution flow is split with each statement of the block being executed concurrently. When all the statements are executed, the main execution thread is resumed from the `join` statement onward. With `fork-join_none` the main execution thread does not wait for all the statements within the concurrent block to be executed. It continues its execution concurrently with all forked threads as if they had not been spun off. A `fork-join_any` block, on the other hand, does stop the execution flow of the main thread. However, the flow is stopped not until all the statement in the concurrent block are executed (as in the case of a `fork-join`), but until any of the concurrent statements has finished. Once this has happened, the main execution thread resumes running (concurrently with the remaining threads of the `fork-join_any`).

Since now the main thread that spun off the concurrent block's sub-threads can run concurrently with them, we need a way of synchronizing the main thread and its sub-threads. This is done in SV with two statements: **`wait fork`** and **`disable fork`**. Upon encountering the former, the main thread stops execution until all of its sub-threads have executed and then resumes. Upon encountering the latter, the main thread kills all of its sub-threads and resumes execution.[52] For an example see Code 3.23.

[52]You can also disable a single sub-thread and not all of them. To do so, give the statement you want to disable a label (say `LABEL`), then, use `disable LABEL;` instead of `disable fork;` to kill only the sub-thread associated with that label.

```
module concurrent_blocks_example
   ...
  $display("main thread 1: executing");
  fork
    $display("main thread 2 = sub-thread 1 of main thread 1: executing");
    $display("main thread 3 = sub-thread 2 of main thread 1: executing");
    begin : sub_3
      $display("main thread 4 = sub-thread 3 of main thread 1: executing");
    end
  join_any

  //once any of the 3 sub-threads finishes main thread 1 resumes from here
  //2 sub-threads still running concurrently with main thread 1
  $display("main thread 1: executing");

  //kill sub-thread 3 (if it was not the first sub-thread to finish executing)
  disable sub_3;

  //main thread 1 could be running concurrently with 0, 1 or 2 of its sub-
    threads
  $display("main thread 1: executing");

  //wait for the remaining sub-threads to finish
  wait fork;

  //main thread 1 is running alone
  $display("main thread 1: executing");
   ...
endmodule : concurrent_blocks_example
```

Code 3.23: SystemVerilog Concurrent Block Examples

3.11.1 `semaphores`

SystemVerilog provides a built-in construct for managing inter-processor synchronization, the **`semaphore`**. This synchronization mechanism works analogously to semaphores in other programming languages like Java. It is essentially a container of tokens, which can be added or removed atomically.[53] SV `semaphores` are an object-like mechanism that requires declaring a handle of type `semaphore` and

[53]An operations is said to be atomic if it cannot be interrupted by other processes, as if it were executed in the same fetch-decode-execute cycle.

initializing it with the `new(n)` constructor. This `function` expects an argument `n` that indicates the `semaphore` size, if not specified, a default value of 0 will be used. The other subroutines that are supported by this mechanism are:

- **`get(n)`**: `task` that extracts `n` semaphore tokens. If `n` is not specified, a default value of 1 will be used. This `task` blocks if the specified number of tokens is not available.

- **`try_get(n)`**: `function` that extracts `n` semaphore tokens. If `n` is not specified, a default value of 1 will be used. This `function` does not block if the specified number of tokens is not available. Instead, it simply returns 0 without modifying the number of tokens.

- **`put(n)`**: `function` that adds `n` semaphore tokens. If `n` is not specified a default value of 1 will be used.

It is the responsibility of the user to make sure that the number of tokens returned by a process is equal to the number of tokens taken. SV does not perform this check. For an example of `semaphore`s see Code 3.24.

```
module semaphores_example
   ...
  //declaring a semaphore named my_semaphore
  semaphore my_semaphore;
   ...
  //initializing the semaphore with 2 tokens
  my_semaphore = new(2);
   ...
  fork
    begin : process_1
      //asking for a token
      //since there are 2 tokens, only two processes can execute concurrently,
     the other will be blocked waiting for a token
      my_semaphore.get();
       ...
      //returning the token
      my_semaphore.put();
    end
    begin : process_2
      my_semaphore.get();
       ...
      my_semaphore.put();
    end
    begin : process_3
      my_semaphore.get();
       ...
      my_semaphore.put();
    end
  join
   ...
endmodule : semaphores_example
```

Code 3.24: SystemVerilog Semapthore Examples

3.11.2 mailboxes

Often times we might need a way of synchronously passing messages between processes in our testbenches. Luckily, we do not need to figure out ourselves how to best implement a mechanisms that enables us to do that. SV has a build-in inter-process communication construct, the **`mailbox`**. The latter can be seen as a FIFO that allows us to atomically add, inspect or remove elements from it. This FIFO can be of a fixed or unlimited size. It can also be specified to accept operations on a specific data type or on any data type. `mailbox`es are an object-like construct that requires declaring a handle of type `mailbox` and initializing it with `new(n)`. The constructor expects an argument `n` that specifies the `mailbox` size, if no value is provided, a default value of 0 will be used indicating a `mailbox` of unlimited size. The other subroutines supported by this construct are:

- **`num()`**: `function` returning an `int` indicating the number of messages currently in the `mailbox`.
- **`get(message)`**: `task` that retrieves the head of the FIFO in the `message` argument that is passed as a `ref`. If the head of the FIFO is not of a type compatible with `message`'s type, an error is generated and the `message` is not retrieved. Blocks if the `mailbox` is empty.
- **`try_get(message)`**: `function` that retrieves the head of the FIFO in the `message` argument that is passed as a `ref`. Returns an `int`. If retrieving was successful the return value is positive. If the head of the FIFO is not of a type compatible with `message`'s type, returns a negative value and the `message` is not retrieved. Does not block if the `mailbox` is empty. Instead, it simply returns 0.

- **`peek(message)`**: `task` that copies the head of the FIFO in the `message` argument that is passed as a `ref`. If the head of the FIFO is not of a type compatible with `message`'s type, an error is generated and the `message` is not copied. Blocks if the `mailbox` is empty.
- **`try_peek(message)`**: `function` that copies the head of the FIFO in the `message` argument that is passed as a `ref`. Returns an `int`. If retrieving was successful the return value is positive. If the head of the FIFO is not of a type compatible with `message`'s type, returns a negative value and the `message` is not copied. Does not block if the `mailbox` is empty. Instead, it simply returns 0.
- **`put(message)`**: `task` that adds `message` at the tail of the FIFO. Blocks if the `mailbox` is full.[54]
- **`try_put(message)`**: `function` that adds `message` at the tail of the FIFO. It does not blocks if the `mailbox` is full. Instead, it simply returns 0.

`mailbox`es can be parameterized indicating the type of data that it is allowed to operate on. The syntax is:

mailbox #(<data_type>) MAILBOX_NAME;

See Code 3.25 for an example of this construct.

```
module mailboxes_example
   ...
  int a, b;
  logic c, d;
   ...
  //declaring a mailbox named my_mailbox_1 that can operate on any data type
  mailbox my_mailbox_1;

  //declaring a mailbox named my_mailbox_2 that can operate only on int data
  mailbox #(int) my_mailbox_2;
   ...
  //initializing my_mailbox_1 to be of unlimited size
  my_mailbox_1 = new();

  //initializing my_mailbox_2 to be of size 5
  my_mailbox_2 = new(5);
```

[54]Note that for `mailbox`es of unlimited size this subroutine will never block.

```
  ...
  fork
    begin : process_1
      ...
      my_mailbox_1.put(c);
      void'(my_mailbox_2.try_put(a));
      ...
    end
    begin : process_2
      ...
      my_mailbox_1.get(d);
      void'(my_mailbox_2.try_get(b));
      ...
    end
  join
  ...
endmodule : mailboxes_example
```

Code 3.25: SystemVerilog Mailbox Examples

3.12 Further Topics

In this chapter we have see only a quick introduction to SystemVerilog. Many of the topics presented above can be covered in much more detail. Furthermore, there are a few major topics that were not covered. In this brief section I would like to mention some of these topics, inviting the reader to consul the reference material [13] [12] [14] [15] [4].

- **strings**.
- **events and event triggers**.
- **randcases**.
- **packages**.
- **program blocks**.
- **interface classes**.
- **Hierarchy** and **Connectivity**.
- **Direct Programming Interface (DPI)**.

- SystemVerilog **Simulation Cycle** and **Event Scheduler**.
- SystemVerilog **Scoping**.

Chapter 4

Universal Verification Methodology (UVM)

As time goes on, digital integrated circuits become more and more complex. This trend can be attributed to a variety of factors. For example: the innovations in the microelectronics field that enable the famous Moore's law, the addition of ever more features to already existing designs, the modern trend of SoCs, and so on.[55] Design complexity drives the verification effort. To keep up with this trend we need a standardized, flexible and reusable verification methodology. Standardization allows the almost plug-and-play use of verification IPs. Furthermore, it eases the concurrent development of verification components by more members of the verification team. Flexibility helps keep the improvisation in testbenches to a minimum, while reusability allows for more verification time.

In this chapter I will introduce one of the most recent and widely adopted verification methodologies, the **Universal Verification Methodology** (**UVM**). UVM supports the Metric-Driven Verification (MDV) paradigm.[56] As such, it provides the necessary code features to define the standard components of a typical testbench that implements this verification paradigm. In practical terms, UVM is a SystemVerilog library with a set of standard classes.[57] UVM, as the name suggests, is also a methodology for performing verification that defines stan-

[55]SoC stands for System-on-Chip (or System-on-a-Chip) and is a design principle consisting in integrating different components of a system (e.g. CPUs, GPUs, Interfaces and DSPs) on the same substrate. SoCs often include digital, analog, RF, and mixed-signal functionalities on the same chip.

[56]As a reminder, this paradigm involves generating stimulus in a constrained randomized way and check the DUT output to update the coverage and adapt stimulus generation.

[57]UVM is also compatible and available for other languages used in verification, such as *SystemC* and *e*. Furthermore, it is possible to develop UVM test environments that use multiple languages.

dard constructs and architectures for the different verification components of the testbench. It is an IEEE standard (IEEE 1800.2) supported by Accellera [16], which also provides the methodology documentation.

One of the key features of UVM (and not only) is the separation of the data layer and the testbench infrastructure. These two components of verification can be developed concurrently and separately. The data layer is represented as standard *data item* constructs (as we will see latter, the `uvm_sequence_item` class). The testbench infrastructure is typically built using UVM environments (as we will see latter, the `uvm_env` class) that contain other environments in a recursive manner. The leafs of this hierarchical testbench structure are the environments that implements *UVM Verification Component*s (*UVC*s). These represent standard verification IP that, once developed, can be used in a variety of verification tasks. There are two main types of UVCs: the *Interface UVC* and the *Module UVC*. We will explore these constructs, and their architecture, in the following sections.

UVM also provides a standard communication protocol, dubbed the *Transaction Level Modeling* (*TLM*) protocol, that we will not be presented in this work. Last, I will mention that it also supports constructs for modeling memory elements (*UVM Register Modeling*), but those will also not be covered here. More in general, following the pattern outlined up to now, what follows is not an exhaustive coverage of UVM. The scope is introducing the methodology and presenting some constructs used in, and justifying, the later chapters. For an in-depth guide to UVM consult, for example, [17]. You can also find plenty of material online. For example, Accellera's site [16], Cadence courses [4], or web pages that cover this topic [18].

4.1 Data Modeling

UVM models the basic blocks of the stimulus we send to the DUT (data items), via classes that specify that type of data. Each data item is derived (`extends`) from a base UVM class (**`uvm_sequence_item`**) and contains the data fields that describe it. Some of these (if not all) might be randomizable (declared with the `rand` or `randc` modifier) to indicate that a random instance of that data item can have different values for those data fields. We must also add the necessary `constraints` to ensure that the randomized data is valid and useful for verification. Since data items are defined as classes, we can extend them using inheritance to create multiple data item types with a restricted amount of code.

When we define a new UVM data item, we must include an explicit constructor that takes a `string` argument specifying the name of the data item instance. We must also provide as argument default the name of the class that defines that data item. The constructor implementation must call the parent constructor and pass it its argument. See Code 4.1 for an example.

Data items defined following UVM (i.e. derived from `uvm_sequence_item`) can inherit subroutines to automatically copy, print, compare, etc. those items, without the need for the user to provide custom code. To enable these subroutines for specific data fields of the data item, you must explicitly indicate that using UVM macros. These macros must be wrapped by the `` `uvm_object_utils_begin `` ... `` `uvm_object_utils_end `` pair. Data fields that do not have declared a macron in this fashion will not be included in these automatically inherited functions. Thus, remember to always add a macro for each data field. You can also specify some flags that allow you to customize how these field are used. For example, you can add the `UVM_NOCOMPARE` flag to indicate that the data field should not be used when comparing two data item instances. To see the technical rules of how to add data fields to these inherited functions, and what flags are available, see the literature. For an example see Code 4.1.

```
//defining new data item, it extends uvm_sequence_item
class my_data_item extends uvm_sequence_item;
  //data fields, can be randomized
  rand bit [7:0] var1;
  rand bit [15:0] var2;
   ...
  //constraints to make sure that data is randomized correctly
  constraint var1_c {var1 > 1; var1 < 250;}
  constraint var2_c {var2 != var1;}
   ...
  //explicit constructor that requires a string name as argument
  //default value set to the class name
  function new(string name = "my_data_item");
    //implementation only requires to call the super constructor and pass it
    the name argument
    super.new(name);
  endfunction
   ...
  //UVM macros to automate copy, compare, print, etc
  `uvm_object_utils_begin(my_data_item)
```

```
    `uvm_field_int(var1, UVM_ALL_ON + UVM_BIN)
    `uvm_field_int(var2, UVM_ALL_ON + UVM_NOCOMPARE)
  `uvm_object_utils_end
   ...
endclass : my_data_item
   ...
  //new data item instances, handle names and constructor string arguments
    match
  my_data_item di_inst = new("di_inst");
  my_data_item di_inst2 = new("di_inst2");
   ...
  //deep copy of di_inst into di_inst2, the name field ("di_inst2") is not
    copied
  di_inst2.copy(di_inst);

  //print the data item instance
  di_inst.print();
   ...
```

Code 4.1: SystemVerilog UVM Data Item Examples

4.1.1 Sequences

Single data items are a higher level representations of the bundles of signals that are sent to the DUT. Often times, however, we want to stimulate the DUT with a sequence of data items, perhaps representing a complex command (e.g. memory read) that cannot be represented as a single data item. UVM provides a mechanism for bundling together a sequence of individual data items. We can represent this sequence as an object (whose type is derived from **`uvm_sequence`**). Then, each time we instantiate an object of this type, we indicate that specific sequence of data items. Sequences are usually passed to sequencers (which we will see later), that send them to the DUT.

When you define a sequence, you must define the **`body()`** `task` that indicates the sequence of data items that composes it. There are multiple ways of defining this sequence of data items. Here, I will present the simplest way of doing so. More complex ways, that allow more control over sequences, can be found in the literature. The easiest (and most limiting) way of defining a sequence of data items in the `body()` is through the use of **`` `uvm_do_with(uvm_sequence_item seq, constraint const) ``** and **`` `uvm_do(uvm_sequence_item seq) ``**. These macros create the data item, wait until it is needed, randomize it, send it to the sequencer, and wait for the sequencer to signal that the data item was sent to the DUT. `` `uvm_do_with() `` also allows the user to specify a set of `constraint`s that will be imposed when the data item is randomized. An example of a simple sequence can be found below (Code 4.2).

Sequences are an extensive topic that needs a fair amount of effort to be covered in any level of detail. The reader is thus invited to consult the literature to actually learn how to define sequences of data items. This subsection serves only as an introduction to the topic.

```
//defining a new sequence of data items, it extends uvm_sequence
//must also provide as parameter the data item class that will compose the
    sequence (in our case my_data_item)
class my_sequence extends uvm_sequence #(my_data_item);
  //utility macro of the uvm_object (uvm_sequence, uvm_sequence_item)
  `uvm_object_utils(my_sequence)

  //explicit constructor that requires a string name as argument
  //default value set to the class name
  function new(string name = "my_sequence");
    super.new(name);
  endfunction

  //task that indicates the sequence of data items to be sent to the sequencer
  virtual task body();
    //my_data_item req was automatically created when my_sequence was
    parametrized
    //indicating to generate a new data item, randomized with the specified
    constraint, as the first element of the sequence
    `uvm_do_with(req, {var1 == 0;})

    //indicating to generate a new data item, randomized, as the next element
    of the sequence
    `uvm_do(req)
  endtask
   ...
endclass : my_sequence
```

Code 4.2: SystemVerilog UVM Sequence Examples

4.2 Simulation Cycle

UVM breaks down simulation into phases. These are executed sequentially. A phase must be fully completed for the next one to begin. The simulation phase sequence defined by UVM is:

- **`build_phase`**: build testbench components.
- **`connect_phase`**: connect testbench components.
- **`end_of_elaboration_phase`**: phase after testbench elaboration. Useful, for example, for printing the testbench topology.
- **`start_of_simulation_phase`**: phase right before simulation begins.
- **`run_phase`**: run simulation.
- **`extract_phase`**: gather information on the final state of the DUT.
- **`check_phase`**: check the gathered data.
- **`report_phase`**: analyze and report the results.
- **`final_phase`**: finish simulation.

The user can specify actions to be performed during any of these phases.[58] To do so, a specific subroutine must be defined and overwritten in the user code. For example, to define an action to be executed during the `connect_phase`, you must declare the `function void connect_phase(uvm_phase)` and specify the desired action inside the definition. Similar `void functions`, that take an `uvm_phase` as argument, can be specified for the other phases if one wants to specify behaviour to be executed during them. There are two exceptions however. One is for the `build_phase` and another for the `run_phase`. `function void build_phase(uvm_phase phase)` must call `super.build_phase(phase);` as the first statement of its definition. This is to ensure that the testbench is build top-down in a proper manner. The other exception is that the `run_phase` is a `task` and needs to be defined as such, i.e. `task run_phase(uvm_phase phase)`. This is the only subroutine that is not executed in zero time. The `run_phase` can be further subdivided into 12 sub-phases, find out more in the literature.

Last, I will mention that in order to start the UVM simulation, the user must call the **`run_test()`** subroutine defined in the **`uvm_pkg::*`** package.

[58] It is also possible for the user to define custom phases.

4.2.1 End of Simulation

UVM has an elegant mechanism for indicating the end of simulation. When the simulation begins, the simulator tries right away to end it. Individual sequences are required to raise objections, that prevent the simulator from ending the simulation, at the beginning for their execution. The sequences are then required to drop these objections once their execution has finished. There are multiple ways of raising and dropping objections. The easiest one is letting UVM 1.2 automatically handle objections for you. To do so, simply add the statement **`set_automatic_phase_objection(1);`** to the sequence constructor (`new()`). For the full list of possibilities consult the literature.

Once all sequences have dropped their objections, the simulation would stop immediately. You can specify a delay, called *drain time*, between the last objection drop and the end of the simulation. This way, you can allow outstanding items to finish propagating. Once again, there are multiple ways of specifying a drain time, the easiest one being adding the `run_phase() task`, containing the following two statements, to the test that is executed:[59]

```
uvm_objection objection = phase.get_objection();

objection.set_drain_time(this, 50ns);
```

Of course, you can specify a different drain time than the `50ns` specified above.

4.3 Interface UVCs

A fundamental testbench component is the interface UVM verification component (or simply interface UVC). This component is responsible for driving stimulus into the DUT, capturing both stimulus as well as DUT response, and sending it to other testbench components (e.g. the scoreboard) for analysis. The interface UVC can be developed as a verification IP (VIP) and used wherever the specific communication interface is used.

As was stated previously, in UVM stimulus and testbench are kept separate. We have see that data items extend `uvm_sequence_item`. This implies that both the constructor and the class macros must follow the rules that we have discussed in the previous section. For testbench components, on the other hand, these rules do not apply. This is because the different components of the

[59] We will see test later in this work.

testbench are not extended from `uvm_sequence_item`. Instead, there is a super-class (the **`uvm_component`**) that has its own rules regarding the constructor and the macros, from which all the testbench components derive. The constructor and macros rules for these components will be illustrated below (Code 4.3). You can see that the constructor must be declared with two arguments, one indicating the name (**`string name`**) of the component, and the other the component (`uvm_component parent`) that created the current instance. The constructor must call, as the first statement, the super-class' constructor (`super.new(name, parent);`). Furthermore, there is not need to add the data fields to the utility macro. A simple `` `uvm_component_utils(CLASS_NAME) `` must be declared.

The interface UVC has a standard architecture. A schematic overview of this architecture can be found below (Figure 4.1). Code examples for each of the interface UVC components can also be found below (Code 4.3, Code 4.4, Code 4.5, Code 4.6 and Code 4.7). Some details about the code is provided as comments but, for the sake of brevity, the full explanation of is not presented. For more details on how to properly build these components see the literature.

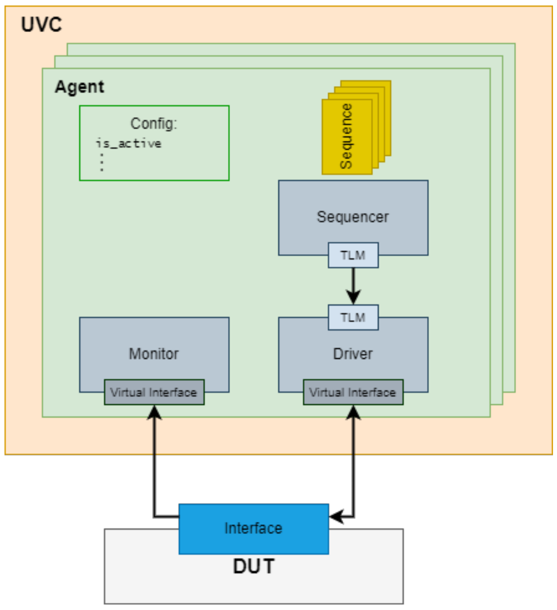

Figure 4.1: Standard Interface UVC Architecture in UVM.

At the top level we have the UVC (`extends uvm_env`, which itself is derived from `uvm_component`). This component instantiates one or more Agents and configures them.

```
//defining an interface UVC, extends uvm_env
```

```
class my_interface_uvc extends uvm_env;
  //declaring the agent of the UVC
  my_iagent iagent_inst;

  //utility macro of the uvm_component
  `uvm_component_utils(my_interface_uvc)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //build the agent instance
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    iagent_inst = new("iagent_inst", this);
  endfunction
   ...
endclass : my_interface_uvc
```

Code 4.3: SystemVerilog UVM Interface UVC Examples

4.3.1 Agents

Agents are the next level down the component hierarchy. These components are obtained by extending the **`uvm_agent`** class (which is a sub-class of `uvm_component`). Agents can be *active*, meaning that they can send stimulus to the DUT, or *passive*, meaning that they can only monitor the communication channel. They instantiate, connect and configure the components at the next (last) layer of the hierarchy.

```
//defining an interface UVC agent, extends uvm_agent
class my_iagent extends uvm_agent;
  //declaring the components of the agent
  my_monitor mon_inst;
  my_driver drv_inst;
  my_sequencer sqr_inst;

  //utility macro of the uvm_component
  `uvm_component_utils(my_iagent)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //build the agent components
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    mon_inst = new("mon_inst", this);

    //instantiate the driver and sequencer if the agent is not passive
    //is_active is an inherited enum instance that can be either UVM_ACTIVE or
     UVM_PASSIVE
    if (is_active == UVM_ACTIVE) begin
      drv_inst = new("drv_inst", this);
      sqr_inst = new("sqe_inst", this);
    end
  endfunction

  virtual function void connect_phase(uvm_phase phase);
  if (is_active == UVM_ACTIVE) begin
    //connect is a TLM method that connects a port to an export
    drv_inst.seq_item_port.connect(sqr_inst.seq_item_export);
  end
  endfucntion
   ...
endclass : my_iagent
```

Code 4.4: SystemVerilog UVM Interface UVC Agent Examples

4.3.2 Monitors

At the lowest level of the component hierarchy we have: the monitor, the driver and the sequencer. The monitor (`extends` **`uvm_monitor`**, which itself is derived from `uvm_component`) is responsible for keeping track of the data that goes into and comes out of the DUT. This data can then be analyzed locally or sent to other components.

```
//defining an interface UVC monitor, extends uvm_monitor
class my_monitor extends uvm_monitor;
  //utility macro of the uvm_component
  `uvm_component_utils(my_monitor)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion
   ...
endclass : my_monitor
```

Code 4.5: SystemVerilog UVM Interface UVC Monitor Examples

4.3.3 Drivers

The driver (`extends` **`uvm_driver`**, which itself is derived from `uvm_component`) is responsible for stimulating the DUT.[60] The monitor and driver should be completely independent to allow for both passive and active agents. This component asks for a data item to be sent to the DUT. This data item is provided at a high level of abstraction, through a class instance, and must be converted into signals by the driver.[61]

```
//defining an interface UVC driver, extends uvm_driver
//must also provide as parameter the data item class that will be driven into
    the DUT (in our case my_data_item)
class my_driver extends uvm_driver #(my_data_item);
  //utility macro of the uvm_component
  `uvm_component_utils(my_driver)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //the action performed by the driver during the run phase
  virtual task run_phase(uvm_phase phase);
    forever begin
      //ask the sequencer for the next data item
      //my_data_item req was automatically created when my_driver was
    parametrized
      seq_item_port.get_next_item(req);
```

[60]Sometimes the driver is also referred to as Bus Functional Model (BFM) in the literature.

[61]In practice the function that translates a high-level representation of the data into the associated signals in usually implemented inside the virtual interface that connects the interface UVC and the DUT. This is particularly important for reusability and hardware acceleration. See details in the literature.

```

      //method, likely defined in the virtual interface, to send the data item
     to the DUT
      send_to_dut(req);

      //indicate the sequencer that the data item was driven successfully
      seq_item_port.item_done();
    end
  endtask
   ...
endclass : my_driver
```

Code 4.6: SystemVerilog UVM Interface UVC Driver Examples

4.3.4 Sequencers

The data item is asked from, and provided by, the sequencer (`extends` **`uvm_sequencer`**, which itself is derived from `uvm_component`). This component takes valid sequences of date items, randomizes them, and sends them to the driver. The agent is responsible for connecting the driver and the sequencer via a TLM connection. The topic of TLM will not be presented in this work. It suffices to know that the sequencer has a built-in TLM connection (namely **`seq_item_export`**) and defines some communication method (e.g. **`get_next_item()`**, that sends the next sequence item to the driver, and **`item_done()`**, used by the driver to signal to the sequencer that the data item was driven into the DUT). Analogously, the driver contains a built-in TLM connection (namely **`seq_item_export`**).

```
//defining an interface UVC sequencer, extends uvm_sequencer
//must also provide as parameter the data item class that will be sent to the
    driver (in our case my_data_item)
class my_sequencer extends uvm_sequencer #(my_data_item);
  //utility macro of the uvm_component
  `uvm_component_utils(my_sequencer)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion
   ...
endclass : my_sequencer
```

Code 4.7: SystemVerilog UVM Interface UVC Sequencer Examples

4.4 Configuration

Sometimes we might need to change the value of certain parameters, for example, setting the `is_passive` field of an agent in different tests. We could manually go through the code and change the desired values. This is however tedious and error prone. Luckily, UVM provides a mechanism that allows us to change the values of parameters, from a higher hierarchical level, without having to manually modify the source code or pass the desired values as method arguments. This mechanism is the configuration database. It allows you to specify the values that certain parameters should have upon creation (e.g. whether `is_active` should be `UVM_ACTIVE` or `UVM_PASSIVE`) and, when the instances are created, the specified values will be set. The syntax that is used to specify the values of parameters via this mechanism is:

```
uvm_config_db#(<type>)::set(uvm_component context,
string instance, string field, T value)
```

where `<type>` is the configure property type. For integral values the type is **`uvm_bitstream_t`**. Check the literature for the other types. `context` is an `uvm_component` that is the root from which the rest of the expression will be evaluated. `instance` is a `string` with the hierarchical name of the object whose parameter (data field) should be changed. `field` is the `string` indicating the name of the data field that needs to be changed. Last, `value` specifies the new value the parameter should have. Its type `T` must be appropriate for the specific data field it modifies. As an example, `uvm_config_db#(uvm_bitstream_t)::set(this, "my_tb_inst.my_ivc_inst.iagent_inst", "is_active", UVM_PASSIVE);` will evaluate the hierarchical path `my_tb_inst.my_ivc_inst.iagent_inst.is_active` from the context of the current object (`this` used as `context`) and set this data field to `UVM_PASSIVE`. It is important to know that only the data fields that have been registered with UVM macros can be modified this way. If the hierarchical path, or the `value` type, is not correct, the configuration will be ignored. To check which configuration have not been used, you can call the **`check_config_usage()`** method during the `check_phase`.

These also exist shortcuts for commonly used configuration and type commands. For example, you can use `uvm_config_int` instead of `uvm_config_db#(uvm_bitstream_t)`. Find out more in the literature. There you can also find information on how to specify the parameter name as regular expressions, allowing you to change multiple parameters with a single configuration statement. There also exist other configuration methods, besides `set`, and another configuration statement syntax (though it is deprecated). Once again, consult the literature to find out more.

4.5 Factory and Type Overrides

Sometimes you might want to change not the value of a specific parameter, but the type of an object. For example, consider the case where you send the DUT a specific data item in one test, and want to send another data item, that is a sub-class of the previous one, in a different test. To do this, you would have to comb through the code and adapt it to each specific test. In UVM there is actually no need to do so. There is a mechanism that allows you to change the types of all the object of a particular type, as well as change the types of specific objects. This mechanism is the *factory* enabled *type override*. When you create a new object, instead of using the `new()` constructor, you can call the factory method **`<object_type>::type_id::create(string name, uvm_component parent)`**, where `object_type` is the class of the object.[62] The factory construct then checks if you have specified a type or instance override and, if so, the actual type of the object will be set to the user specified one. In detail, use:

```
set_type_override_by_type(<source_type>::get_type(),
<target_type>::get_type());
```

to specify that all the instances of `source_type` should be created as type `target_type` instead. Use:

```
set_inst_override_by_type(string <instance_name>,
<source_type>::get_type(), <target_type>::get_type());
```

[62]Note that, while the `new()` constructor required a `string name` and `uvm_component parent` for `uvm_components`, and only a `string name` for `uvm_sequence_items` and `uvm_sequences`, the factory method `create()` requires a `string name` and `uvm_component parent` whenever the object is created inside a `uvm_component`. Otherwise, `parent` can be omitted.

to change the type of `instance_name` from `source_type` to `target_type` upon creation.[63] Then, when you `create()` the object, the type will be set based on the specified type override (if you have specified one). Consider the following example: we call `set_type_override_by_type(my_agent::get_type(), my_agent_1::get_type());`. Now, instead of calling `my_agent agent_inst = new("agent_inst", this);` we call `my_agent agent_inst = my_agent::type_id::create("agent_inst", this);`. Thanks to the type override, the type of `agent_inst` will actually be `my_agent_1` and not `my_agent`.

Even though in this work, for the sake of simplicity, we use `new()` to instantiate new objects, this should not be done in actual testbenches. As a general UVM rule, it is important to use the factory method `create()` instead of the `new()` constructor.[64] Furthermore, type overrides work only for types registered with the factory. These are the inherited UVM types, as well as those you have included the `` `uvm_object_utils() ``, `` `uvm_component_utils() `` macros for.

Find out more about this topic, including a different syntax for specifying overrides and rules for multiple type overrides, in the literature.

4.6 Module UVCs

A coverage model and a checker, that makes sure that the DUT responds to stimulus in the expected way, are fundamental components of our testbench. These components are usually bundled together is a single testbench entity, the *module UVC*. This UVC once again `extends uvm_env` and can be used wherever the DUT it was developed for is deployed. Note that, unlike interface UVCs, the module UVC is much less flexible and reusable since it does not model a particular information exchange protocol, but rather, a family of DUT configurations. What we have discussed for interface UVCs also apply here (e.g. how to declare an UVC and how to deploy it in the testbench). However, the module UVC is not used to stimulate the DUT and monitor its response. It is instead connected to the testbench interface UVCs' monitors to gather stimulus/response data for elaboration. The module UVC then analyzes the data it has captured and determines whether the DUT is working properly and updates the coverage progress. The key element of the module UVC is the scoreboard as it implements these functionalities. Scoreboards separate and group together interface UVCs that need

[63] `instance_name` can also be a regular expression, allowing you to override multiple objects with a single statement.

[64] This is not always true (e.g. for TLM connections), see more details in the literature.

to be elaborated concurrently. In Figure 4.2 you can see the typical architecture of a module UVC. You can also find a code example below (Code 4.8).

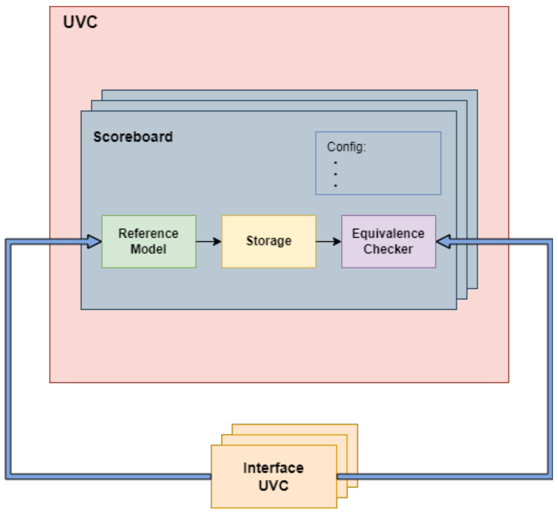

Figure 4.2: Standard Module UVC Architecture in UVM.

```
//defining a module UVC, extends uvm_env
class my_module_uvc extends uvm_env;
  //declaring the scoreboard of the UVC
  my_scoreboard sb_inst;

  //utility macro of the uvm_component
  `uvm_component_utils(my_module_uvc)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //build the scoreboard instance
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sb_inst = new("sb_inst", this);
  endfunction
   ...
endclass : my_module_uvc
```

Code 4.8: SystemVerilog UVM Module UVC Examples

4.6.1 Scoreboards

Besides the coverage model, that can be scoreboard or module UVC specific, a typical scoreboard has three components: a reference model, a storage and an equivalence checker. The reference model describes the desired input-output behaviour of our DUT. It is usually developed in SystemVerilog, C or SystemC. The storage is needed to maintain the data that is to be processed. Last, the equivalence checker compares the DUT and reference model outputs and acts accordingly. Developing any of these three requires some thought and effort, as such, a further discussion on these topics is beyond this introduction to UVM.[65]

Scoreboards are extended from **`uvm_scoreboard`** and require some specific code to setup their communication channel with interface UVC monitors.[66] The basic scoreboard must declare and instantiate an **`uvm_analysis_imp`** and define the associated **`write() function`** that specifies how to monitor should send data to the scoreboard. You must also modify the monitors to enable them to communicate with scoreboards. The basic modification consists in declaring and instantiating an **`uvm_analysis_port`**. Code 4.9 and Code 4.10 show how to do so for scoreboards and monitors respectively. The examples presented below are very basic and only indicate how to connect a monitor to a scoreboard. It is also possible to establish more complex connections (e.g. multiple monitors to one scoreboard). To see how, consult the literature as it is not an obvious extension of what we have seen here.

Last, I will mention that once the interface and module UVCs have been properly developed, we must connect the monitor `uvm_analysis_port` to the scoreboard `uvm_analysis_imp`. This is done in the testbench during the `connect_phase` as shown in Code 4.11.

```
//definition of a scoreboard, extends uvm_scoreboard
module my_scoreboard extends uvm_scoreboard;
  //utility macro of the uvm_component
  `uvm_component_utils(my_test)

  //declaring an uvm_analysis_imp named ap_in
  //it is parametrized with the data item (my_data_item) and the scoreboard (
```

[65]For example, you will find information on how UVM helps you implement the equivalence checker thanks to a build-in `uvm_comparer` and its associated methods. Or, the need for cloning data items passed to the scoreboard.

[66]The communication channel is established according to the TLM protocol, which is not covered in this work.

```
    my_scoreboard)
  uvm_analysis_imp #(my_data_item, my_scoreboard) ap_in;

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);

    //instantiating the uvm_analysis_imp
    ap_in = new("ap_in", this);
  endfucntion

  //specifies how the monitor should send data to the scoreboard
  function void write(input my_data_item d_item);
     ...
  endfunction
   ...
endmodule : my_scoreboard
```

Code 4.9: SystemVerilog UVM Scoreboard Examples

```
//defining an interface UVC monitor, extends uvm_monitor
class my_monitor extends uvm_monitor;
  //utility macro of the uvm_component
  `uvm_component_utils(my_monitor)

  //declaring an uvm_analysis_port named ap_out
  //it is parametrized with the data item (my_data_item)
  uvm_analysis_port #(my_data_item) ap_out;

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);

    //instantiating the uvm_analysis_port
    ap_out = new("ap_out", this);
  endfucntion
   ...
  my_data_item d_item;
   ...
  //writing the data item d_item to the scoreboard
  ap_out.write(d_item);
   ...
endclass : my_monitor
```

Code 4.10: SystemVerilog UVM Monitor Configured for a Scoreboard Examples

4.7 Testbenches

The testbench is a verification component (usually derived from **`uvm_env`**, as UVM does not have a `uvm_testbench` sub-class) that instantiates and connects other `uvm_envs`, e.g. UVCs. See an example of a testbench architecture below (Figure 4.3). Since the testbench is a `uvm_component`, its declaration follows the

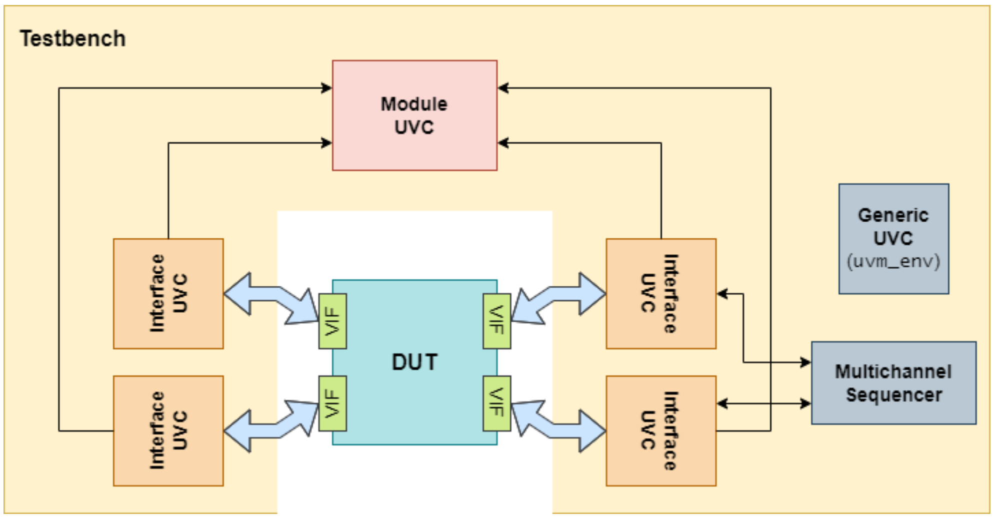

Figure 4.3: Example of a Testbench Architecture in UVM.

typical component macros and constructors that we have seen for UVCs.[67] For an example of a testbench see Code 4.11. In this example, note how the UVCs were instantiated in the `build_phase` and connected in the `connect_phase`. UVM components have access to some `functions` that `return` a `string` indicating the instance name of the component, component `class` type and the full hierarchical pathname of the instance. These `functions` are: **`get_name()`**, **`get_type_name()`** and **`get_full_name()`** respectively.

```
//definition of a testbench, extends uvm_env
class my_testbench extends uvm_env;
  //UVCs instantiated in the testbench
```

[67]The full list of `uvm_components` is: `uvm_monitor`, `uvm_driver`, `uvm_sequencer`, `uvm_scoreboard`, `uvm_agent`, `uvm_env` and `uvm_test`.

```
  my_interface_uvc my_ivc_inst;
  my_interface_uvc my_ivc_inst1, my_ivc_inst2;
  my_module_uvc my_mvc_inst;

  //declaring a multichannel sequencer (see the rest of this section)
  my_mc_sequencer my_mc_sqr;

  //utility macro of the uvm_component
  `uvm_component_utils(my_testbench)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //instantiate the UVCs in the build_phase
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    my_ivc_inst = new("my_ivc_inst", this);
    my_ivc_inst = new("my_ivc_inst1", this);
    my_ivc_inst = new("my_ivc_inst2", this);
    my_mvc_inst = new("my_mvc_inst", this);

    //building the multichannel sequencer
    my_mc_sqr = new("my_mc_sqr", this);
  endfunction

  //connect the UVCs in the connect_phase
  virtual function void connect_phase(uvm_phase phase);
    //connecting the multichannel sequencer to the sequencers of the
    individual interface UVCs
    my_mc_sqr.sqr1 = my_ivc_inst1.iagent_inst.sqr_inst;
    my_mc_sqr.sqr2 = my_ivc_inst2.iagent_inst.sqr_inst;

    //connecting the monitor uvm_analysis_port with the scoreboard
    uvm_analysis_imp
    my_ivc_inst.iagent_inst.mon_inst.ap_out.connect(my_mvc_inst.sb_inst.ap_in)
    ;
     ...
  endfunction
   ...
endclass : my_testbench
```

Code 4.11: SystemVerilog UVM Testbench Examples

4.7.1 Tests

There is one component that is above the testbench in the UVM hierarchy of classes, the test (which `extends uvm_test`). The test is the bridge between the two aspects of the UVM, namely stimulus generation and testbench. As such, this component (`uvm_test` is a sub-class of `uvm_component`), on the one hand, controls the stimulus generated by UVCs and, on the other, instantiates the testbench. Usually there are multiple tests developed, and contained in a test library, that are meant to verify certain functionalities or simulate different environments for the DUT. An example of a test can be found below (Code 4.12).

```
//definition of a test, extends uvm_test
class my_test extends uvm_test;
  //testbench instantiated in the test
  my_testbench my_tb_inst;

  //utility macro of the uvm_component
  `uvm_component_utils(my_test)

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion

  //instantiate the testbench in the build_phase
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    my_tb_inst = new("my_tb_inst", this);

    //setting the default test sequence
    uvm_config_wrapper::set(this, "my_tb_inst.iagent_inst.sqr_inst.run_phase",
     "default_sequence", my_sequence::get_type());

    //setting the default test multichannel sequence for the multichannel
    sequencer
    uvm_config_wrapper::set(this, "my_tb_inst.my_mc_sqr.run_phase", "
    default_sequence", my_mc_sequence::get_type());
  endfunction

  //connect the testbench in the connect_phase
  virtual function void connect_phase(uvm_phase phase);
     ...
  endfunction

  //phase right after the connection phase, when the testbech topology is set
  virtual function void end_of_elaboration_phase(uvm_phase phase);
    //can execute this statement to print the structure of the testbench
    uvm_top.print_topology();
  endfunction
   ...
endclass : my_test

//test class that will inheret the testbench of my_test
class my_test_1 extends my_test;
   ...
endclass : my_test_1
```

Code 4.12: SystemVerilog UVM Test Examples

You can run a test by calling the **`run_test()`** `task` from the top `module` of your hierarchy. You must pass it the name of test as an argument. See example of Code 4.13. This is the most basic way of running a test, more flexible ways exist. Consult the literature for details.

```
module my_top;
  //importing UVM with all the class structure
  import uvm_pkg::*;

  //including UVM macros
  `include "uvm_macros.svh"

  //including the testbench and tests
  `include "my_testbench.sv"
  `include "my_test.sv"
   ...
  intitial begin
    run_test("my_test_1");
  end
   ...
endmodule : my_top
```

Code 4.13: SystemVerilog UVM Running Test Examples

4.7.2 Multichannel Sequences

Often times we want to coordinate the sequences driven into the DUT by each individual interface UVC. This can be achieved with the use of a *multichannel sequencer* (sometimes also called virtual sequencer). This type of sequencer works very similarly to the other sequencers we have seen before (e.g. it `extends uvm_sequencer`). As you can see in this section's code (Code 4.11, Code 4.12, Code 4.14, Code 4.15), there are, however, a few key differences. For example, a virtual sequencer must have access to the references of the sequencers it needs to coordinate.

This type of sequencer drives a particular type of sequence, a *multichannel* (or virtual) *sequence*. The latter needs to have access to the individual sequences it will interlace. It must also use the **`` `uvm_do_with(uvm_sequence_item seq, uvm_sequencer sqr, constraint const) ``** and **`` `uvm_do(uvm_sequence_item seq, uvm_sequencer sqr) ``** macros, instead of the ones presented previously in this work. Furthermore, the multichannel sequence needs to know the multichannel sequencer it will be executed by. To indicate the sequencer, you need to use the **`` `uvm_declare_p_sequencer() ``** as shown below (Code 4.15). It is of course possible to define more sophisticated multichannel sequencers/sequences than the ones illustrated in this section. As always, see the literature for details.

A multichannel sequencer is a component that, unlike non-multichannel sequencers, is not part of any UVC. Instead, it is instantiated directly in the testbench as seen in Code 4.11.

```
//defining a new multichannel sequencer, extends uvm_sequencer
//note that, unlike non-multichannel sequencers, the class is not parametrized
class my_mc_sequencer extends uvm_sequencer;
  //utility macro of the uvm_component
  `uvm_component_utils(my_mc_sequencer)

  //reference to the sequencers that will be coordinated
  my_sequencer sqr1, sqr2;

  //constructor of the uvm_component
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfucntion
   ...
endclass : my_mc_sequencer
```

Code 4.14: SystemVerilog UVM Multichannel Sequencer Examples

```
1  //defining a new multichannel sequence of data items, extends uvm_sequence
2  //note that, unlike non-multichannel sequences, the class is not parametrized
3  class my_mc_sequence extends uvm_sequence;
4    //utility macro of the uvm_object (uvm_sequence, uvm_sequence_item)
5    `uvm_object_utils(my_mc_sequence)
6
7    //macro that declares the type of multichannel sequencer the multichannel
       sequence will be executed by (in our case my_mc_sequencer)
8    `uvm_declare_p_sequencer(my_mc_sequencer)
9
10   //declaring the sequences that will run on the individual sequencers
11   my_sequence m_seq1, m_seq2;
12
13   //constructor of the uvm_object
14   function new(string name = "my_mc_sequence");
15     super.new(name);
16   endfunction
17
18   //task that indicates the sequence of data items to be sent to the sequencer
19   virtual task body();
20     //my_mc_sequencer p_sequencer was automatically created when you declared
       the macro at line 8
21     //indicating to run sequence m_seq2 on sequencer p_sequencer.sqr1, with
       the var1 == 0 constraint
22     `uvm_do_on_with(m_seq2, p_sequencer.sqr1, {var1 == 0;})
23
24     //indicating to run sequence m_seq1 on sequencer p_sequencer.sqr2
25     `uvm_do_on(m_seq1, p_sequencer.sqr2)
26   endtask
27    ...
28 endclass : my_mc_sequence
```

Code 4.15: SystemVerilog UVM Multichannel Sequence Examples

4.7.3 Logging

UVM provides some macros for automatically logging information. These are:

- **`` `uvm_info(string id, string message, int verbosity = UVM_MEDIUM) ``**
- **`` `uvm_warning(string id, string message) ``**

- **``uvm_error(string id, string message)``**
- **``uvm_fatal(string id, string message)``**

where `id` identifies the message, `message` is the message to be displayed and `verbosity` is one among: **`UVM_NONE`**, **`UVM_LOW`**, **`UVM_MEDIUM`** (default), **`UVM_HIGH`**, **`UVM_FULL`**, **`UVM_DEBUG`**. A `uvm_info` message will be displayed only if its verbosity is not greater than the global verbosity of the test. The latter is set to `UVM_MEDIUM` by default. In the literature you can find information on how to change it. You can also find information on how to modify the default behaviour upon message display.

A `uvm_fatal` will terminate simulation once the message is displayed.

Part II

Automated Testbench Generation

Chapter 5

Tools for automatic code generation

As we have seen in the previous chapter, in order to keep up with the increasing complexity of the design, and thus the required verification effort, we have adopted a standard verification architecture. This choice brings with it a lot of advantages, but it also comes with drawbacks. One of the most notable ones being the significant code overhead needed. Writing even small testbenches or simply setting up the required code infrastructure, without performing any verification, requires writing a lot of repetitive and tedious (thus error prone) code. This second part of the dissertation presents the thesis project, which aims to help alleviate this burden. This chapter present the fundamental tools that were used. The last (6th) chapter describes how they were used and the achieved results.

5.1 Metamodels

Take a look at Figure 5.1. Clearly only one circuit sticks out by making sense. Why is that so?

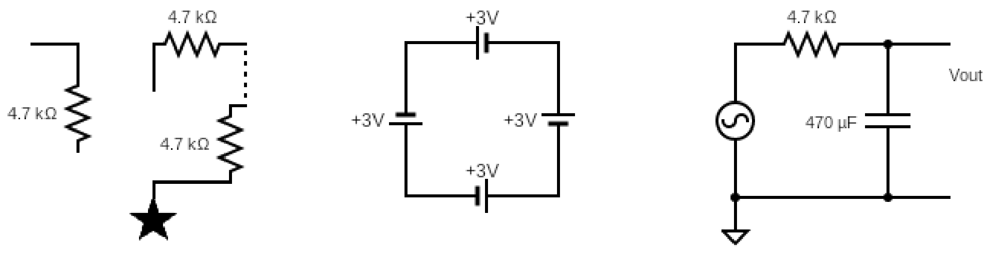

Figure 5.1: Example of "electric circuits".

The leftmost circuit does not make any sense whatsoever. It uses graphical elements that have no meaning in the context of circuit diagrams (e.g. the star and the dashed line). We cannot easily interpret this diagram and the system it tries to model. The diagram in the middle, on the other hand, is quite easily readable since it uses standard circuit notation. Despite this, the circuit does not make sense once again. This time however it is due to the improper use of diagram elements. It violates the rules we need to follow in describing the lumped model of an electric system. The rightmost circuit does not present such problems. We can easily read it and interpret the physical system it models. This example shows that when we want to describe the model of a system we must: (*i*) make sure that we use the constructs that have been agreed upon and have meaning; (*ii*) make sure not to violate the rules the model must obey. Before continuing, let's briefly discuss what modelling actually is.

5.1.1 Modeling

We are often interested in studying subsets of reality (e.g. electric systems). These can be physical (e.g. electric systems) or not (e.g. verification testbenches). To avoid dealing with their complexity, and to hide the unnecessary detail, we develop restricted representations: models. The same reality can be modeled in different ways based on a particular purpose. For example, given the same electric system, we can describe its electric characteristics (with a circuit diagram) and its physical layout (layout diagram). Thus, a model can be seen as a simplified representation of (a subset of) reality, with a specific purpose. Since a model is useful precisely because it is a simplification, we need to make sure that the model's complexity is adequate to it's purpose. We can have different models, of varying complexity and purpose, that describe the same reality. It is our job to pick the one that suites our needs best.

5.1.2 Metamodeling

Definition 5.1.1 (Metamodel) *The set of constructs, such as entities, structures, semantics and constraints, that are used to describe, with a specific purpose, subsets of reality is called a **metamodel**.*[68]

[68]The entities of a metamodel are the basic components of a model (e.g. resistors, wires, generators, etc.). The structures describe how entities can be composed to form other entities (e.g. the entity generator is composed of the entities ideal generator and equivalent series resistor). The semantics describe the meaning of entities and models (e.g. what is a resistor).

A portion of reality is described by a model, which must conform to a metamodel.

Definition 5.1.2 (Modeling Language) *A **modeling language** specifies the concrete syntax we use to describe models.*

For example, a modeling language might specify whether we use the IEC standard or the American standard to represent resistors (see Figure 5.2), what are the symbols for generators, capacitors, etc.

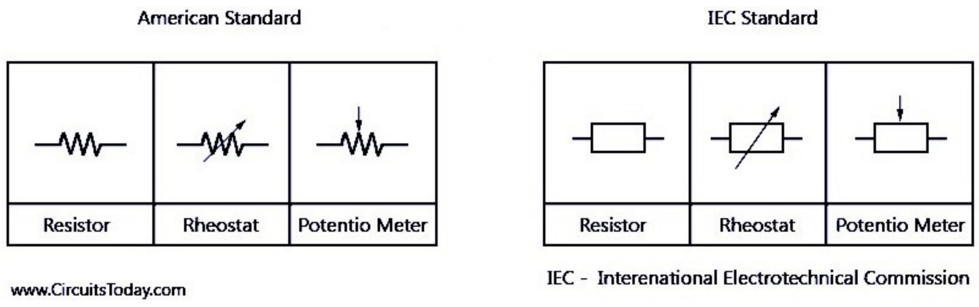

Figure 5.2: IEC and American Resistor Notations.

In conclusion, we can express reality with models, these are described using a modeling language and must conform to a metamodel. Thus, a metamodel can be seen as a special type of model, one that describes the abstract syntax of a modelling language.[69] We can now explain why the two circuit diagrams from Figure 5.1 do not make sense: they do not conform to the metamodel we implicitly learned when studying circuit analysis. The circuit in the middle violates this metamodel's constraints, the one on the left also does not use the appropriate modeling language (e.g. it uses the star symbol).

The metamodel itself must conform to a metamodel, the **meta-metamodel**. The latter is represented via a **metamodeling language**. Meta-metamodels are powerful enough to describe themselves in their own metamodeling language.

Take these definitions with a grain of salt as they are sometimes used loosely. You might find slightly different versions elsewhere. For example, some definitions might state that the constraints a model needs to satisfy are part of the modeling language and not the metamodel. The morale is that we use models to describe reality with a certain purpose. These models must conform to a set of rules,

[69]The concrete syntax is the way in which we represent things. For example, the symbol that we use to describe a resistor or the keywords we use to define SystemVerilog ports. The abstract syntax specifies the constructs behind these symbols. We can change the keywords for specifying ports (changing the concrete syntax) but keep the same underlying meaning (same abstract syntax).

which we can summarize in a metamodel. We represent models following the concrete syntax of modeling language.

5.2 Eclipse Modeling Framework

In practical terms a metamodel describes the models that we can create. For example, we might have a metamodel that describes all the legal circuit diagrams. The **Eclipse Modeling Framework** (**EMF**) allows us to work with metamodels. EMF is based on the *Java Programming Language* [19][20][21] and the *Eclipse IDE*, to find out more about EMF and Eclipse visit [22] and [23] respectively. At [24] you will find a variety of tutorials on metamodeling. In particular, *Ecore* is the meta-metamodel at the heart of EMF. Tutorial [25] shows how to use it to define metamodels, which Ecore calls domain models.

5.2.1 Defining an Ecore Domain Model

The gist of it is that we use a system that is very similar to entity-relationship modeling of relational databases.[70] Furthermore, this system adheres to the *Object Oriented Programming* (*OOP*) design philosophy. The basic structure is:

- **Entities** are represented either as *datatypes*, *enumerations* or *classes* (concrete, abstract of interfaces).

- Entities have associated **features** that describe them, these can be constants (called *literals*), functions (called *operations*) or *attributes*.

- Entities form **relations**. These can be *hierarchical* (e.g. entity A is a super-class of entities B and C), *compositional* (e.g. entity A is is made of two entities B) or *referential* (e.g. entity A knows of entities B and C). Referential relations can be *unidirectional* (e.g. entity A knows of entity B) or *bidirectional* (e.g. both entities A and B know of each other). Relations do not have to necessarily be one-to-one (e.g. entity A references only one entity B). They can also be one-to-many (e.g. entity A references multiple entities B) or many-to-many (e.g. multiple entities A and B reference each other bidirectionally).

[70]If not familiar with entity-relationship modeling worry not as it is not necessary to understand Ecore.

Below (Figure 5.3) you can see a very simple and limited metamodel of electric circuits. This metamodel is incomplete, for example it does not contain the constraints that would prevent us from connecting two DC voltage generators in parallel. Its purpose is simply to illustrate Ecore. In the example you can see

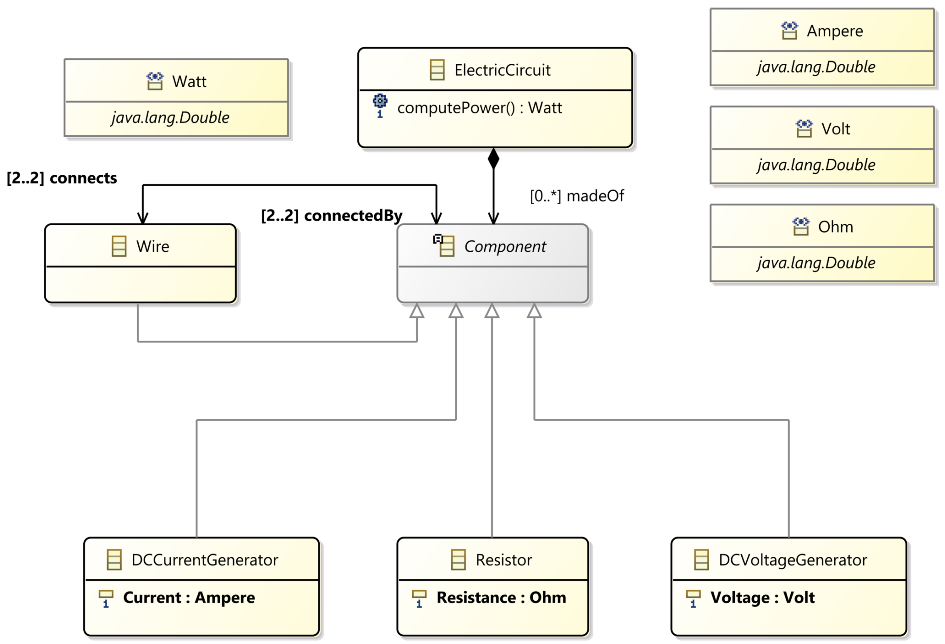

Figure 5.3: Simplified Electric Circuit Ecore Domain Model.

the definition of four datatypes: `Watt`, `Ampere`, `Volt` and `Ohm`. These are used as the datatypes of the various attributes and operations of the different entities. The entity at the highest hierarchical level is `ElectricCircuit`. You can see that it has a `computePower()` operation (that returns `Watts`) which might, for example, compute the power dissipated by the circuit. This entity is `madeOf` zero or more `Components` (abstract class). The latter is the super-class of the entities `Wire`, `Resistor`, `DCCurrentGenerator` and `DCVoltageGenerator`. `Resistor` has one attribute named `Resistence` (in `Ohms`) that must be specified. Analogously `DCCurrentGenerator` and `DCVoltageGenerator` have the `Current` (in `Ampere`) and `Voltage` (in `Volt`) mandatory attributes. When defining an attribute you must choose whether it can appear zero or more times in the related entity. In our case each attribute must appear exactly one time. `Wire` does not have attributes, it however has a bidirectional referential relation with `Component`. Each `Wire` `connects` from a minimum of 2 to a maximum of 2 `Components`. Each `Component` is `connectedBy` a minimum of 2 to a maximum of 2 `Wires`. This metamodel is clearly, as stated before, very limited and not a whole lot useful. It will however describe circuit diagrams made of resistors and DC current and voltage generators connected in series (without considering the constraints of these connections).

5.2.2 Use of Metamodeling Tools

What was hopefully shown in this section is that we have a tool for abstracting out models. This will turn useful in the next chapter when we will discuss abstracting out the testbench architecture of UVM. As we have seen, UVM follows a standard architecture for defining testbenches and their components. We can thus summarize this architecture, or parts of it, as an Ecore domain model. We might now wonder how is this useful. Before delving into the next sections, let's illustrate how it is so. The next tool that we will introduce is Sirius. The latter will allow us to define graphical interfaces that can be used to create models which conform to an Ecore domain model. In other words, Sirius is an EMF-based tool for defining modeling languages. Last, we will introduce Acceleo. This tool allows us to generate arbitrary user defined code, based on models specified using a modeling language developed with Sirius. An example of a project that illustrates the synergy of these tools might be: (*i*) develop a metamodel that describes all the legal circuit diagrams (EMF); (*ii*) develop a graphical interface that enables the user to specify circuit diagrams consistent with the previous metamodel (Sirius); (*iii*) translate the user specified circuit diagrams into text form (e.g. XML) that can be used by some other software (e.g. text-based circuit simulator) (Acceleo).

5.3 Sirius

As stated before, **Sirius** is a tool that allows us to specifier the modeling language of an Ecore domain model. It is fairly extensive and relies on Eclipse's *Graphical Modeling Framework* (*GMF*), a framework for developing *Graphical User Interfaces* (*GUI*s) and other visual elements. Sirius' documentation is available at [26][27]. Tutorials [28][29][30][31][32] provide a quick guide to adopting this tool. Sirius is non trivial, as such, a comprehensive overview is beyond the scope of this work. The above mentioned tutorials should however provide a good starting point. Here I will limit myself to describe the general idea of how Sirius works and introducing some basic features and notions. A few more advance features will be mentioned in the next chapter.

5.3.1 Overview

Sirius allows us to define modeling languages through **Viewpoint Specification Project**s. These contain **Viewpoint**s, which are sets of graphical functionalities that can be enabled/disabled all at once. One such functionality is the **Representation**, which specifies the type of visual entities. The two most common and useful Representations are **Diagram**s and **Tree**s. The former represents the entities of the metamodel with graphical elements (*nodes*), that can range form simple rectangles, rhombi and circles to complex user-provided images. These graphical elements can be contained within other elements (*containers*) or have them fixed at their boundary (*bordered nodes*). Nodes, containers and bordered nodes can be connected to each other by *edges*. The user can specify the look of the different elements and edges through what are called **Style**s. Furthermore, it is possible to hide/show diagram details through what are dubbed **Layer**s. Trees are much simpler as they represent the entities of the metamodel as a hierarchical list of elements. Two important features that are common to both Representations are: (*i*) we map metamodel entities and relations to graphical elements. When we instantiate such elements in the model (e.g. add a generator or connect two resistors) we can impose constraints to this mapping. For example, keeping in mind Figure 5.3, we might add the constraint that a `Wire` which `connects` two `Component`s that are both either `DCCurrentGenerator`s or `DCVoltageGenerator`s should not be allowed. This is one way to impose constraints on the models the user can create. (*ii*) we can define mechanisms for creating, deleting, editing, etc. diagram (i.e. model) elements through what are called **Tool**s.

Specifying the mapping from Ecore domain model to graphical elements can be done with the use of the **Acceleo Query Language** (**AQL**). [33] provides a comprehensive overview of the latter.

We can add more advanced custom functionality via user-developed Java code. For example, we can call user-written functions, called **Java Service**s, that perform some specific action not provided by Sirius.

It is possible to automatically arrange Diagram elements by specifying a layout algorithm. See the **Eclipse Layout Kernel** (**ELK**) [34] for the more advance algorithmic solutions available.

5.4 Acceleo

The last fundamental tool in our toolbox is **Acceleo**. It is once again non trivial. A brief and comprehensive overview is thus not possible. Furthermore, this tool does not require mastering some complex concepts as much as simply learning about its technicalities. You can consult [35] for its documentation. To get started with this tool follow tutorial [36]. An in-depth guide can be found at [37] instead.

5.4.1 Overview

Acceleo is capable of producing multiple files that contain automatically generated text (e.g. code) based on models that has been developed using Sirius tools.[71] It achieves this by providing the user with language constructs that allow the information in the models to be extracted. [37] illustrates the supported constructs. An important note is that when we want to generated code based on a model that conforms to one or more Ecore domain models, we must include these metamodels in the `.mtl` file using the

```
[module <module_name>('metamodel1_URI', 'metamodel2_URI', ...)]
```

syntax. This should be done only with disjoint metamodels (i.e. metamodels that do not reference each other). The files generated by Acceleo can contain **Code Block**s. These are places in the file where the user can add code without fear of being overwritten by the subsequent code generations. A feature that will prove to be very useful are the **Java Service Wrapper**s that allow us to add custom functionality by invoking Java code, analogously to what was done in Sirius.

5.4.2 Useful Tips when Developing Acceleo Projects

The reader is invited to consult the above cited websites to actually learn how to use Acceleo. The rest of this section is comprised of a couple of notes that might be useful and should be consulted once the reader has acquired some familiarity with this tool. (*i*) if you try to run code generation as a *Java Application* the way it is described at [38] you might encounter some erroneous behavior. This is especially true if you have multiple metamodels that reference each other. Trying to debug these problems might take away countless hours, without much progress. Try running code generation as an *Acceleo Plug-in* instead [39], this might save

[71]To be precise, Acceleo only requires a model that conforms to an Ecore domain model. How this model was obtained (e.g. with Sirius) does not matter.

you a lot of headaches. (*ii*) during the development of your Acceleo project you are likely to test it by running as an Acceleo Plug-in (or Java Application). Later on you are likely to want to develop a plug-in that generates code for any model represented by files with a specific suffix, as described at [40][41].[72] If you now test code generation through this new method you might find out that a previously working project is not performing as expected anymore. One of the most probable culprits for this is the fact that these two methods register *Universal Resource Identifier*s (*URI*s) in different ways. You can solve this problem by rewriting your project to account for the new way of registering URIs. An easier alternative exists and is advisable. It consists in modifying the **Acceleo UI Launcher Project** to register URIs the same way it is done when running as an Acceleo Plug-in. To do so, open the Acceleo UI Launcher Project's `src` folder in the *Model Explorer view* and look for the `*.popupMenus` *package*. Expand it, inside you should find a `.java` file. Modify this file as shown in Code 5.1. As you can see, we need to find the `run()` method and look for the statement where the model URI is generated. We need to eliminate this statement (or better just comment it out) and add a new one that generates the model URI as if running as an Acceleo Plug-in. This new statement is reported below.

```
...
public void run(IAction action) {
    ...
    //comment out the following line:
    //URI modelURI = URI.createPlatformResourceURI(model.getFullPath().
toString(), true);

    //add the following line instead:
    URI modelURI = URI.createFileURI(model.getLocation().toString());
    ...
}
...
```

Code 5.1: Modification to register Acceleo UI Launcher Project's URIs the same way as when running as an Acceleo Plug-in.

[72]Plug-in's are a standard Eclipse mechanism for generating, importing and exporting code that provides the IDE with functionality. We will discuss a little about Eclipse plug-in's in the last chapter.

Chapter 6

evigen

This chapter presents the project, dubbed **`evigen`** (enhanced verification IP generator), that was carried out during my six months or so of internship at Infineon. As such, you will find below quite a few technicalities and implementation tricks that were useful during development. The discussion that follows is centered around a first iteration of this project. One that aims at developing a general framework for writing and expanding tools for automating testbench generation based on EMF, Sirius and Acceleo.

6.1 `vipgen`

As we have seen in Chapter 4, UVM provides some standard architectures and guidelines for developing testbenches. For example, interface universal verification components (UVCs) typically follow the architecture summarized by Figure 4.1.

Given the tools we have introduced in the previous chapter, we might have the idea to abstract out some of these architectures with metamodels. The latter can then be used to generate code automatically. What we would like to ultimately achieve is giving users the possibility to develop, with a graphical tool, models that represent some UVM component. The user should be able to develop these models, that conform to the rules imposed by UVM, according to their needs. As an example, a component that the user might want to build is an UVC, which implies that we need to develop a metamodel for UVCs. Once a model has been developed, the users should be able to automatically generate as much as possible of the actual SystemVerilog code used in the testbench to implement what the model represents, e.g. an UVC.

6.1.1 Automated UVM Testbench Generation Framework

The above discussion implies that we need to perform certain tasks which are schematized in Figure 6.1. There you see that what we need to do is:

- Develop one or more metamodels that capture the desired architectures (e.g. UVCs).
- Develop a graphical tool, based on the above metamodels, that allows the user to define models.
- Develop a code generator, based on the same metamodels, that generates SystemVerilog code based on the model defined at the previous point.

The user can then:

- Build models (e.g. UVC models) using the graphical tool.
- Use the code generator to translate the above models into SystemVerilog code.

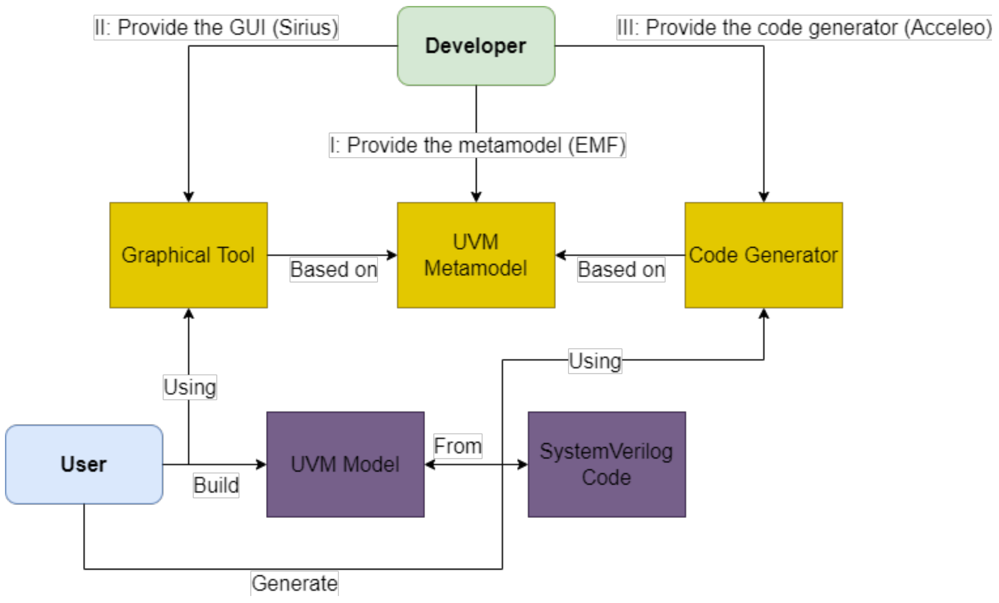

Figure 6.1: Schema of the Automated Code Generation Framework.

6.1.2 Interfacing with an Existing Testbench Generator

In this first iteration of `evigen` we have not dealt with defining metamodels for standard UVM architectures. This is due to the fact that Infineon already has an internal tool for UVM testbench generation dubbed `vipgen` (verification IP generator). This tool allows the user to specify the characteristics of a testbench component, whose SystemVerilog code will be generated, via an `.json` file.[73] It works in three phases:

- (*i*) Component generation: we invoke the tool and indicate that we want to create a new component, which represents a standard element of a testbench. The tool will then generate an empty project with an `.json` file.
- (*ii*) Component specification: we compile the `.json` file with the desired characteristics of the component.
- (*iii*) Component update: we can now invoke `vipgen` to generate the SystemVerilog code associated with the component specified via the `.json` file. We can change this file as many times as we want and call `vipgen` to update the generated code accordingly.

A more detailed description of `vipgen` will not be provided in this work. The three phases is all we need to understand the rest of this dissertation.

The focus of this first iteration of `evigen` has been to develop a more general framework for testbench generation, one that is capable of interfacing with other automated code generation tools, in particular with `vipgen`. This framework has been summarized in Figure 6.2. There you can see that we are no longer interested in defining UVM metamodels. Rather, we need to develop metamodels that capture the various components `vipgen` supports. The user interface that we need to develop will allow the specification of these components, which are a proxy for UVM components. The code generator will not produce SystemVerilog code directly. It will instead generate the appropriate `.json` file that will then be used by `vipgen` to produce SystemVerilog code. From user's perspective the workflow is unchanged. Models must be built using a graphical tool. These are then translated into SystemVerilog code via a code generator. The user is not aware of the existence of `vipgen` as the code generator takes the responsibility of managing it. This new framework requires us to perform some additional tasks:

- Interface with the Operating System to invoke `vipgen`.
- Read/write from/to files to fill in the `.json` file and check the generated `.json`/SystemVerilog code.

[73]json stands for JavaScript Object Notation. It is a data interchange format used to store and exchange arbitrary data.

- Manage the interface between the graphical tool and `vipgen`.

The details of the different tasks we need to perform in order to develop this framework will be discussed in the rest of this chapter.

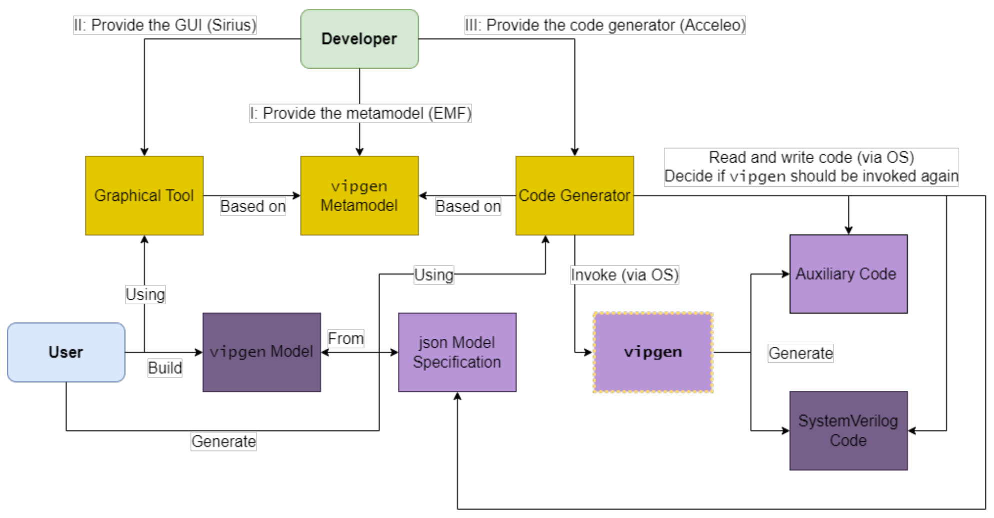

Figure 6.2: Schema of the `vipgen`-based Automated Code Generation Framework.

The are a few advantage of using `evigen` over simply relying on `vipgen`. The use of a Graphical User Interface (GUI) over a text format allows for faster and less error prone development of the testbench components. `evigen` is also easier to learn and the models that you develop with it are easier to maintain and update. The one drawback of `evigen` is that very complex components might clutter the GUI too much, a text format is thus more appropriate for those instances. It is however seldom the case, if ever, components that complex are needed.

Before moving on, let's briefly mention what are the plans for the future iterations of `evigen`. What we would like to do is expand the `vipgen` metamodel to support new functionality. In other words, design new metamodels, code generators and graphical tools that are super-sets of the first iteration `vipgen` metamodel and tools. The user will then be able to model UVM components that are more complex than the ones supported by `vipgen`. The functionality supported by the latter will be managed through `.json` files as usual. What is not supported will be managed by the new code generator that will read the SystemVerilog code written by `vipgen` and modify it accordingly. The technicalities of how this could be done should not be too complicated thanks to the framework that we have developed in this first iteration.

6.2 Interfacing with the Operating System

What follow in this section are some useful ways of interacting with the Operating System. As discussed previously, in order to use `vipgen` and have more control over the generated code, we need a way of performing some actions that involve the OS. Since the project was developed for *Linux*, the presented material refers to this Operating System.

6.2.1 Command Execution

The first fundamental task that we must be able to do is invoking shell commands from within the Java code we develop. This can be done as described in Code 6.1. The method presented below will execute `command1` followed by `command2` as if they were called from a shell with working directory `directoryPath`. It is easy to generalize this method to execute more than two commands. The function will report to `stdout` the shell output, if any, followed by a message that indicates the executed commands and the resource path where the commands were run.

```java
...
public static void executeCommands(String command1, String command2,
String directoryPath) {

    //add 'cmd + " && " +' to execute another command
    List<String> command = java.util.Arrays.asList("bash", "-i", "-c",
                                    command1 + " && " +
                                    command2 + " && " +
                                    "exit");

    //get the resource path where the commands should be executed
    //create directory is such path does not exist
    File componentParentDirectory = new File(directoryPath);
    if (!componentParentDirectory.exists())
        componentParentDirectory.mkdir();

    ProcessBuilder pb = new ProcessBuilder(command);
    pb.directory(componentParentDirectory);
    pb.redirectErrorStream(true);
    Process p = null;
    try {
        p = pb.start();

        BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));

        String line;
        while ((line = reader.readLine()) != null) {
            //report shell output
            System.out.println("[Shell Output]  " + line);
        }

        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    //wait for the process to finish
    try { p.waitFor(); }
    catch (InterruptedException e) {
        e.printStackTrace();
    }

    //report the command and the resource path
    System.out.println("Executed command: \"" + String.join(" ", command) + "\", in directory: \"" + componentParentDirectory.getName() + "\" (" + directoryPath + ")");
}
...
```

Code 6.1: Function for executing shell commands from within a Java program.

6.2.2 Resource Management

The second fundamental task we must be able to do is managing OS resources. In particular, we must be able to: (*i*) detect whether directories/files exist; (*ii*) select directories/files; (*iii*) create directories/files. Detecting the existence of files and directories is pretty straight forward. See Code 6.2 for the Java functions.

```java
...
//checks if the file already exists
public static boolean fileExists(String fileName, String directoryPath) {

    //Linux uses '/' to delimit files and folders
    //Windows uses '\' for the same purpose
```

```
        String pathDelimiter = "/";
        String completeFilePath = directoryPath + pathDelimiter + fileName;
        File file = new File(completeFilePath);

        return file.exists();
    }

    //checks if the directory already exists
    public static boolean directoryExists(String directoryPath) {

        File directory = new File(directoryPath);

        return directory.exists();
    }
    ...
```

Code 6.2: Functions for checking file/directory existence from within a Java program.

Being able to select directories and files is crucial for allowing the user to generate code in specific folders, as well as importing models that were developed at a different time. The functions that perform these operations are once again fairly simple. Code 6.3 reports them. You can see that the function that selects a file allows the user to specify two file suffixes (e.g. `".java"` and `".json"`). Only files that end with those suffixes can be selected. It is easy to modify this function to include a different number of suffixes.

```
    ...
    //lets the user specify a file with the suf1 or suf2 suffixes
    public String selectFile(String suf1, String suf2) {

        FileDialog dialog = new FileDialog(new Shell(), SWT.OPEN);
        dialog.setFilterExtensions(new String [] {suf1, suf2});
        dialog.setFilterPath(System.getProperty("user.dir"));
        return dialog.open();
    }

    //lets the user specify a directory
    public String selectDirectory() {

        DirectoryDialog dialog = new DirectoryDialog(new Shell(), SWT.OPEN);
        dialog.setFilterPath(System.getProperty("user.dir"));
        return dialog.open();
    }
    ...
```

Code 6.3: Functions for selecting file/directory from within a Java program.

Generating new files and directories is also pretty simple. See Code 6.4. The functions return `true` if the generation was successful, `false` otherwise.

```
...
//generate new file
public static boolean createFile(String fileName, String directoryPath) {

    String pathDelimiter = "/";
    String completeFilePath = directoryPath + pathDelimiter + fileName;
    File file = new File(completeFilePath);

    return file.createNewFile();
}

//generate new directory
public static boolean createDirectory(String directoryPath) {

    File directory = new File(directoryPath);

    return directory.mkdir();
}
...
```

Code 6.4: Functions for generating file/directory from within a Java program.

6.2.3 Reading and Writing to/from Files

The last fundamental task we must be able to perform is reading and writing to/from files. Unlike in the previous cases, developing such functions is quite a bit more complex. At the same time, there are multiple strategies for reading and writing to/from files. Different approaches, based on the specific needs, can yield more or less complex implementations. For example, we might scan the entire file and implement a parser that extracts the necessary information. Implementing a parser, however, can be very time consuming and thus should be done only if necessary. We might also use *Java Standard Library* functions that helps us perform this task.

During this first iteration of `evigen` we found it necessary to implement the following: (*i*) a function that looks for the first occurrence of a specific keyword in a file, and reports a string that begins with that keyword and goes to the end of the line. This function is used to parse the `.json` file generated by `vipgen` in order to extract some information. (*ii*) a function for copying the contents of a file to another. This function is used to copy the contents of the `.json` file generated by the Acceleo code generator to the `.json` file that `vipgen` uses. See Code 6.5 for the first function and Code 6.6 for the second one.

```java
...
public static String extractKey(String keyword, String fileName, String directoryPath) {

    String pathDelimiter = "/";
    String completeFilePath = directoryPath + pathDelimiter + fileName;
    File file = new File(completeFilePath);

    String line = "";
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        //while the file end has not been reached
        while ((line = reader.readLine()) != null) {
            //first occurrence of the keyword
            if (line.indexOf(keyword) >= 0) break;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    if (line == null) {
        //the end of the file was reached without finding the keyword
        System.err.println("Unable to find the " + keyword + " keyword");
    }
    else
        line = line.substring(line.indexOf(keyword), line.length());

    //report the results
    System.out.println("Analyzed file: \"" + file.getName() + "\" (" + completeFilePath + ") for the " + keyword + " keyword");
    System.out.println("Reporting: " + line);
    return line;
}
...
```

Code 6.5: Java function for extracting the first line that begins with a specific keyword from a file.

```java
...
public static void copySrcToDst(String srcName, String srcPath, String
dstName, String dstPath) {

    String pathDelimiter = "/";
    String srcCompletePath = srcPath + pathDelimiter + srcName;
    String dstCompletePath = dstPath + pathDelimiter + dstName;
    File src = new File(srcCompletePath);
    File dst = new File(dstCompletePath);

    try (BufferedReader reader = new BufferedReader(new FileReader(src));
         BufferedWriter writer = new BufferedWriter(new FileWriter(dst)))
{
        String line = "";
        //while the file end has not been reached
        while((line = reader.readLine()) != null) {
            //copy the src line to dst and add a new line character
            writer.write(line);
            writer.newLine();
        }
    } catch(IOException e) {
        e.printStackTrace();
    }

    //report the operation
    System.out.println("Filled in \"" + dst.getName() + "\" (" +
dstCompletePath + ") with \"" + src.getName() + "\" (" + srcCompletePath +
 ")");
}
...
```

Code 6.6: Java function for copying the contents of one file to another.

6.3 Customizing the Sirius Layout Algorithm

Sirius provides standard tools for specifying the layout of a Diagram. These tools are however somewhat limited and do not allow the developer to fully customize the user experience and UI. In order to have full control over the graphical elements of a Diagram, we must combine working with Sirius standard tools and developing Java code that implements custom functionality. Developing a fully satisfactory graphical tool can be a challenging task. This is due to both poor documentation and online support, as well as complexity and amount of work required. The development of the graphical tool is by far the most time consuming part of the project. This section will present a general guide for developing custom graphical functionality.

6.3.1 Notification Listener

The first step is understanding a little bit how Sirius works under the hood and how we can inject custom behaviour. Each action that the user can perform with the graphical tool can be intercepted. It does not matter if the action involves semantic elements (e.g. the Agents of an UVC) or their graphical representation (e.g. the position in the Diagram of the rectangle that indicates an Agent). We can detect both cases and provide custom actions that overwrite the default Sirius behaviour. To do so, we need to implement a **SessionManagerListener** [42] and a **ModelChangeTrigger** [43]. Let's walk you through how to do this. Start a new **Plug-in Project** in Eclipse (assuming you want to export the developed tools as Eclipse plug-ins)[74]. Suppose to name it `my.layout`. Now create a new Java class, suppose to name it `LayoutListener.java`, and fill it in with Code 6.7.[75] Fix all the plug-in dependencies. Eclipse should automatically present the option to do so if you hover the mouse over the icon indicating an error.

```java
package my.layout;

import org.eclipse.sirius.business.api.session.Session;
import org.eclipse.sirius.business.api.session.SessionManagerListener;
import org.eclipse.sirius.viewpoint.description.Viewpoint;

public class LayoutListener implements SessionManagerListener {

    public LayoutListener() {}

    @Override
```

[74]Click on "File" (top left), "New", "Other" and type "Plug-in Project", select the project type that appears and follow the wizard instructions (filling in only the project name suffices).

[75]Right click on the `my.layout` project, "New", "Class" and fill in the name.

```
    public void notifyAddSession(Session newSession) {
        newSession.getEventBroker().addLocalTrigger(LayoutInitializer.
    IS_GMF_NODE_ATTACHMENT,
        new LayoutInitializer(newSession.getTransactionalEditingDomain()));
    }

    @Override
    public void notifyRemoveSession(Session removedSession) {}

    @Override
    public void viewpointSelected(Viewpoint selectedSirius) {}

    @Override
    public void viewpointDeselected(Viewpoint deselectedSirius) {}

    @Override
    public void notify(Session updated, int notification) {}
}
```

Code 6.7: Implementation of a SessionManagerListener.

Next, we need to create another class, say `LayoutManipulator.java`, and fill it in with Code 6.8. Once again fix any dependency problems. Last, open the `MANIFEST.MF` file that you can find in the `META-INF` folder of the project and go to the "Extensions" tab.[76] There, click on the "Add..." button and type "org.eclipse.sirius.sessionManagerListener" in the search bar. If you do not see any results with this name being displayed, unselect "Show only extension points from the required plug-ins". Select the element that appears and click "Finish". Now right click on the newly added extension point named
"org.eclipse.sirius.sessionManagerListener" and select "New", "listener". Click on the newly added listener, you should see a new "Browse..." button appear on the right in the Editor. Click on that button and type the name of the second class we created (i.e. `LayoutManipulator`), select it among the elements that appear and click "OK".

```
package my.layout;

import java.util.Collection;

import org.eclipse.emf.common.command.Command;
```

[76]Tabs can be found at the bottom of the **Editor**, where the latter is the graphical element that popped up when you opened `MANIFEST.MF`.

```java
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.transaction.NotificationFilter;
import org.eclipse.emf.transaction.RecordingCommand;
import org.eclipse.emf.transaction.TransactionalEditingDomain;
import org.eclipse.sirius.business.api.session.ModelChangeTrigger;
import org.eclipse.sirius.ext.base.Option;
import org.eclipse.sirius.ext.base.Options;

public class LayoutManipulator implements ModelChangeTrigger {

    private final TransactionalEditingDomain domain;

    public LayoutManipulator(TransactionalEditingDomain domain) {
        super();
        this.domain = domain;
    }

    private static boolean notificationOfInterest(Notification notification) {
        ...
    }

    private static void performAction(Notification notification) {
        ...
    }

    public static final NotificationFilter IS_GMF_NODE_ATTACHMENT = new
    NotificationFilter.Custom() {
        @Override
        public boolean matches(Notification notification) {
            if (notificationOfInterest(notification))
                return true;
            return false;
        }
    };

    @Override
    public Option<Command> localChangesAboutToCommit(Collection<Notification>
    notifications) {
        Command result = new RecordingCommand(domain) {
            @Override
            protected void doExecute() {
                for (Notification notification : notifications)
                    performAction(notification);
            }
48         };
49         return Options.newSome(result);
50     }
51 
52     @Override
53     public int priority() {return Integer.MAX_VALUE;}
54 }
```

Code 6.8: Implementation of custom graphical functionality.

What we have done up to now is setting up a mechanism that is able to detect changes in our models, both semantic and graphical. To reiterate, semantic changes involve alterations in the underlying model. For example, the number, types and characteristics of UVC Agents. Or, thinking back at circuit models, the number, type, rating and interconnections of the different components of a circuit. Graphical changes, on the other hand, do not affect the underlying model we want to construct, only the way it is represented. For example, where and how big are the rectangles that represent UVC Agents, or the length of the wires between two circuit components. The model changes are detected as **Notification**s [44]. These can be filtered by implementing the `notificationOfInterest()` function of Code 6.8 in such a way that it returns `true` only if the Notification refers to an event that requires custom behaviour. `performAction()` of Code 6.8 allows us to implement this custom behaviour, based on the specific Notification. Deciding which Notification, or their sequence, corresponds to which action performed by the user is non trivial and depends on the specifics of the project. The same can be said about developing custom behaviour based on the detected Notifications. This guide's aim is that of introducing the topic and presenting a general way of approaching the problem. The actual implementation of the two aforementioned functions requires knowledge of `vipgen` and is beyond the scope of this essay. A good starting point to implement these functions, in a different project the reader might want to develop, is the Eclipse debugger. You might want to set a breakpoint at the very beginning of the `matches()` function of Code 6.8, e.g. at line 34. Then, use the debugger to reverse engineer which actions correspond to which Notification sequences. To do this, you can use their attributes that are visible with the debugger. This reverse engineering task could take a lot of trial and error and time. In the process, you will hopefully get an idea of how these functions can be implemented in your specific case.

6.3.2 Graphical Structure

Even thought the specific functions that implement Notification filtering and custom behaviour must be studied case by case, there is some underlying structure that Sirius Diagrams use. The semantic changes are usually reported via Notifications where the associated **Notifier**s are semantic elements.[77] Graphical changes, on the other hand, usually also involve Notifiers that represent graphical elements. Among these, two are of most importance: **NodeImpl**s and **BoundsImpl**s. The former relate to model entities that have an associated Diagram element (i.e. node, bordered node or container) and are the glue that holds together the semantics with the representation. The latter represent the layout of the Diagram elements that the user sees (i.e. the x, y coordinates and the *height* and *width*). In other words, each Diagram element has both an associated NodeImpl, which keeps track of the semantic element associated with it and Diagram structure, as well as a BoundsImpl, which keeps track of the layout. If you check the **Type** attribute of NodeImpls you might observe that NodeImpls of a certain Type (e.g. with Type *2002* representing a root container, *3008* representing a non-root container, *3012* representing a bordered node, etc.) have the **Element** attribute that is not *null*.[78] These are NodeImpls directly associated with a semantic element, which you can access using the `getElement()` and `getTarget()` functions, both of which return an **EObject**. For these Types of NodeImpls you can also access the layout of the graphical element using the `getLayoutConstraint()` function, which returns a BoundsImpl. Once you have the BoundsImpl, you can modify the layout characteristics by using the appropriate get and set functions (i.e. `getX()`, `setX()`, `getHeight()`, `setHeight()`, etc.). NodeImpls are usually composed in a tree-like structure to represent the hierarchical arrangement of the Diagram. This tree can be navigated using the `eContainer()` and `getPersistedChildren()` functions. The former returns the father NodeImpl of the current NodeImpl. The latter returns the curent NodeImpl's children NodeImpls. Note that not all NodeImpls have and associated semantic element and layout (e.g. NodeImpls of Type *7001*, *7002*, etc.), so you must navigate the tree carefully to determine the different semantic elements, their layout and hierarchy. The discussed structure has been summarized in the diagram that follows (Figure 6.3).

[77]A Notifier is nothing more than the element that triggered the Notification. You can check the Notifier of a Notification using the `getNotifier()` function.

[78]You can use the `getType()` and `getElement()` functions for checking.

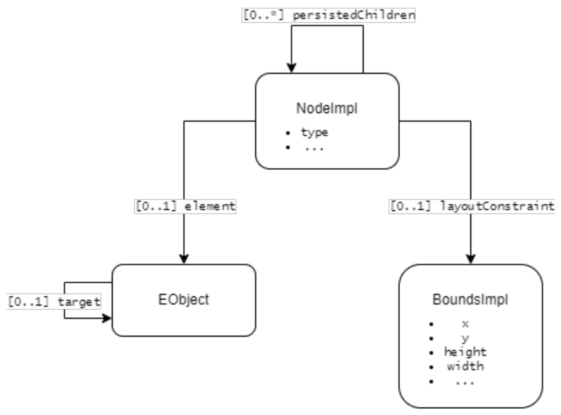

Figure 6.3: Sirius Diagram structure.

6.3.3 Modifying Semantics

An important characteristic of the interaction with Sirius is the type of element we want to manipulate. If we wish to modify the layout of Diagram elements, we can simply access and overwrite them. This can be done, as stated before, using the get and set functions for layout attributes and does not require any particular procedure. On the other hand, if the custom functionality that we implement modifies the semantics of the model, we must proceed in a specific way. We cannot simply access semantic elements and modify them. We must work within the *Transactions* framework. This requires us to implement the desired semantic modifications as Transactions. To do so, we need to wrap the commands we want to execute as shown in Code 6.9. There you can see the case in which we modify model semantics within the `performAction()` function of Code 6.8. In particular, we need the semantic element we want to modify (`semanticObject` in the code). To change it, we can add the necessary commands inside the `doExecute()` function of line 10, which must be wrapped by the line 7 to 20 code block.

```
1   ...
2       private static void performAction(Notification notification) {
3           ...
4           //the semantic object we want to modify
5           EObject semanticObject;
6           ...
7           TransactionalEditingDomain domain = TransactionUtil.getEditingDomain(
        semanticObject);
8           RecordingCommand command = new RecordingCommand(domain) {
9               @Override
10              protected void doExecute() {
```

```
                //write here the commands you want to execute
                // ...
            }
        };
        try {
            domain.getCommandStack().execute(command);
        } catch (Exception exception) {
            System.err.println(exception);
        }
    }
...
```

Code 6.9: Wrapper code necessary to modify model semantics.

6.4 Customizing the Eclipse User Interface

Once we have finished our project and are ready to deploy it, we might want to further customize the user experience by modifying the Eclipse environment. We can change the options that are visible in menus, the wizards that are used to create new projects and files, the type of information that is displayed to the user through Views, etc. This sections introduces this topic and some of the customizations implemented in `evigen`.

6.4.1 Custom Eclipse Perspective

If you have followed the Sirius and Acceleo tutorial linked in the previous chapter, you should be fairly familiar with the concept of Eclipse **Perspective**. To quickly recap, Perspectives are an Eclipse mechanism that allows the user to switch between different working environments. The environments are characterized by the menu options that are directly visible, the toolbar options, the default **Views**, etc. The concept of View is also an important one in the Eclipse IDE. It indicates a graphical tool that is conveying the user some information and can be interacted with. Examples of Views that are available in the default *Modeling* Perspective are the *Model Explorer*, *Outline*, *Properties* and *Problems* tabs. A more thorough overview of Perspectives and Views can be found at [45][46][47]. What follows next is a quick, tutorial style, introduction on how to create your own Perspective and how to customize it. Much of this is a condensation of the series of 24 tutorials that begin with [48].

The first step is creating an Eclipse Plug-in Project, opening its `MANIFEST.MF` file, and going to the "Extensions" tab, as described earlier (subsection 6.3.1). Now click the "Add..." button and type "org.eclipse.ui.perspectives" in the "Extension Point filter" search bar. If no results are displayed unselect "Show only extension points from the required plug-ins". You should be able to pick the "org.eclipse.ui.perspectives" extension point. Do so and click "Finish". You should see that a new "org.eclipse.ui.perspectives" extension point has been added with a sub element representing the Perspective (it can be recognized by the "(perspective)" tag). If that is not the case, right click on the newly added "org.eclipse.ui.perspectives" extension point and select "New", "perspective". By now clicking on the Perspective we have just added, you should be able to see new fields on the right side of the Editing window. Fill in these fields as follows:

- "id*" with an unique identifier for the new Perspective (e.g. `com.myOrganization.projectName.perspective`),
- "name*" with the new Perspective's name (e.g. `projectName Perspective`),
- "class*" with "org.eclipse.sirius.ui.tools.internal.perspectives.DesignerModelingPerspective" for Sirius based projects.

We can also click on "Brows..." next to the "icon" field to select an image to represent the custom Perspective. This concludes the creation of the custom Perspective. After saving, you should be able to see the new Perspective appear in the list of the available Perspectives ("Open Perspective" icon in the top right). We have thus created a new Perspective but we have not populated it with any custom features. Let's do this next.

The first element to add to the newly created Perspective is a wizard for creating a custom project. This project will contain the models we develop. I will explain the procedure for creating custom projects that contain models developed with Sirius, EMF and Acceleo. The principles can be applied to other types of custom projects. Start by clicking the "Add..." button and type "org.eclipse.ui.newWizards" in the "Extension Point filter" text box. Unselect "Show only extension points from the required plug-ins" if necessary, pick the "org.eclipse.ui.newWizards" extension point and click "Finish". Right click on the newly added "org.eclipse.ui.newWizards", "New", "category". Click on the newly added category, you should see a few field on the right in the Editing window. Among them should be present "id*" and "name*". Fill in the first text box with a unique id that will identify your custom projects wizards (e.g. `com.myOrganization.projectName.category.wizards`), and the second one with the category name (e.g. `projectName.wizards`). Once again right click on the "org.eclipse.ui.newWizards" extension point, "New", "wizard". Fill in the fields:

- "id*" with a unique identifier for the custom project wizard (e.g. `com.myOrganization.projectName.wizards.new.project`),
- "name*" with custom project's name (e.g. `projectName Project`),
- "class*" with "org.eclipse.sirius.ui.tools.internal.wizards.ModelingProjectWizard" for Sirius based projects,
- "category" with the previously defined category id (i.e. `com.myOrganization.projectName.category.wizards`) and
- "finalPerspective" with the custom Perspective id (i.e. `com.myOrganization.projectName.perspective`).

Also select "true" in the "project" drop down selection box, and "Browse..." for a custom project icon. The last step is right clicking the wizard we have just added to select "New", "description", and filling in the description that will pop up when the user hovers the mouse cursor over the custom project's icon. What we have done in these steps is adding a new wizard for creating custom projects. For example, projects that can store `vipgen` models.[79]

The second and last customization I will present here is adding the wizards for creating custom projects and models to menus for easy access. Doing so if fairly simple and similar to the procedure we followed before. Once again click the "Add..." button in the "Extensions" tab of the `MANIFEST.MF`. Look for the "org.eclipse.ui.perspectiveExtensions" and add it as an extension point. Set the "targetID*" field to the id we gave our custom Perspective (i.e. `com.myOrganization.projectName.perspective`). Now, for each element you want to be visible in the menus, right click of the perspectiveExtension we have just filled in and select "New", "newWizardShortcut". Use the "Browse..." button to find the ids of the wizards you want to add and fill in the "id*" fields of the newWizardShortcuts. Each newWizardShortcut should correspond to a different wizard. For example, in our case the id of the custom project wizard is `com.myOrganization.projectName.wizards.new.project`. At this point we

[79]In reality these projects can store any model developed with the EMF.

need to do something very similar to what we have just done. Start by clicking the "Add..." button once more and look for the "org.eclipse.ui.navigator.navigatorContent" extension point, add it. For each element that must be visible in the menus, right click the newly added "org.eclipse.ui.navigator.navigatorContent" extension point and select "New", "commonWizard". Now fill in the fields as follows:

- select "new" as "type*",
- fill in the "wizardId" the same way you did for the "org.eclipse.ui.perspectiveExtensions" extension point just now (e.g. `com.myOrganization.projectName.wizards.new.project` being the id of the new custom project wizard),
- the "menuGroupId" should be filled with a common identifier that will indicated the cluster of menu options associated with the new custom Perspective (e.g. `com.myOrganization.projectName.menu`).

After saving, you should see that icons for calling the wizards you have just added can now be found among menu options.

There are many other customizations that can be explored, I have but scratched the surface. For the sake of brevity and since the way these modifications are implemented is quite similar, I have presented just a few. You can find more information online. The last note on Perspectives is that if you right click a Perspective icon, a menu will pop up. If you select the "Customize..." option you will be prompted to a wizard that allows you to customize the Perspective further.

6.4.2 EMF Wizards

If you have tried to develop your own project, you might have noticed that when creating new models EMF takes care of the creation with standard wizards. We might not be completely happy with these and want to modify them. This can be done by directly altering the source code of the wizards that EMF has generated automatically. You can find the source code of the wizards in the packages of the "`.editor`" suffixed projects EMF has automatically generated when we compiled our metamodels with the "Generate, All" command.[80] Furthermore, if you check the `MANIFEST.MF` extension points, you will find the ids that uniquely identify these wizards, and that were used when adding these to menu options as described a few line above. You will also be able to see the source code by simply clicking on the "class*" field hyperlink. Now that you have access to the source

[80]What was just said is only clear if you have followed the tutorials on metamodeling.

code, you need to study how to best modify it according to your specific needs. It is important to remember that if the model semantics need to be modified during model creation (e.g. to set the model name according to some custom convention), we must use the Transactions framework introduced in subsection 6.3.3.

6.5 Exporting the Project as a Plug-in

Now that everything is done and we have a deployable project, we need to export it so that others can make use of it. You can learn how to do this by following tutorial [32]. Just remember to also include the Acceleo, custom layout and custom Eclipse UI projects that you have developed, not just the Sirius projects as is described in the tutorial. You can find out more about Plug-in development at [49]. Once you have finished this step, you should have a fully developed and distributable tool for automatic code generation based on the standard tools described in the previous chapter.

In the last few sections we have seen the steps and methods used to develop `evigen`. Granted, what was presented has been highly condensed and at points simplified. It should however still suffice to get you started in developing your own projects and tackling some of their most challenging parts.

Conclusions

In this work I have presented the basics of some of the various tools that are commonly used in the digital verification workflow. Specifically SystemVerilog (that required an introduction to Verilog) and UVM. These topics, as state already multiple times, have not been covered in depth. The reason for this choice is the amount of effort and time needed for such an extensive work. Still, enough basic notions have been given to get the reader started in the study of these topics and to have an overview.

For the second part of this dissertation, the focus was on the methodology and the tools that were used to carry out the project, and not as much on the project itself. This choice was once again deliberate. We think most of the academic value of this project is concentrated in these topics. Furthermore, we considered disclosing information about the internal tools developed by Infineon, which would enable us to present the project in more detail, not very useful and outside the scope of this work.

As we have stated in the last chapter, we plan on continuing to work on the project that was developed so far. In fact, it is necessary to do so. The reason being that we have found significant roadblocks, some of which have not yet been solved, when using the tools. This is especially (if not exclusively) true for Sirius. A notable amount of effort is needed when developing the GUI. Furthermore, the obtained tool is not very stables and needs a lot of patching to improve stability and fix the plethora of bugs. Still, with enough work anything can be fixed (we hope).

There are multiple metamodeling frameworks, not just EMF (with Acceleo and Sirius). For example, Spark Systems' Enterprise Architect is a commercial tool that can be used instead of the Open-Source framework we have adopted. You can find out more about Enterprise Architect at [50].

In the future we intend to explore potentially also different frameworks for developing a suitable code generation tool. All of these, however, build on the metamodeling topics we have presented here (developing metamodels, generating code by querying the models, and developing a GUI). As such, the hope is to have illustrated the power of the described methodology.

Bibliography

References

[1] Brock J LaMeres. *Quick Start Guide to Verilog*. Springer, 2019 (cit. on pp. 7, 36).

[3] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008 (cit. on p. 7).

[5] Jan M.. Rabaey, Anantha P Chandrakasan, and Borivoje Nikolić. *Digital integrated circuits: a design perspective*. Pearson Education, Incorporated., 2003 (cit. on p. 7).

[6] William J Dally and Tor M Aamodt. *Digital design using VHDL*. Cambridge University Press, 2016 (cit. on p. 7).

[7] Pong P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006 (cit. on p. 7).

[8] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988 (cit. on p. 8).

[9] Kim N King. *C programming: a modern approach*. WW Norton & company, 2008 (cit. on p. 8).

[12] S. Sutherland. *RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design*. CreateSpace Independent Publishing Platform, 2017 (cit. on pp. 46, 88).

[13] Chris Spear and Greg Tumbush. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2012 (cit. on pp. 46, 88).

[14] Ashok B Mehta. *System Verilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications*. Springer Nature, 2019 (cit. on pp. 75, 80, 88).

[15] Eduard Cerny et al. *SVA: the power of assertions in systemVerilog.* Springer, 2015 (cit. on pp. 80, 88).

[17] Hannibal Height. *A practical guide to adopting the universal verification methodology (UVM).* Lulu. com, 2010 (cit. on p. 92).

[19] Ken Arnold, James Gosling, and David Holmes. *The Java programming language.* Addison Wesley Professional, 2005 (cit. on p. 122).

[20] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. *Data structures and algorithms in Java.* John Wiley & Sons, 2014 (cit. on p. 122).

[21] Herbert Schildt and Danny Coward. *Java: the complete reference.* McGraw-Hill Education New York, 2014 (cit. on p. 122).

Sitography

References

[2] *Online tutorial on Verilog.* URL: https://www.chipverify.com/verilog/verilog-tutorial (cit. on p. 7).

[4] *Cadence website.* URL: https://www.cadence.com (cit. on pp. 7, 46, 88, 92).

[10] *Online SystemVerilog tutorial.* URL: https://www.chipverify.com/systemverilog/systemverilog-tutorial (cit. on p. 45).

[11] *ASIC world SystemVerilog tutorial.* URL: https://www.asic-world.com/systemverilog/tutorial.html (cit. on p. 45).

[16] *Accellera website.* URL: https://www.accellera.org/ (cit. on p. 92).

[18] *Online UVM tutorial.* URL: https://www.chipverify.com/uvm/uvm-tutorial (cit. on p. 92).

[22] *Eclipse Modeling Framework website.* URL: https://www.eclipse.org/modeling/emf/ (cit. on p. 122).

[23] *Eclipse website.* URL: https://www.eclipse.org/ (cit. on p. 122).

[24] *Online Sirius tutorials.* URL: https://wiki.eclipse.org/Sirius/Tutorials (cit. on p. 122).

[25] *Basic online tutorial on how to define a metamodel (domain model).* URL: https://wiki.eclipse.org/Sirius/Tutorials/DomainModelTutorial (cit. on p. 122).

[26] *Sirius documentation.* URL: https://www.eclipse.org/sirius/doc/ (cit. on p. 124).

[27] *Sirius wiki page.* URL: https://wiki.eclipse.org/Sirius (cit. on p. 124).

[28] *Sirius starter online tutorial.* URL: https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial (cit. on p. 124).

[29] *Sirius advanced online tutorial.* URL: https://wiki.eclipse.org/Sirius/Tutorials/AdvancedTutorial (cit. on p. 124).

[30] *Sirius compartments online tutorial.* URL: https://wiki.eclipse.org/Sirius/Tutorials/CompartmentsTutorial (cit. on p. 124).

[31] *Sirius properties view online tutorial.* URL: https://wiki.eclipse.org/Sirius/Tutorials/PropertiesViewTutorial (cit. on p. 124).

[32] *Sirius update site online tutorial.* URL: https://wiki.eclipse.org/Sirius/Tutorials/UpdateSiteTutorial (cit. on pp. 124, 149).

[33] *Acceleo Query Language (AQL) documentation.* URL: https://www.eclipse.org/acceleo/documentation/ (cit. on p. 125).

[34] *Eclipse Layout Kernel (ELK) algorithms.* URL: https://www.eclipse.org/elk/reference/algorithms.html (cit. on p. 125).

[35] *Acceleo wiki page.* URL: https://wiki.eclipse.org/Acceleo (cit. on p. 126).

[36] *Acceleo starter online tutorial.* URL: https://wiki.eclipse.org/Acceleo/Getting_Started (cit. on p. 126).

[37] *Acceleo online user guide.* URL: https://wiki.eclipse.org/Acceleo/User_Guide (cit. on p. 126).

[38] *Guide to run Acceleo code generation as a Java application.* URL: https://wiki.eclipse.org/Acceleo/Getting_Started#Run_as_Java_Application (cit. on p. 126).

[39] *Guide to run Acceleo code generation as an Acceleo plug-in.* URL: https://wiki.eclipse.org/Acceleo/Getting_Started#Run_as_Acceleo_Plug-in (cit. on p. 126).

[40] *Guide to building a plug-in for UI code generation launcher.* URL: https://wiki.eclipse.org/Acceleo/Getting_Started#Creating_a_UI_launcher (cit. on p. 127).

[41] *Guide to building a plug-in for UI code generation launcher.* URL: https://wiki.eclipse.org/Acceleo/User_Guide#Creating_An_Acceleo_UI_Project (cit. on p. 127).

[42] *SessionManagerListener interface repository.* URL: https://git.eclipse.org/r/plugins/gitiles/sirius/org.eclipse.sirius/+/5cf53378e4a084aefb90d9648d5db61d83881f58/plugins/org.eclipse.sirius/src/org/eclipse/sirius/business/api/session/SessionManagerListener.java (cit. on p. 139).

[43] *ModelChangeTrigger interface repository.* URL: https://git.eclipse.org/r/plugins/gitiles/sirius/org.eclipse.sirius/+/5cf53378e4a084aefb90d9648d5db61d83881f58/plugins/org.eclipse.sirius/src/org/eclipse/sirius/business/api/session/ModelChangeTrigger.java (cit. on p. 139).

[44] *Notification documentation.* URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/common/notify/Notification.html (cit. on p. 142).

[45] *Article on Eclipse Perspectives and Views.* URL: https://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html#Extend%5C%20an%5C%20Existing%5C%20Perspective (cit. on p. 145).

[46] *Article on Eclipse Views.* URL: https://www.eclipse.org/articles/viewArticle/ViewArticle2.html (cit. on p. 145).

[47] *Eclipse user interface guidelines.* URL: https://wiki.eclipse.org/User_Interface_Guidelines (cit. on p. 145).

[48] *Hidden Clause tutorial on writing Eclipse Perspectives and Views.* URL: https://cvalcarcel.wordpress.com/2009/07/08/writing-an-eclipse-plug-in-part-1-what-im-going-to-do/ (cit. on p. 145).

[49] *Eclipse Plug-in Development Environment (PDE) introcution.* URL: https://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html (cit. on p. 149).

[50] *Spark Systems website.* URL: https://sparxsystems.com/resources/index.html (cit. on p. 151).