POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Implementation of UVM-Based Framework for Enhancement of DFT DV Flows

Supervisors

Prof. Stefano QUER

Prof. Paolo BERNARDI

Candidate

Pietro MACORI

December 2022

Abstract

One of the main issues that companies have to face is the presence of legacy systems: tools, methodologies, and processes that are not updated over time and that can become a relevant bottleneck in company production.

The issue of legacy systems is present in the context of Apple's Design-For-Test team. Specifically, during the Design Verification (DV) phase, where the design of a chip is verified through software simulations, the engineers have to manually check that the behavior of some specific signals is the one expected. This step is time-consuming, repetitive, and error-prone. In this Master Thesis, the process of developing an automation tool able to verify the behavior of a chip's signals during the Design Verification is presented.

The tool consists of a library of checkers (i.e., SystemVerilog code able to verify a specific behavior of the signals) which are instantiated inside the test bench used to run the DV simulations using a UVM module. In this way, the instantiated checkers behave as a probe, reading in real-time the values assumed by the signals and internally verifying that their behavior is the expected one. At the end of the simulation, the engineer is informed of the outcome of each checker, possibly including messages to provide insights on the causes of an error.

The benefits provided by the tool lead to its extension to solve a different limitation of Apple's flow. Specifically, when the playback simulations are executed, the analog signals are not present. As a consequence, the engineers cannot perform any waveform review and a comparison with the software simulation of the chip's design is not possible. For these reasons, an improvement of the tool is performed, by allowing the presence of analog checkers and drivers to force and measure analog signals during the playback simulations.

The impact of the tool is not a trivial task to be performed. Indeed, since the previous approach was based on manual checks performed by the engineers, it is not easy to compute the time saved for each check. The metrics that have been considered are related to the impact of the tool over the simulation times and the

memory impact. In general, the tool increases the simulation time by about 9% and the memory usage by 2% which is a reasonable impact from the point of view of the tool user.

Acknowledgements

First of all, I would like to thank my thesis supervisors, Professors Stefano Quer, and Paolo Bernardi, who followed and supported me for the whole duration of my journey. In particular, their visit to Munich was very much appreciated, since it showed the high regard they had for me and my work.

I would also like to extend my gratitude to the Apple organization, which allowed me to develop my Master's Thesis during the internship. In particular, I must thank Alessandro, which helped me daily and gave me important insights related to the DFT field. Without his support, my work would have been much more complicated. Thanks also go to Dirk, my manager, who spent a lot of time and effort reading, checking, and providing feedback on the thesis. Eventually, I am very grateful to Andreas, the one who had the idea for the project I developed. His vision and his guidance have been fundamental for the success of the thesis.

I must express my very profound gratitude to my family, that supported me during my academic path. Their teachings have been fundamental to me and have allowed me to be a better person. They allowed me to completely focus on my career and they did not put any unnecessary pressure on me.

Finally, I have to thank Greta. She has been by my side during the last few years and she has been always present during the best and worst days, ready to share the happiness of a good day as well as cheer me up when things were not going in the right way. I'm sure that without her, this journey would not have been as beautiful and intense as it was.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In big companies, it is common that processes, methodologies, and technologies are not updated and require human intervention, even in cases where some automated tool could be used. The use of sub-optimal tools and processes is typically caused by a lack of time and resources while developing new projects. Consequently, the firm is locked into a legacy system that tends to become obsolete soon.
This issue affects every company and being able to recognize it and try to develop possible improvements as soon as possible can be an essential step in a firm's success.

In the context of Apple's Design For Testability (DFT) team and, more specifically, in the Design Verification (DV) flows, some processes are not done following the optimal approach and can be improved to simplify the engineers' tasks. During the DV phase, the design of a chip is verified through software simulations so that the possible problems in the design can be identified and fixed before reaching the silicon phase. This step is crucial since it allows the avoidance of issues that otherwise would occur during the silicon phase, resulting in a much more expensive and time-consuming flow. On the other hand, during the Design Verification simulations, checking that specific conditions are respected by the internal chip's signals can be very complex, and often the engineers have to write custom tests and perform manual checks on the waveforms after the simulation to be sure that the behavior of some specific signals is the one expected. As a consequence, the DV phase becomes time-consuming, repetitive, and error-prone

The goal of this Master's Thesis is to give an insight into the motivations, the challenges, and the technical aspects related to the development a Universal Verification Methodology (UVM) based framework dedicated to automating the checks on a chip's signals during a software simulation.
More specifically, the tool consists of two different parts, aiming to solve slightly different issues. The first one is a library containing some generic and reusable checkers that allow verifying that during a DV simulation the behavior of some specific signals is the expected one, avoiding manual time-consuming, and error-prone reviews of the waveform of the simulation.
The second part of the tool aims to solve a limitation of the playback simulations, which do not carry any information about the analog signals. As a consequence, the engineers cannot do any waveform review on the analog signals and they cannot compare the DV simulation with the playback one. The idea is to extend the checker library with analog checkers and analog drivers to allow the force and measure of analog signals when required.

It must be underlined that due to non-disclosure agreement (NDA) reasons, this document is a reduced version of an internal report owned by Apple. Indeed, in the following chapters, all the sensible data and information (such as details on the chip and IPs' names) have been removed, substituting them with generic names to allow the publication of the thesis.

Document organization The document is composed of six chapters. The current one is a general introduction to the Master's Thesis topic and goals. Chapter 2 introduces various topics needed as foundations for the subjects presented in the following chapters, such as an introduction to the DFT field, as well as a SystemVerilog and UVM overview. The third part describes the state of the art of the current DV flows and presents some of their limitations, explaining the need of having the tool previously described. Chapter 4 illustrates a detailed description of the goals, needs, requirements, and implementation of the UVM-based framework. This chapter is where the idea and the technical details of the project are presented, and it is the core of this Master Thesis.
The fifth chapter reports the results achieved by the framework. Specifically, a focus on the impacts that it has on the user's workflow and on the performance

of the simulations is made. Eventually, the last part contains the conclusions of the Master's Thesis project, and some possible improvements and future work are described.

Chapter 2

Background Concepts

To fully understand the concept behind the Master's Thesis project some fields must be analyzed and explained. In this chapter some side concepts such as DFT, UVM and Verilog are expanded.

2.1 Introduction to DFT Engineering

In the past, when the development of Integrated Circuits (ICs) was in its infancy and their complexity was remarkably lower than today, the correctness of a given IC was tested by simply providing stimuli to it and evaluating its behavior. In the last decades, the complexity of ICs has increased exponentially. Billions of transistors are packed on a single chip, and applying stimuli to test its correctness is simply unfeasible. To overcome this issue, a new engineering branch was born: Design For Testability or simply DFT.

Design For Testability is a computer engineering field consisting of IC design techniques, methodologies, and algorithms that add testability features to a chip's design [1]. The additional features allow the creation and use of manufacturing tests in order to check that the hardware product design does not contain manufacturing defects that could negatively impact the product's correct functioning. As stated previously, the main DFT objective is to add testability features to an IC. This means that new components must be added to the original chip's design to simplify its test. As a consequence, the work done by the DFT team in a company is

strongly bonded with others. The DFT engineers have to interact with the chip's designers, the physical design engineers, and the test engineers to add the needed components. In the following section, some of the main techniques are described.

2.1.1 DFT Techniques

JTAG JTAG is an industry standard for verifying designs and testing printed circuit boards after manufacture. At the turn of the eighties and nineties, the complexity of the chips was increasing year after year and functional testing was getting impractical. This led the IC companies to define a new protocol for functional testing [2]. To be able to use the protocol, the chips have to add a JTAG interface. An example of JTAG interface is reported in Figure 2.1.
The main features provided by a JTAG interface are:

- Boundary scan testing: JTAG provides access to many logic signals of a complex integrated circuit, including the device pins. The signals are represented in the boundary scan register (BSR) accessible via the Test Access Port (TAP). This permits testing as well as controlling the states of the signals for testing and debugging [3]
- Debugging: JTAG is used as the primary means of accessing sub-blocks of integrated circuits, making it an essential mechanism for debugging embedded systems which might not have any other debug-capable communications channel
- Storing firmware: JTAG allows device programmer hardware to transfer data into internal non-volatile device memory
- Daisy chaining connections: several devices to be connected to a single interface in a daisy-chain layout. The target devices must all share a common ground node and also be powered by the same supply voltage [4]

Scan logic Another technique applied in the context of DFT is the insertion in the design of the so-called scan logic. Testing a specific configuration of a chip may require a great effort and a long time since many flip-flops have to be loaded with predefined values in a given instant of time. To ease this task, the DFT engineers

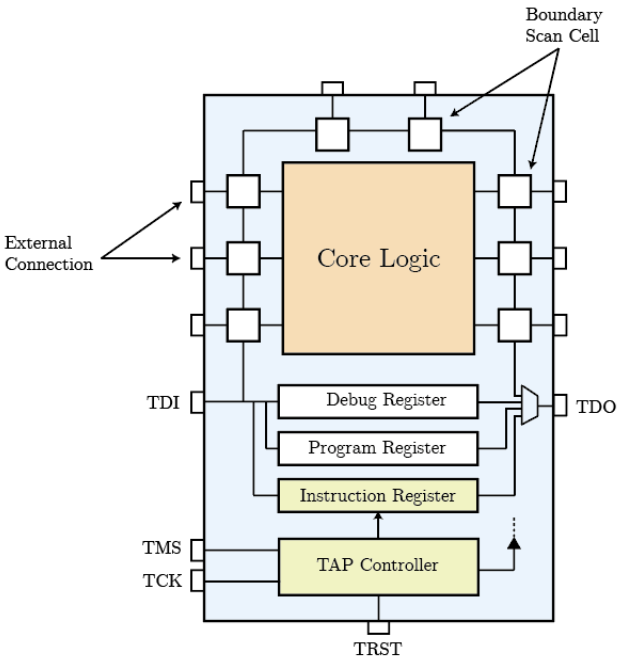

Figure 2.1: JTAG interface components

add the logic to connect all the various flip-flops in a chain, as a sort of shift register. Moreover, multiplexers are used to indicate to each of them if to shift the value to the next flip-flop or if to behave as if the added logic was not present. This allows to easily load the flip-flops as a shirt register and when all the values are set, to behave as they were supposed to do. In this way, a single clock toggle can allow the execution of one status of the chip and register the next one.

BIST The last presented DFT technique is the built-in self-test. This methodology is used to make faster, less-expensive integrated circuit manufacturing tests. This class of techniques is composed of controller logic which uses various algorithms to generate input patterns that are used to exercise the chip functionalities and verify its behavior [5].

2.1.2 Design Verification

In parallel to the presented techniques described above, Design For Testability carries out also the tasks related to the so-called Design Verification (DV). The

purpose of this process is to evaluate if the output of a design meets the required specifications. To make a simple example, the *chain* tests are written to verify that all connections have been correctly designed. The test itself simply reads and writes the same values from registers and assures that the values are the same. If they are different, it implies that there is a bug in the design and some improvements have to be carried out.

2.2 Testchip Structure

Since the thesis's project has been developed on the testchip and since some concepts related to its structure are described later, a description of the chip's content is made.

A testchip is a chip that contains the Intellectual Properties (IPs) of a chip that will reach the mass market. Its role is to test the various IPs placed on a product chip and find the best working parameters (e.g., voltages and frequencies). An IP can be seen as a pre-compiled module that provides some functionalities to the chip. They are divided into two categories according to their functional purpose: megacells and HardIps. Note that for each IP, there may be multiple instances in the same chip.
The various IP instances are grouped into clusters which typically contain instances of the same IP or IPs strictly related among themselves.
Various clusters compose a partition that can have multiple identical instances. In addition, a partition is typically further divided into sub-partitions needed to differentiate separate voltage domains.

2.3 Verilog and SystemVerilog Introduction

A Hardware Description Language (HDL) is a computer language used to precisely and formally describe the structure and behavior of electronic circuits. Besides automated analysis and simulation of an digital circuit, the information described by an HDL allows for the synthesis into a netlist (a low-level description of the connectivity of an electronic circuit). Currently, two of the most used HDL are

Verilog (standardized as IEEE 1364) and its extension, called SystemVerilog (IEEE 1800). This chapter includes a brief introduction and comparison between these two languages.

2.3.1 Verilog

Verilog is an HDL created in 1984 by the engineers of Gateway Design Automation whose purpose was to develop a language similar to C to easily allow the design and verification of electronic circuits at the register-transfer level (RTL)[6]. Verilog building blocks are called *modules*. These components can communicate with each other through a set of declared input and output ports and can be connected and arranged so that they create a design hierarchy. A Verilog module contains net and wire declarations, concurrent and sequential statement blocks, and instantiations of other modules.
In Verilog, there are two alternative ways to describe a digital circuit:

- **Structural description**: the circuit is described using basic modules representing logic gates
- **Behavioral description**: boolean operators are used to describe the functional behavior of the circuit

An example of the structural code of a half-adder (whose composition is shown in Figure 2.2) is reported in Listing 2.1.

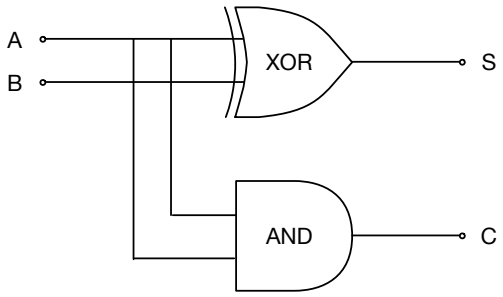

Figure 2.2: Half-Adder Schematic

```
1  // half_adder module
2  module half_adder (a,b,c,s);
3    // Define input and output pins
4    input  a,b;
5    output c,s;
6
7    // Structural description
8    // Note: first argument is the name of the output, the others are
       inputs
9    xor(s,a,b);
10   and(c,a,b);
11
12 endmodule
```

Listing 2.1: Verilog code of an half-adder circuit using the structural description

As the name suggests, the structural description's purpose is to represent the digital circuit by describing the logic gates of the circuit itself. As can be seen from the code above, in the structural description, the logic gates of the circuit can be represented using some corresponding Verilog primitives. For instance, the XOR gate having as input the signals a and b and as output the signal s can be described using the *xor(s,a,b)* command.
The structural description approach is practical for immediately providing an overview of the module's gates. However, for large and complex circuits, its use is not suggested since the readability of the code becomes very complex. Indeed, a module can be composed of thousand of logic gates, and understanding which is its functional behavior can become almost impossible. As a consequence, for big and complicated modules, a behavioral description is preferred.

The Verilog behavioral description aims to describe how the outputs are computed as functions of the inputs. To do so, this approach does not use any concept of a logic gate. Instead, boolean operators (such as &, |, $\sim$) are used to relate the module's input and output ports. The behavioral description of a half-adder is reported in Listing 2.2.

```
1  // half_adder module
2  module half_adder (a,b,c,s);
3    // Define input, output
4    input  a,b;
5    output c,s;
6    // Behavioral description
7    assign s = (~a&b)|(a&~b);
8    assign c = a&b;
9
10 endmodule
```

Listing 2.2: Verilog code of an half-adder circuit using the bahvioral description

For example, the output signal s is not described using the *xor* logic gate but directly relating the a and b signals using the boolean operators and applying the xor definition:

$$s = a \cdot \bar{b} + \bar{a} \cdot b$$

The behavioral approach is widely used for the description of complex modules since it increases the readability of the Verilog code and it allows an easy understanding of the module behavior. Moreover, an additional feature of this approach is that when the circuit is synthesized (the RTL is converted into a Netlist), it is open to many circuit optimizations.

2.3.2 SystemVerilog

SystemVerilog (SV) is a HDL created around the early 2000s. From an high level perspective, its functionalities can be divided into two sets:

- SystemVerilog for register-transfer level (RTL) design, which is an extension of traditional Verilog
- SystemVerilog for verification uses extensive object-oriented programming techniques (typically not synthesizable into a physical circuit)

For this reason SystemVerilog can be considered as a super-set of the traditional Verilog [7]. In the following paragraphs, some of the main features introduced by SV are presented and described.

Object-oriented programming model SV allows for the use of classes and objects similar to some famous object-oriented programming languages such as Java or C++. SystemVerilog supports inheritance, polymorphism, data encapsulation, and the use of the class constructor. The object-oriented model is typically used for verification purposes (e.g., UVM is entirely based on SV) since objects, interfaces and classes are not synthesizable into a physical structure.

Extended data types SystemVerilog introduced some new data type to increase the flexibility and hide some limitations of plain Verilog. Some examples are:

- strings
- logic (hides difference between reg and wire present in Verilog)
- multidimensional packed arrays
- structures
- dynamic arrays

Interface Is a SystemVerilog component that is able to encapsulate signals in the same block. It can be seen as an "adapter" between different components since it allows to easily connect heterogeneous elements among themselves. This powerful components behaves as a probe during the simulation and allows to have a direct access to the current value of a specified signal.

Assertions They are a language construct that allow to write constraints, checkers, and cover points for the design. Their use let the RTL designer express rules in the design specification in a SystemVerilog format that the simulation tools can understand. Moreover, errors are shown if a given assertion is not respected during a simulation, and the violation is immediately reported.

Coverage It is used to determine when the Device Under Test has been exposed to a good variety of stimuli that there is high confidence that the DUT is functioning correctly. Note that this differs from code coverage which instruments the design code to ensure that all lines of code in the design have been executed. Instead, functional coverage ensures that all desired corner and edge cases in the design space have been explored.

2.4 Universal Verification Methodology

The Universal Verification Methodology (UVM) is a standardized methodology for verifying the design of integrated circuits. This standard aims to develop modular, reusable, and scalable test bench structures. Indeed, UVM is composed of a hierarchy of class libraries defined using the SV language, where each component in the verification environment has a specific role. Moreover, the UVM standard introduces some features such as communications mechanisms among classes, timing phases for the synchronization, and generic utilities as a configuration database that simplify the task of writing the code.
It must be noticed that, unlike other verification methodologies, the Universal Verification Methodology maintains compatibility among the various simulation vendors (e.g., Cadence and Xilinx). For this reason, it quickly becomes an industry standard. The main Universal Verification Methodology components are briefly presented in the following section [8].

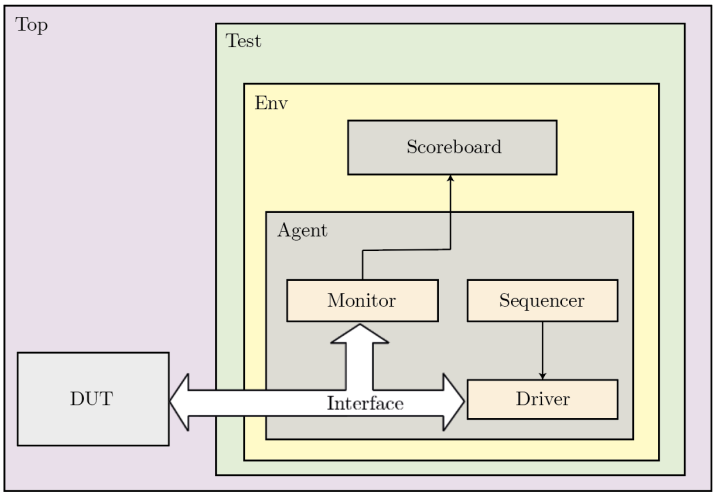

Figure 2.3: Components of a UVM test bench

2.4.1 UVM Components

Test-bench Top It is the only SystemVerilog *module* and it's where all other components are instantiated. Each simulation is executed on one UVM test bench.

Test This is SystemVerilog class which goal is to wrap all the other classes and define a single test-case. To execute a given test, the *run_test* method must be executed from the TB.

Environment It contains multiple, reusable verification components and defines their default configuration as required by the application.

Agent This component encapsulates the UVM driver, monitor and sequencer and connects them via a TLM interfaces. An agent can be:

- Active: instantiate all the three components and allows data to be driven to the DUT
- Passive: only the monitor is instantiated and no data can be driven to the DUT but it can only be read from it

Driver This is an active entity that has knowledge of how to drive signals to a particular interface of the design, forcing values into the DUT.

Sequencer Is the components that define the data that has to be driven into the DUT during the simulation. After the generation, the data is passed to the UVM driver via an interface.

Monitor it is responsible for capturing signal activity from the design interface and translating it into transaction-level data objects that can be sent to other components (e.g. the Scoreboard).

Scoreboard It is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction-level objects captured from the interfaces of a DUT via TLM Analysis Ports.

2.4.2 UVM Features

UVM Phases

UVM implements a synchronization mechanism called UVM Phases. Each component follows a predefined set of phases, and it cannot move on to the next

one until all other components have completed the current phase's execution [9]. As presented previously, SystemVerilog is an enriched version of Verilog which support OOP techniques and *classes* besides the traditional *modules*. Unlike from modules, which are static and are all created before the simulation, classes are structured entities that can be reused and deployed when required. This implies that without a synchronization mechanism there may be cases in which some components are exploiting not-initialized objects which can lead to wrong results or even non-created ones which will make the whole simulation fail. In the table below, the main phases are summarized.

Phase Category	**Phase Name**	**Description**
Build	build_phase	Used to build test bench components and create their instances
	connect_phase	Used to connect between different test bench components via TLM ports
	end_of_elaboration_phase	Used to display UVM topology and other functions required to be done after the connection
	start_of_simulation_phase	Used to set initial run-time configuration or display topology
Run	run_phase	Actual simulation that consumes time happens in this UVM phase and runs parallel to other UVM run-time phases
Clean	extract_phase	Used to extract and compute expected data from scoreboard
	check_phase	Used to perform scoreboard tasks that check for errors between expected and actual values from design
	report_phase	Used to display result from checkers, or summary of other test objectives
	final_phase	Typically used to do last minute operations before exiting the simulation

Table 2.1: Summary of UVM Phases

The various phases can be divided into three different groups: build, run, and clean. The run group is composed of the run phase which is the moment where the actual simulation is executed. It consumes simulation time and all UVM components execute their run phase in parallel. The build group is composed of the phases that must be executed before the run phase. During these phases, the SystemVerilog objects are created by calling the constructor and various classes can be connected among themselves. The clean set is formed by phases that are executed after the run phase. Their purpose is to check for possible errors, collect the results and report them.

UVM Configuration Database

UVM has an internal database table in which it is possible to store values under a given name and can be retrieved later by some other TB component. For instance, when the UVM TB module creates an interface and connects to it some signals, the UVM classes cannot access it directly. To overcome this issue, the module can store the interface in the configuration database labeling it with a given name. Then when a class needs to retrieve the interface handler, it simply has to retrieve it from the database using its name. The class needs just to receive a string from the TB module and then is able to get the previously generated interface.

Chapter 3

State of the Art

This chapter describes the state of the art in the context of Design Verification (DV) and Pattern Generation flows in Apple. Subsequently, two different limitations of the current approach are highlighted to give some insight into the needs that led to the development of this Master Thesis project.

The process is represented in Figure 3.1 and starts when a new version of an IP's design is delivered to the DFT team by the Design team. First, with the IP's design, a spreadsheet containing all the tests that must be executed on it is provided. Next, the engineers involved in the DV translate the spreadsheet containing the description of which register has to be written and read to verify the design correctness into a Tcl file. The Tcl syntax allows using human-readable commands to manipulate internal registers and signals (e.g., Read to read and compare a value of a register, Write to force a value into a register). The Tcl file is the starting point of both the DV and pattern generation flow.

Design Verification Flow A software verifies that a given IP is correctly integrated into the chip by simulating its behavior using its design. The simulation allows the engineers to find bugs or problems before starting silicon production. This saves time, effort, and money if many bugs are discovered before the beginning the chip production. Simulating the chip's design and building up the so-called Design Verification flow requires some steps that will be described as follow.
The flow starts from the Tcl file, which is manually written by an engineer starting

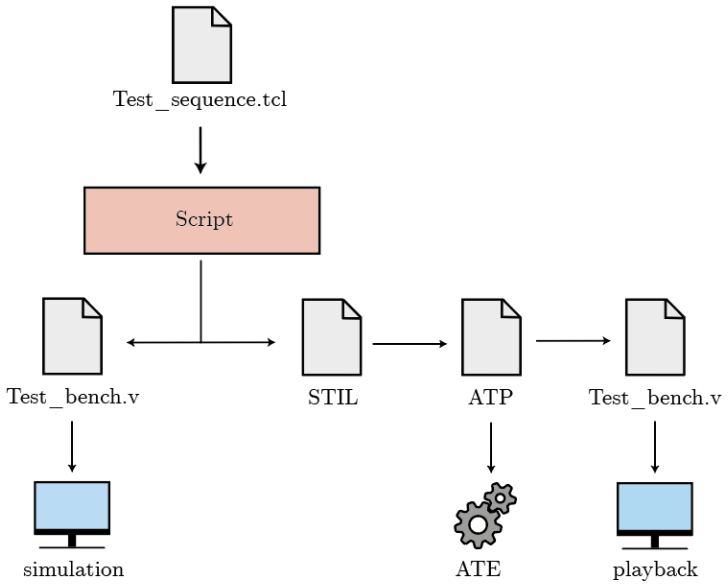

Figure 3.1: Design Verification and Patter Generation Flows

from the information provided by the spreadsheet. The Tcl is received as input by a script. This program is tasked with producing two different files. The one related to the DV flow is the Verilog test bench (TB) which will be executed during the simulation. In the context of digital design, a test bench is a Verilog component that includes the chip's design that must be verified. During the simulation, the TB module forces some stimuli into the Device Under Test (DUT) and monitors its registers and signals to understand if the behavior is equivalent to the one expected.

Pattern Generation Flow Even if the software simulations can detect many design bugs during the DV phase, the physical chip's functional behavior must also be verified. In fact, running tests on silicon can highlight bugs that the software simulation cannot catch. This is due to the fact the software simulation uses an ideal chip as DUT and cannot consider some physics limitations (e.g., leakage currents and delays). To verify the behavior of the silicon, some stimuli have to be applied to it, and to define them, a flow called Pattern Generation is defined.
As for the DV flow, the pattern generation starts from the Tcl file described above. The script uses it to generate a Standard Test Interface Language (STIL) file that defines the test vectors applied to the device under test. This file describes the test

pins, voltage specification, timing specification, and test pattern. An Automatic Test Pattern (ATP) file is produced from the information in the STIL file. This file contains the same information as the STIL, stored in a format that is usable by the Automated Test Equipment (ATE). The ATE is the machine that physically tests the silicon of a chip by applying currents and voltages to the various pins. The ATP file is then converted into Verilog to perform a software simulation. This second simulation verifies that no errors happened during the creation and execution of the patterns, such as a missing initialization pattern or misconfiguration of said patterns. An example of pattern misconfiguration could be the non-retention of status from a previously executed pattern. Furthermore, generating a Verilog TB from the ATP allows re-creating in software the simulation run on the actual chip by the ATE. The software simulation takes the name of **playback simulation**.

3.1 Limitations of the Design Verification Flow

The current method to perform Design Verification has some limitations. The first issue is related to the fact that the Tcl commands used to read, write and compare signal values are finite in number and applicable in discrete moments. As a consequence, the checks that can be done are limited. For example, assume an isolation signal needs to be deactivated during the power-up of a given IP. Currently, this is verified by adding two *Compare* commands inside the Tcl file, the first one checks that the signal is high while the second that the signal is low. If the design is correct and has no bugs, this approach could be enough. However, the same cannot be said if the design has some issues and the isolation signal flickers when it is supposed to stay low. If the *Compare* command is not placed at the right moment in time, this abnormal event would not be detected, and a user would not catch a design bug. Figure 3.2 shows a graphical representation of both the described cases.

Another DV flow limitation is the check on the relations among signals. Indeed, verifying that multiple signals toggles following a given pattern could be quite challenging using the approach based on the *Compare* command. Let us consider again as an example the power-up of a Megacell. Typically, this process involves three signals: two power signals (small and big) and an isolation signal. To correctly

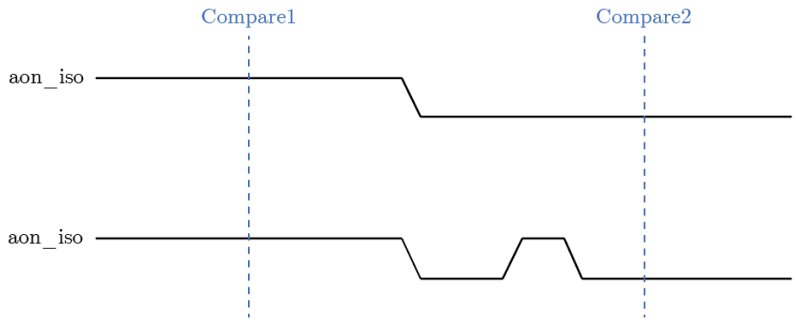

Figure 3.2: Verification of disable of isolation signal during power-up using two Compare commands

power up the IP, the signals need to toggle in a specific order:

- power small to high state
- power big to high state
- isolation to low state

Figure 3.3 graphically represents the described example. Verifying that the relationship is respected using the Tcl commands becomes very complex. A manual review of the waveform dumped from the simulation and visually checking that all the power-up IPs respect the requirements is tedious. Moreover, these tasks can take much time (especially for chips with dozens of IPs) and often becomes a bottleneck in the development process.

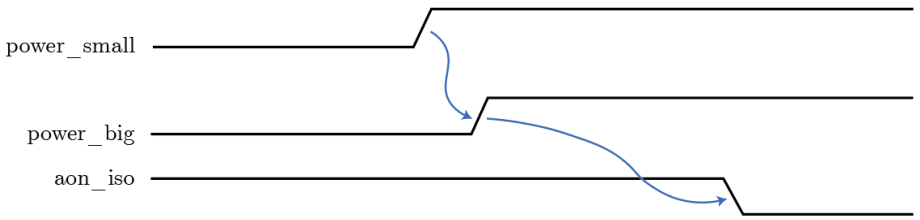

Figure 3.3: Isolation and power signals during power-up sequence

One more issue is that the writer of the checker must have a deep knowledge of architecture under test. For example, to write and check the value of a specific signal of the DUT, the test writer must know the internal registers names, the signals' path, and their relationship with the chip.

3.2 Limitations in Playback Simulations

As discussed previously, the playback simulation goal is to verify that no conversion error has happened during the generation of the ATP file. The behavior of the playback simulation is compared with the simulation executed in the DV flow to detect possible errors during the conversion of the files. The nature of the ATP file causes the problem in this context. Indeed, the ATP allows only four possible values to be forced into the chip: 0, 1, X, and Z. Having these four valid values implies that no analog values are allowed inside the ATP file. In fact, all analog operations are reported as comments in the ATP file and are not executed by the ATE. Consequently, the Verilog test bench generated to perform the playback simulation will not contain any information related to analog signals' operations (e.g., force and measure).
The impossibility of executing analog operations leads to some issues. First, the two Verilog test benches (the one generated by the DV flow and the one for the playback) are not comparable since they do not contain the same information on analog signal operations. Moreover, waveform reviews of the playback simulation are pointless since no information is reported for the analog signals of the chip.

Chapter 4

UVM-based Checker Library

After seeing the state of the art for DFT flows and their limitations, a possible solution for improving them is presented in the current chapter. Specifically, the idea, the technical choices, and implementations are discussed.

The purpose of the Master's Thesis project is to integrate into an existing DFT flows a UVM-based framework that automatically verifies the logical correctness of the signals of a chip during the DV simulations. The logic to verify that the signals of an IP are correctly behaving is contained inside components called **checkers**. These components are SV classes that implement the code to check signals' behavior and relationships. Each checker must be generic and reusable to be easily applied to different IPs simultaneously. Moreover, its code must be manually written once by a user and then can be reused without the need to modify it.
The set of created checkers defines a **checker library** from which the SystemVerilog code can be easily retrieved and applied to the various IPs of the DUT during the simulations.

The application of the predefined checkers of the library during a simulation must be supported using some SystemVerilog components. Specifically, a UVM-based module is instantiated inside the Verilog test bench generated from the Tcl. This module retrieves and applies the needed checkers of the library during the simulation and connects them with the IPs' internal signals via SystemVerilog interfaces. It is essential to notice that since the checkers provide an outcome on the status of

the simulation, the UVM-based module will often be named UVM TB. A high-level schematic of the elements composing the Verilog TB is shown in Figure 4.1.
Building the framework on top of the generated Verilog TB guarantees a solution that does not affect the behavior of the pre-existing simulation environment. Indeed, the stimuli applied to the DUT are the ones defined in the Tcl and are not modified by the UVM TB, which will behave as a passive component, probing the internal signals of the IPs without changing their behavior. Moreover, this solution supports different types of DUT, such as devices defined with RTL or Netlist.

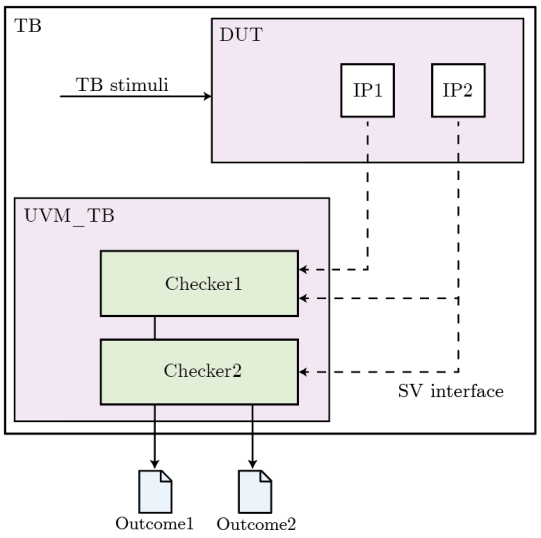

Figure 4.1: Verilog TB instantiating the DUT and the UVM-based TB

Creating a UVM test bench supporting the use of the checker library could be a demanding task for a user. Every simulation requires a dedicated UVM TB, typically composed of hundreds of lines of SystemVerilog code that would need to be written from scratch. This limitation has been solved by automatically generating the required UVM code through a few python scripts to make the framework more user-friendly and effortless. In this way, the framework allows the user to use an **automated checker library** which can be easily managed through a few configuration files and requires no SV code to be manually written. Furthermore, the scripts have been fully integrated into Apple's regression environment to make the framework almost transparent to the user.

Eventually, the framework is extended to support a new feature: the force and measure of analog signals during the playback simulations. Indeed, using the same infrastructure, it is possible to define checkers of the library that can verify the correctness of some analog signals of the DUT. Similarly, it is possible to create some components able to drive analog values inside the signals. These components would solve the limitation related to the lack of analog information in the context of the playback simulations.
It is crucial to notice that this last part of the project has not been fully developed and integrated into the framework. Indeed, a first working prototype has been implemented, but more refinement must be carried out.

In the following sections, a bottom-up description of the framework is presented, starting from the basic concept of the checker, going through describing the UVM TB module needed for the use of the checkers, and finally analyzing how the framework has been automatized. Furthermore, a focus on the prototype supporting the analog operations during the playback simulations is made.

4.1 Checkers

In the context of this project, the term "checker" indicates SV code that can verify the logical correctness of one or more signals during a software simulation. Typically, a checker is a function that receives as input an IP's signals (coming from an interface) and contains the code needed to check that their value is correct. Given the inputs and the logic, the checker returns the outcome of the verification as a boolean value.

Using SV for the definition of the checker logic provides the user writing the checker code with great flexibility and power. SystemVerilog's features include dynamic arrays and logic data types. Furthermore, OOP allows writing checkers that could not be developed with plain Verilog. For instance, using classes and objects allows the definition of class variables. These can be used as "memory" for the objects, which permits the definition of methods emulating a finite-state machine (FSM) behavior. This way, values received as input can be put into a relationship with values in the past, enhancing the complexity of the controls that

the checker can make. It this possible that in this document, the term FSM is used as a synonym of checker even if a checker may not implement a finite-state machine.

The logic of a checker cannot verify the correctness of the whole IP during a simulation but can check specific and relevant use cases. For example, an IP's power-up sequence is a critical process, and verifying the absence of design bugs is fundamental. Therefore, a checker can be defined to prove that the signals involved in the power-up are correctly behaving during the simulation.

During the project's development, it was noticed that the more a checker needs to be generic and reusable, the more its SV code becomes complex. For instance, creating a checker that supports a variable number of signals, each composed of a different number of bits, becomes very challenging. The need was to reduce the complexity of the checkers while keeping the possibility to cover any use case. For this reason, two possible checker types are supported: **specific** and **generic** checkers.
A specific checker supports a limited number of signals composed of multiple bits each. This type of checker is helpful when the checker must verify a well-defined number of signals. For instance, the power-up sequence of the megacells is often managed with one isolation and two power signals. This case could be easily managed using a specific checker
On the other hand, a generic checker can deal with any number of signals as inputs, but they all have to be composed by a single bit. This case is practical when a checker does not know a priori which is the number of signals that will have to be verified. An example is verifying that some signals are at default during a reset. In this case, each IP could be required to check a different number of signals, and a specific checker would not be able to deal with this use case. Using this distinction between specific and generic checkers, a user writing a checker can verify any signal's behavior without creating highly complex SystemVerilog code.

Since the code of the checker can be written once and then be reused multiple times, the idea is to collect various of them in a so-called checker library. In this way, the framework can access the library to retrieve the SV code that needs to be

applied to verify the correctness of some IPs' signals. The following paragraphs report a description of the checkers developed and integrated into the checker library.

4.1.1 Megacells Power-Up Sequence

As described in Section 3.1, the power-up of megacells in the test-chip is a process that involves three signals that have to toggle in a specific order. Verifying that the power and isolation signals behave correctly during the simulation is a challenging task to perform using Tcl commands. It is often carried out by visually inspecting the waveform of the simulation. This checker aims to verify that the sequence of toggle of the various signals is correct by implementing a state machine. At every clock cycle, the FSM is updated with the values of the three signals. At the end of the simulation, the final states indicate if the sequence of toggles has been recognized.

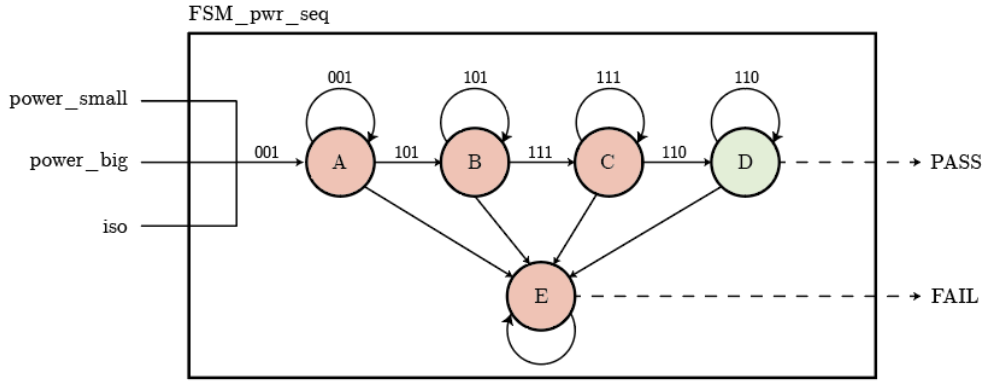

Figure 4.2: States of the FSM that verifies the correctness of the power-up sequence

As can be seen in Figure 4.2 and Figure 4.3, there is a one-to-one relationship between the possible combination of the signals' values and the FSM states. State A is the initial state, implying that the power signals are low while the isolation is active. When the power_small becomes high, the state B is detected; when the power_big is activated, the current state becomes C. Eventually, if the sequence

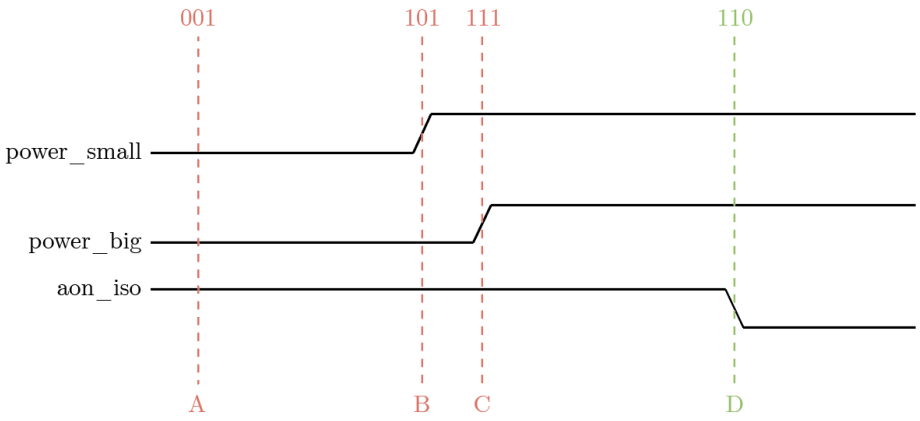

Figure 4.3: Signal relationship during power-up sequence and FSM states

is correct and the isolation signals are deactivated, the state D is found. If the sequence is incorrect, state E is detected, and the checker reports a negative outcome.
The FSM is implemented using SystemVerilog in the code reported in Listing 4.1. The various states of the machine are represented using local parameters, and they follow the nomenclature used previously. Additionally, the *current* and *following* variables store the values of the state before and after the update of the FSM. The update of the FSM is managed through a function called *update_FSM*, which receives the values of the three signals as input. The update of the FSM is implemented using some *if-else* statements. Eventually, the outcome of the FSM is computed. A positive result is provided if state D is detected. Else, a negative outcome is returned.

4.1.2 Default Values During JTAG Reset

One of the most time-consuming tasks in the DV context is verifying the default values during the JTAG reset. Indeed, at the beginning of the simulation, the JTAG reset is made to toggle one or two times. During these events, some signals must be at the default value and manually verifying that this condition is respected can be challenging.

```
// Power Sequence Checker
class FSM_pwr_seq;
  // Define output variable and current and next state
  reg out;
  reg [1:0] current, next;
  // Define the states of the FSM
  localparam [1:0] A = 2'b000, B = 2'b001, C = 2'b010,
                   D = 2'b011, E = 2'b100;

  // Update function
  function bit update_FSM(logic pwr_small, logic pwr_big, logic iso);
    case (current)
      A: if (~pwr_small && ~pwr_big && iso)       next = A;
         else if (pwr_small && ~pwr_big && iso)   next = B;
         else                                     next = E;
      B: if (pwr_small && ~pwr_big && iso)        next = B;
         else if (pwr_small && pwr_big && iso)    next = C;
         else                                     next = E;
      C: if (pwr_small && pwr_big && iso)         next = C;
         else if (pwr_small && pwr_big && ~iso)   next = D;
         else                                     next = E;
      D: if (pwr_small && pwr_big && ~iso)        next = D;
         else                                     next = E;
      E:                                          next = E;
    endcase

    // Update current status
    current = next;
    // If current state is D then positive outcome
    if (current == D)  out = 1;
    else out = 0;

    return out;
  endfunction
endclass
```

Listing 4.1: SV code of the power sequence checker

The SV class presented in Listing 4.2 reports the checker's code that verifies that a set of signals is at default during the reset.
The primary function is *update_FSM* which inputs are an array containing the values of the signals and an array containing the corresponding default value. After verifying that the arrays are not empty, a simple comparison is made on each couple. If a discrepancy is found, the error counter is increased. If at least one error has been detected, the function returns 0. Else, it returns 1.

4.1.3 Memory Addresses Usage During MBIST

Built-in self-test (BIST) is the standard approach to testing the memories of a chip. During the test execution, it must be sure that all memory addresses have been read and written at least once. If this does not happen, likely, that something did not work as expected. A checker has been developed to verify that all addresses are used during the memory BIST tests.
At every clock cycle, the *update_FSM* function receives the memory address used by the simulation, a clock enabler (called CEB) and a write enabler (WEB) as inputs. The combination of these two signals indicates which operation is being executed on the memory. The Table 4.1 summarized the combinations of the two signals.

CEB	WEB	Operation
0	0	Write
0	1	Read
1	0	Illegal State
1	1	Idle

Table 4.1: Possible combinations of CEB and WEB and relative operation

```
class FSM_default;
  // Define variables
  reg out;
  string msg;
  int err;

  // Update function
  function bit update_FSM(logic values_arr [], logic default_arr []);
    // If no signal received, throw error
    if (values_arr.size() <= 0 || default_arr.size() <= 0) begin
      msg = "No signal found";
      return 0;
    end

    msg = "Wrong default value: ";
    err = 0;    // Init count errors
    // Iterate over all elements of the two arrays
    for (int i = 0; i < values_arr.size(); i++) begin
      // Check if value equal to default
      if (values_arr[i] !== default_arr[i]) begin
        msg = {msg, values_arr[i].name, "; "}; // Error message
        err = err + 1; // Count number of errors
      end
    end

    // If error found then return FAIL
    if (err == 0) begin
      msg = "Ok";
      out = 1;
    // Else return PASS
    end else out = 0;

    return out;
  endfunction
endclass
```

Listing 4.2: SV code of the default value checker

The logic of the checker is reported in the code reported in Listing 4.3. First, the SV class is parameterized, indicating the number of bits composing the address. From this information, the dimension of the memory can be retrieved with simple computation. Then, two arrays can be defined with a size equal to one of the memory. In this way, the address used to access a memory cell can also be used to point to a specific element of the arrays. If each element of the arrays contains a counter, this can be increased every time a read or write is performed on the relative cell. At the end of the simulation, each array will store the number of times a given address has been read or written. These arrays are initialized to 0 when the class's constructor is executed.
In the *update_FSM* function, the memory address (received as an array of bits) is converted into an integer to access the relative cell of the read and write arrays. Then the CEB and WEB values are used to understand which operation has been executed. If it is a read or writes, the relative array is updated, increasing the value stored by the cell in the position indicated by the memory address. The whole simulation is aborted if the illegal state is detected and a fail message is reported. If the program identifies the idle state, then it executes no operation. Then, the number of elements of the arrays still with a value equal to zero is counted, and if at least one is found, an error is reported since it implies that a memory cell has not been read or written.

```
1  class FSM_address_space #(parameter ADDR_SIZE = 0) ;
2    // Compute memory dimension
3    localparam MEM_SIZE = 2 ** ADDR_SIZE;
4    // Define variables
5    string msg;
6    int mem_pos;
7    reg out;
8    // Arrays counting read and write of the various address
9    int read_count[MEM_SIZE], write_count[MEM_SIZE];
10   // Counters for errors
11   int check_read;
12   int check_write;
13
14   function new();
```

```
15      // Init the arrays to zero
16      for (int i = 0; i < MEM_SIZE; i++) begin
17        read_count[i] = 0;
18        write_count[i] = 0;
19      end
20    endfunction
21
22    function bit update_FSM(logic [ADDR_SIZE] addr, logic ceb, logic web);
23      mem_pos = addr; // Store the memory address as integer
24
25      // CEB = 1 and WEB = 0 -> illegal combinations
26      if (ceb == 1 && web == 0)
27        `uvm_fatal("FSM", "ILLEGAL STATUS: CEB = 1 AND WEB = 0 !!!")
28      // CEB = 0 and WEB = 1 -> READ
29      else if (ceb == 0 && web == 1) begin
30        read_count[mem_pos] = read_count[mem_pos] + 1;
31      // CEB = 0 and WEB = 0 -> WRITE
32      end else if (ceb == 0 && web == 0) begin
33        write_count[mem_pos] = write_count[mem_pos] + 1;
34      end
35
36      // Count how many address have not been read or written
37      check_read = 0;
38      check_write = 0;
39      for (int i = 0; i < MEM_SIZE; i ++) begin
40        if (read_count[i] == 0)
41          check_read = check_read + 1;
42        if (write_count[i] == 0)
43          check_write = check_write + 1;
44      end
45
46      // Define outcome and error messages
47      msg = "";
48      if (check_write != 0) begin
49        out = 0;
50        msg = "Not all cells are written; ";
51      end
```

```
52      if (check_read != 0) begin
53        out = 0;
54        msg = {msg, "Not all cell are read"};
55      end
56      if (check_read == 0 && check_write == 0) begin
57        out = 1;
58        msg = "Ok";
59      end
60      return out;
61    endfunction
62  endclass
```

Listing 4.3: SV code of the address space checker

4.1.4 Active Partition Instances

During a chip-level simulation, it must be ensured that the active partition instances are the intended ones. In the test-chip, there are two different cases. The first one is related to high-temperature operating life (HTOL) simulations which require that all partitions are executed in parallel during the simulation. In all other cases, one and only one partition can be running during a simulation, while all others must be deactivated. Due to the existence of these two use cases, two different checkers have been developed.

Check All Active Instances This checker verifies that all partition instances are active during the HTOL simulations. The *update_FSM* method receives as input three arrays, the first storing the enable signal of each partition instance, the second storing the WSI (Wrapper Serial Input) signals, and the last storing the various WSO (Wrapper Serial Output) signals. The enable signals indicate which partition instance is active while the WSI and WSO are, respectively, the input and output scan signals of the protocol IEEE 1500 protocol. If their behavior must be the same across the various partition instances, it implies that the operations executed in each partition are the same.
The SV class shown in Listing 4.4 has been developed to verify that all partition instances are active and executing the same scan operations.

```
class FSM_check_all_instances;
  // Define class variables
  string msg;
  int num_inst;
  bit active_correct = 1;
  bit wsi_correct = 1;
  bit wso_correct = 1;

  // Define constructor
  function new(int num_inst = 0);
    this.num_inst = num_inst;
  endfunction

  function bit update_FSM(logic enable_arr[], logic wsi_arr[], logic
    wso_arr[]);
    // Save wsi and wso of first instance
    logic first_wsi = wsi_arr[0];
    logic first_wso = wso_arr[0];

    for (int i = 1; i < num_inst; i++) begin
      // Check that each partition is not enabled
      if (enable_arr[i] != 1) begin
        active_correct = 0;
      end
      // Check that all WSI are equals
      if (first_wsi != wsi_arr[i]) begin
        wsi_correct = 0;
      end
      // Check that all WSO are equals
      if (first_wso != wso_arr[i]) begin
        wso_correct = 0;
      end
    end

    // Update the error/pass messages and return
    if (active_correct == 0) begin
```

```
36        msg = "One instance is not active";
37        return 0;
38      end else if (wsi_correct == 0) begin
39        msg = "One instance has wrong WSI";
40        return 0;
41      end else if (wso_correct == 0) begin
42        msg = "One instance has wrong WSO";
43        return 0;
44      end else begin
45        msg = "Passed";
46        return 1;
47      end
48
49    endfunction
50  endclass
```

Listing 4.4: SV of the checker that verifies that all partition instances are active

Check Single Active Instance All the chip-level simulations that are not HTOL simulations require a single and specific partition instance to be active during the simulation. All the others must be deactivated. The checker logic is quite similar to the one presented previously. The only difference that the WSI and WSO are not taken into account. The *update_FSM* function receives as input an arrays containing the enable signal of the various partitions. If more than one is found to be high or if no instance is active, an error is reported. The SystemVerilog code of the class is reported in Listing 4.5

```
class FSM_single_active;
  // Define class variables
  string msg;
  int num_inst;
  bit count_active = 0;

  function new(int num_inst = 0);
    this.num_inst = num_inst;
  endfunction

  function bit update_FSM(logic enable_arr[]);
    // Check that one partition is enabled
    for (int i = 1; i < num_inst; i++) begin
      if (enable_arr[i] == 1)
        count_active = count_active + 1;
    end

    // Update the error/pass messages and return
    if (count_active > 1) begin
      msg = "More than one instance is active";
    end
    else if (count_active == 0) begin
      msg = "No instance is active";
    end else begin
      msg = "Passed";
      return 1;
    end

    return 0;
  endfunction
endclass
```

Listing 4.5: SV of the checker that verifies that a single partition instance is active

4.2 UVM Test Bench

The current section presents a profound description of the UVM modules needed to support the library's checkers. Due to the vast number of possible combinations among checkers and IPs in the various configurations (chip or partition level simulations, RTL or Netlists), it is not possible to explain the code generated for each of these cases. The following explanation of the various modules, classes, components, and their organization will be completely unrelated to a specific use case. The description will attempt to be as generic as possible.
It is important to notice that even if the standard UVM nomenclature for the classes is used, their functionalities could have been changed and do not entirely comply with the methodology standard.
A high-level schematic of the organization of the various UVM components is represented in Figure 4.4.

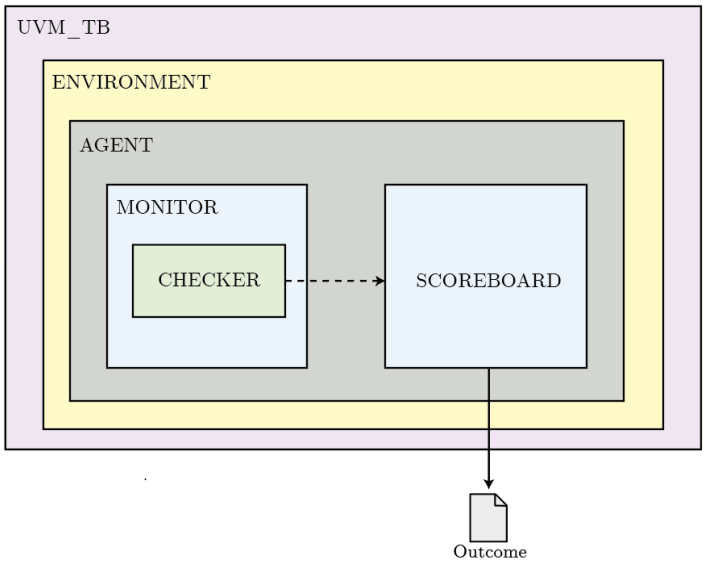

Figure 4.4: Organization of UVM components

4.2.1 UVM TB Module

As described in the chapter's introduction, the UVM test bench is the component that allows instantiating all the needed checkers for a specific simulation. Its

purpose is to define and create the SV interfaces that will work as adapters between the DUT and the checkers. This component is the only *module* of the whole framework because it has to connect the interfaces to the DUT's signals using their paths. The connection of interfaces is a task that a SystemVerilog class cannot do since it is not part of the modules hierarchy and does not have the concept of "path". A SV *class* can only deal with interfaces. For this reason, after connecting the interfaces to the DUT's signals, the UVM TB stores them in the UVM configuration DB so that the various classes can retrieve them. The relationship among the UVM TB, the Device Under Test, the interfaces and the UVM configuration DB is graphically represented in Figure 4.5. The steps that the module does are:

- Define the needed interfaces
- Connect each interface to a signal using the signal's path
- Store the interfaces into the UVM configuration DB
- Execute the UVM code using the *run_test* function

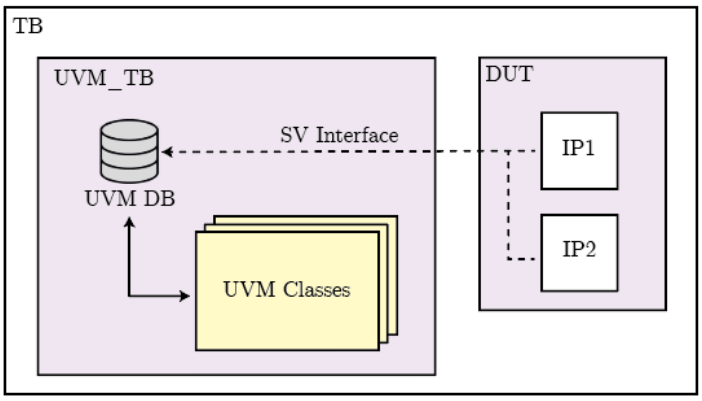

Figure 4.5: Relationship among UVM TB, DUT, interfaces and UVM DB

4.2.2 UVM Environment Class

The UVM Environment is an elementary component. Its role is to define and create the UVM Agents Objects required for the simulation. They are logically representing the partition instances of the DUT. Indeed, as discussed in Section

2.2, chips are typically composed of many instances of various partitions. The instances of a given partition are identical since they contain the same clusters and IPs. Sometimes is necessary to verify the IPs in all the various partitions, for example, during HTOL simulations. These simulations' goal is to test the reliability of a chip when working at high temperatures and to achieve this purpose, all partition instances of a chip are executed in parallel. The UVM Environment class is instantiated in identical objects that contain all the UVM components needed to verify the IPs' behavior. If the simulation is an HTOL, then the framework will create several UVM Environment objects equal to the number of instances. Else, for all others simulation, a single Environment will be created.

4.2.3 UVM Agent Class

The UVM Agent is a SystemVerilog class whose purpose is to contain the UVM Monitors and the UVM Scoreboard. Being wrapped in the same class, the Monitors can easily communicate through the analysis port the outcome of the check to the Scoreboard, which will display the summary report after the simulation terminates.

If the UVM Environment corresponds to the instances of a partition, the Agents logically represent the various checkers. All UVM Monitors inside a given UVM Agent instance will implement the SV code of a specific checker. It is not strange that the same of the various Agents classes contains the name of the checker since, in this way, they will have unique names.

4.2.4 UVM Monitors Class

The UVM Monitors classes are the core of the framework. Indeed, they instantiate the code of the checker defined in the library. Since every checker can be applied to multiple IPs, a Monitor object represents the checker verifying a specific IPs of the DUT.

The task of the Monitor is to retrieve the interface relative to the signals of a given IP. This operation is possible through a naming convention. The interface related to a given checker applied to a given IP is labeled in the UVM DB with a name indicating all the information. This way, the Monitors (who know the

checker's name and IP) can reconstruct the SV interface name and correctly retrieve it. During the run phase, at every clock cycle, the checker is updated, passing as arguments the values of the IP's signals, and the current outcome of the checker is saved. Then the result is forwarded to the Scoreboard together with some information needed by the report generator (e.g., name of the IP, possible error messages).
To summarize the tasks of the Monitors objects are:

- Retrieve interface
- Instantiate checker object
- Retrieve signals' values from interface
- Update checker and get outcome
- Forward outcome and meta information to UVM Scoreboard

4.2.5 UVM Scoreboard Class

The Scoreboard Class has the task of generating the summary report that will be logged and that will help the engineers during the debug.
The core of the class is the *write* method that receives the values written by the various Monitors of the UVM Agents. The data that it receives are:

- IP name
- Outcome of evaluation (PASS/FAIL)
- Error/warning message

The scoreboard stores this information in associative arrays for the duration of the whole run phase. Then, during the report phase, it iterates over the array and prints the various information of each IP in a tabular format composed of ASCII values. An example of a report summary generated by the Scoreboard is shown in Figure 4.6.

UVM SUMMARY

IP_NAME	STATUS	MESSAGE
IP_01	PASSED	Warning: isolation signal not present;
IP_02	PASSED	Warning: isolation signal not present;
IP_03	PASSED	Warning: isolation signal not present;
IP_04	FAILED	Pwr_small not turned off
IP_05	FAILED	No signals for this IP:
IP_06	PASSED	Ok
IP_07	PASSED	Warning: prw_big signal not present;
IP_08	PASSED	Warning: prw_big signal not present;
IP_09	PASSED	Ok
IP_10	PASSED	Warning: prw_big signal not present;
IP_11	PASSED	Warning: prw_big signal not present;
IP_12	PASSED	Ok
IP_13	PASSED	Ok
IP_14	PASSED	Ok
IP_15	FAILED	No signals for this IP:
IP_16	PASSED	Ok
IP_17	PASSED	Ok
IP_18	PASSED	Ok
IP_19	PASSED	Ok
IP_20	PASSED	Ok

Figure 4.6: Example of summary report produced by the UVM Scoreboard

4.3 Automated Checker Library

The generation of the UVM test bench that allows the instantiation of the checkers of the library must be automated to reduce the effort required by the user. Since the SV code of the classes composing the UVM module is quite standard, it can be easily generated by a script, provided that some information is provided to it.
Among others, the information related to the chip under test (such as IPs' path or pins polarity) is the one that is more complex to retrieve since there is not a centralized location in which the script can recover it. This issue has been solved by developing a second script that can generate the information of a given project and store them in a centralized location using a user-friendly file format.
The two scripts and the needed configuration files are analyzed in the following sections.

4.3.1 Generation of Structural Information

The first requirement is to retrieve and store all the needed information related to the tested chip. Each checker can be connected (through SV interfaces) to the signals of the chip. As a consequence, the script needs to be aware of the hierarchical organizations of IPs, clusters, and partition of the chip, as well as the pin characteristics (e.g., number of bits, polarity, name) of each IP in order

to automatically connect the generic checker with the specific IP. Retrieving this information is an issue since the information on the design of a chip is not located in a single point, and often, the information must be extracted from various files. All this data could be generated by an engineer, which would need to retrieve all the information from the project documentation, and from the various HDL files and should save them in a usable format, updating them every time the design is modified. This manual work is an effort that would limit the use of the framework. For this reason, a python script has been developed, allowing the automated retrieval of as much information as possible on the chip's design. The goal is to produce a file for each cluster of the chip containing the information (name, path, list of pins) of the IPs of the cluster itself. This way, the python program that instantiates and generates the UVM-based TB can have a well-defined point of access to the needed information. The format used to store these data is JSON since it is human-readable and easy to manage with python.

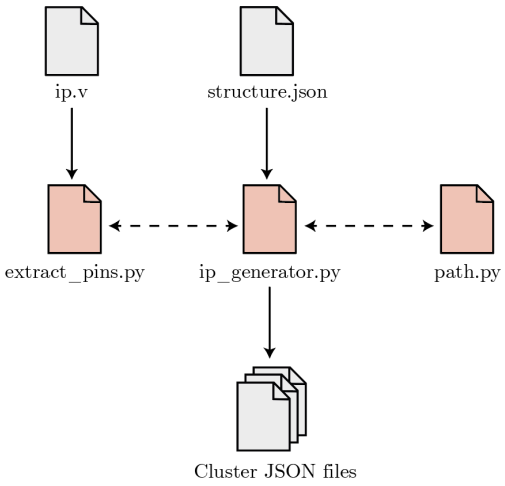

Figure 4.7: Python scripts and files for generation of chip's information

As is shown in Figure 4.7, the generation of the cluster files is a task performed by the script *ip_generator.py*. It requires obtaining some information from the **structure.json** file that describes the hierarchy of partitions, clusters, and IPs of the project. This file is quite static, and it must be manually created once and then can be easily updated when some components of the chip are changed. The

use of a JSON format for this file is because it can represent hierarchies, which is very helpful when describing a chip composed of nested elements.
As can be seen in the example of *structure.json* for the mock project in Appendix A, the *structure* field contains the information on the hierarchy stored as nested associative arrays. As first level, the various partitions (e.g., *partition_A* and *partition_B*) are present. Each contains its second-level hierarchy elements, which store the data of the various clusters of IPs, divided into megacells and hardips. Each cluster shows the IPs it contains, also indicating the number of instances of each of them.

The **ip_generator.py** script scans the structure file, and for each of the cluster of IPs found, it generates a file storing the details of each of the IPs. To do so, it exploits two different programs (written in python as well) to retrieve the information of each IP. The first one is called **path.py**, and as the name suggests, it can reconstruct the path (of both RTL and Netlist) of the IP inside the chip's hierarchy. The path of an IP is strictly related to its position in the hierarchy, and it typically follows a standard naming convention. For this reason, starting from the position of an IP in the *structure.json* file, it is quite easy to reconstruct the path of the IP itself.
The script **extract_pins.py** is used to retrieve the pin information of each IP. It searches for a Verilog or SystemVerilog file containing the name of the IP inside some predefined folders. When it finds it, the file is parsed, looking for the module name and its input and output pins. From the Verilog, the script can obtain the list of pins, their names, and their dimension (in bits). Moreover, starting from the pin's name, the script tries to extrapolate further information. Indeed, often the name of the pin indicates some important features of the pin itself, such as its functionality (*aon_iso* is an isolation signal), its polarity (a pin terminated with *_n* or *_L* is a signal which is active low), and its direction (*_in* and *_out*). Using regular expressions is quite trivial to retrieve this information from the pins. It must be noticed that even if many pins of the project contain such information in their names, some do not embed them, and the script cannot assume any pin's feature. In this case, the user has to manually update the file inserting the missing information.
After the *extract_pins.py* script is executed the fields for each pin returned to

ip_generator.py are:

- properties: list of tags of the pins (needed for the checker library)
- direction: input or output pin
- verifOnly: boolean to indicate if the pin was removed during the synthesis and it is not available in Netlist
- polarity: polarity of the pin (high/low)
- pin_width: number of bits of the pin

After the *ip_generator.py* script retrieves the needed information on each IP and its pins, it stores them into a dedicated file for each cluster. An example of the content of JSON file of a cluster produced by the script is presented in Appendix B. It must be noticed that the script stores the various cluster files in a hierarchy of directories similar to the chip structure. In this way, the script dedicated to the generation of the UVM code can retrieve the position of a given IP using the relative position of its JSON file in the file-system and does not have to reuse the *structure.json* file.

4.3.2 Generation of UVM TB Module

A python script called **uvm_generator.py** has been developed to automatize the generation of the UVM test bench and of the nested SV classes. The script requires external files and information in order to be able to generate the needed UVM components. A high-level schematic of the various components involved in the generation process is shown in Figure 4.8. The various elements are described in the following paragraphs.

CSV Control File The python script does not know which checker must be applied to the IPs, so the engineer launching the simulation must specify the relationships between checker and IPs. This selection can be made through a comma-separated values (CSV) file defined for each partition of the chip, that stores in a tabular format the checkers that have to be applied to the various IPs of the partition. This format has been chosen since it is both human-readable and

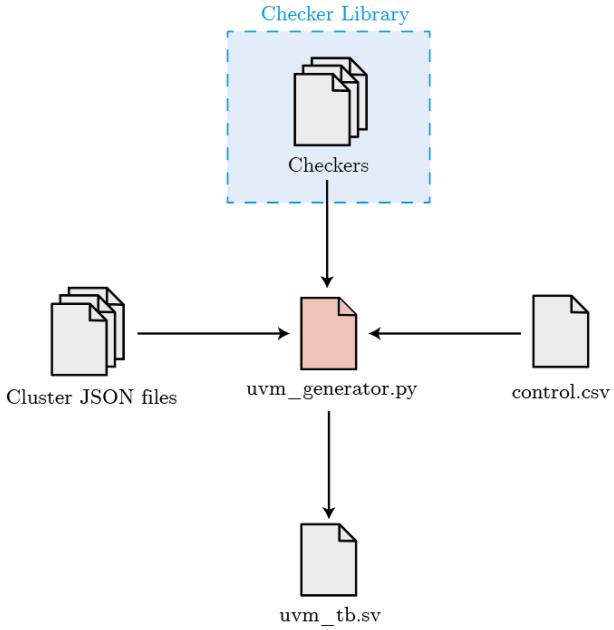

Figure 4.8: Script and files involved in the UVM TB generation

easily manageable with a python script (e.g., using *Pandas* library). Moreover, it is a textual file, allowing a simple integration in version control software. A simple example of the CSV configuration file is shown in Table 4.2
To apply a checker to a given IP, the user has to modify the CSV file by inserting "T" (True) in the cell related to the corresponding IP and checker. Similarly, to remove a checker from an IP, the user has to insert "F" (False) in the correct cell. It is important to notice that the last row of the table is used to indicate which will be the active partition during the simulation. This way, when working at the chip level, the tool can instantiate the checker that verifies that the correct partition instance is active. The accepted values for the *active_partition* field are:

- none: partition level simulation
- 0-N: indicates that only a given instance of the partition must be active
- all: checks that all instances are running in parallel (HTOL)

Ip_Name	check_default	check_address	check_pwr_seq
mc_1	T	T	F
mc_4	F	T	T
hp_1	T	F	T
hp_2	F	T	F
active_partition	0		

Table 4.2: Example of CSV control file for selecting which checker has to be applied to various IPs

Clusters JSON Files These are the files produced by the *ip_generator.py* script presented before. They contain all the details about each IP such as the RTL and Netlist paths and all the pins information.

Checker Each checker is composed of two elements: a JSON setup file and the file storing the SV class. The SV file contains the code of the checker, and it is directly included in the UVM Monitors. For this reason, it is not directly used by the python script to generate the UVM TB code. On the other hand, the JSON file contains meta-information about the checker that is used to create the UVM TB. So, it is a mandatory file since it is required by the script to generate the SV code composing the UVM TB. The JSON file has to be defined every time a new checker is created, and it must contain the following fields:

- name: full name of the checker
- code: string used to generate unique classes' names in SV
- description: adding details on the checker purpose
- fsm: name of the file storing the SV code of the checker
- generic: boolean indicating if the checker is generic or specific
- clock: name of the clock that synchronizes the FSM
- signals: list of signal types that will feed the FSM

The most important field is *signals* since it allows for the definition of some "tag" that the framework will use to identify the pins of the IPs under test. The user creating the checker can specify keywords that represent the signals under test. These tags then must be inserted inside the JSON of the IP, specifically in the field *properties* of the pins. In this way, the script will be able to identify the signals that have to be verified by each checker. As shown by the example in Listing 4.6, the tags *iso*, *mem_pwr_small*, and *mem_pwr_big*, which represent isolation and power signals, must be inserted in the *property* field of three pins of the IPs under test. The script, knowing which are the tags of the checker, will be able to retrieve the path of the signals that need to be verified. This mechanism is an adapter between the checker's generic definition and the IPs' specific structure.

It is important to underline that the name of the field *fsm* could be misleading. Indeed, this field indicates the file's name containing the SystemVerilog code that implements the checker's logic. The code does not necessarily have to implement a FSM, but any kind of control of the signals is acceptable. This naming convention derives from the early stages of the project development and has not been updated.

```
{
  "name": "powerup sequence",
  "code" : "pwr_seq",
  "description": "check power-up sequence is correct",
  "fsm": "FSM_pwr_seq",
  "generic": false,
  "clock" : "_if_clock.clk",
  "signals": [
    "iso",
    "mem_pwr_small",
    "mem_pwr_big"
  ]
}
```

Listing 4.6: Example of a checker's JSON file

Python Script The script's purpose is to retrieve the information from the JSON and CSV files and generate the code of the UVM TB.
The whole process starts from the opening and the retrieval of the checker-IPs relationship from the CSV configuration file. The script uses *Pandas*, a common python library that simplifies the management of various tabular file formats. The script opens the file and stores it in a dictionary. The dictionary's keys are the checkers, while the values are the list of IPs that have to be verified by the checker. This information allows defining the UVM Environments, Agents, and Monitors. To make an example, using as reference the data presented in Table 4.2, the script will generate only one Environment object since, in the *active_partition* field, only the IPs of a specific partition have to be tested. Then the Environment will instantiate three different UVM Agents since all the checkers present in the CSV must verify at least one IP. Eventually, each of the three agents will contain several UVM Monitors equal to the number of IPs that need to be verified by each checker (e.g., two UVM Monitors in the Agent related to the *check_default* checker).
Subsequently, the script uses the information present in the JSON file of the checker to instantiate inside the UVM Monitor classes the correct checker. Indeed, as reported previously, the file contains the field *fsm*, which indicates the file's name containing the SV class, which implements the checker logic. Since the file is part of a library, no path is needed, and just the name allows its identification. Given the name of the file, the script can add a line in the SV code of the test bench and import it to make it usable by the UVM Monitors.
The last thing that the script has to do is the creation of SystemVerilog interfaces and their connection with the signals of the IPs. The management of interfaces is done using the mechanism of "tags". Each checker defines some signal types in its JSON file's field *signals*. The checker writer places these tags into the field *properties* of the pins of the various IPs. In this way, the script can understand which signals of the DUT need to be verified, and it can connect them to the SystemVerilog interfaces using the path of the IP and the name of the pin.

The code generated by the script is typically composed of hundreds of lines of code. A template system has been used to avoid the creation of the whole UVM TB from scratch. Indeed, even if each simulation will have a custom code depending on the content of the configuration files, the basic structure of the various modules

and classes is very similar. The code is added to the template files of the various UVM classes and then merged, allowing easy readability of both the python code and the SV generated.

4.4 Force and Measure of Analog Signals

As mentioned in Chapter 3, when playback simulations are executed,force or read values of analog signals are lost since the ATP file cannot describe them. This issue leads to long, manual debugging sessions, waveform reviews, and reduced test coverage. This section describes the extension of the checker library framework with analog checkers and UVM Drivers to support the force and measure of analog signals during the playback simulations.

As shown previously, a UVM test bench can be easily instantiated inside a pre-existing Verilog TB. A similar approach can also be applied to the TB generated from the Test Vector file before starting the playback simulation. It is enough to instantiate a UVM test bench inside the one obtained from the ATP, connect the UVM Monitors or Drivers to the analog signals of the Device Under Test, and then it is possible to read and write them.
The idea is to extend the functionalities that the checker library framework to allow the definition of an **analog checker**, which can monitor and verify that the signals of an IP assume specific analog values during the simulation and a **driver component** that can force analog values in the analog signals of the DUT. A high-level schematic of the relationship of the components that will extend the framework can be observed in Figure 4.9.

It is crucial to underline that this part of the project has not been fully integrated into the checker library framework. A first working prototype has been successfully developed but requires some refinements and improvements. More specifically, it must be made project independent and requires a better mechanism for making the framework able to force or measure the analog value at the right moment in time. Indeed, the current version can force or measure analog values using some trigger registers as a reference. Depending on the value stored in a given trigger register, the framework knows that it has to force (or measure) a predefined value into a

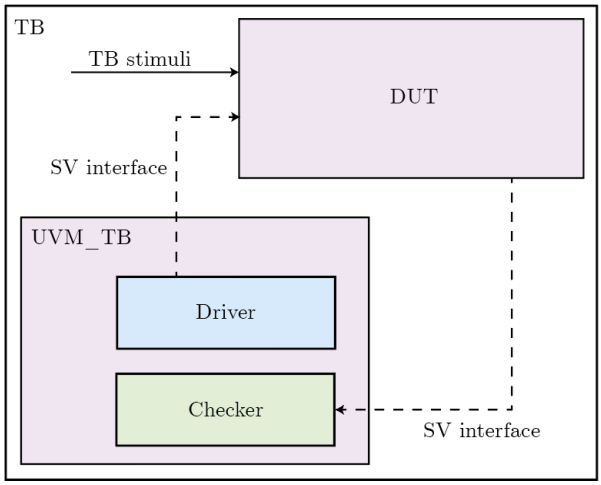

Figure 4.9: High-level schema of relationship between TB, DUT and UVM_TB in context of analog signals

given signal. However, this method is not accurate enough since there is not always a one-to-one relationship between the trigger register and the analog operation. Indeed, analog operations may have to be done after some delay from the change of the trigger register. In other cases, an analog value has to be forced or measured even if no changes happen in the trigger register. In the last paragraph of this section, an improved mechanism is proposed to solve this limitation.

4.4.1 Automated UVM TB Generation

Since the support of analog operations is still not fully integrated into the framework, some custom configuration files and scripts have been developed to automatically generate the UVM-based test bench instantiating the needed checkers and drivers. Figure 4.10 represents the various scripts and configuration files involved in generating the SV code. It is important to notice that the script *uvm_generator.py* is different from the one presented in the previous section, even if they have very similar functionalities.

Two configuration files are exploited to provide simple user interaction with the python script that generates the needed code. The file format has been decided to be CSV. The content of the configuration files represents in a tabular format

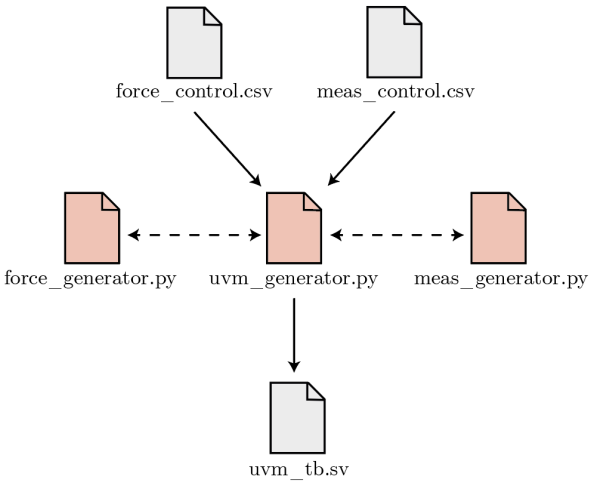

Figure 4.10: Python scripts and CSV configuration file relationship

the relationship among the simulations, registers values, and signals values. Two files are separated to distinguish the configuration information when forcing and measuring the analog signals. An example of the file for the force is shown in Figure 4.3 (even if displayed in tabular format, the files are CSVs).

The columns of the files are identical. Their meaning is:

- Sim Name: name of the simulation to which apply the other information
- Reg Path: RTL path of the trigger register
- Reg Dim: number of bits of the register
- Reg Val: triggering register value
- Analog Path: RTL path of the analog signal
- V/I: flag that indicate if the analog value is a voltage (V) or a current(I)
- Val: analog value to force or compare in Volts or Ampere

Sim Name	Reg Path	Reg Dim	Reg Val	Analog Path	V/I	Val
sim_1	reg_1	4	2	signal_1	V	1.5
sim_1	reg_1	4	4	signal_1	V	0.5
sim_1	reg_2	6	3	signal_2	I	10.3
sim_2	reg_1	4	5	signal_1	V	1.7
sim_2	reg_1	4	4	signal_3	I	0.2
sim_2	reg_3	6	6	signal_4	V	1.9

Table 4.3: Example of Force Configuration File

Three python scripts have been defined to remove some effort from the user. Their task is to generate the UVM-based test bench, which instantiates all the classes and modules needed to force and measure analog signals during the playback simulations. The main script is called **uvm_generator.py**, and its job is to retrieve the information about the trigger registers and analog signals for the current simulation for both analog operations. Once the two CSV files are parsed and the needed data is retrieved, the program interacts with the **force_generator.py** and **meas_generator.py** scripts sending the respectively the information about the force and measure operations. The two programs have very similar logic since they both have to generate the needed SV code of the various UVM components. Once they create the code, it is returned to the main script, which puts all the generated code together in a single file and generates the *uvm_tb.sv* file. The file code is not fully generated from scratch; templates are used. These files contain the constant and fixed code and add the needed code in specific template points to compose the entire file. The python library *Mako* has been used for this project.

The decoupling of the code generation for the force and measure cases is due to two main reasons. The first one is that separating the scripts reduces the complexity of each of them and increases the readability. For example, having a single program for retrieving the information from the files and generating the UVM code would lead to a vast, unreadable, and not maintainable file. The second reason is that any combination of reading and force of the same signals is possible, and this would be

complex to manage in a single script. In addition, having a component dedicated to creating the components for the force and measure simplifies the script logic.

4.4.2 Possible Improvements

The main issue of the developed prototype is related to the mechanism which triggers the framework to force or measure specific analog values during the simulation. Indeed, using the values of some registers to understand which analog operation to perform is not precise enough since there can be operations that are not related to the value of a register. A possible solution is proposed to overcome this limitation. The idea is to avoid using the CSV configuration files and retrieve the information about the analog operations from the comments generated during the conversion from the ATP file to the Verilog test bench. Indeed, all analog operations are represented as comments in the ATP file. These are ignored by the tool that converts the ATP into Verilog TB. Therefore, enabling the report of the comments from the ATP to the Verilog of the tool that carries out the conversion will allow getting the comments related to the analog operations in the Verilog file. Then, the Verilog TB file can be parsed by a python script which can retrieve all the comments describing the analog operations and the simulation time at which the framework should execute them. This way, all necessary information can be retrieved automatically without using CSV files and trigger registers.

Chapter 5

Results

Evaluating the impact of the UVM-based framework is not a simple task. Indeed, there is no objective metric to assess the improvements in the user workflow brought by the proposed solution. The previous approach was based on manually written Tcl files and waveform reviews, and it is not easy the estimate the time required to verify that all IPs of a chip were correctly behaving.
In the next section, a more detailed analysis of the old and new flows is made to underline the reduction in effort and time that the framework can bring. Even if workflow comparison is evidence of the improvements that the framework introduces, some objective and measurable metrics are necessary to prove that the presence of the UVM components does not affect the simulation performances heavily. Indeed, having an automation tool that slows down the simulation by many factors would lead the user to employ the old workflow. In the second section of this chapter, the information retrieved by a profiler is used to understand how much the framework impacts the simulation performances.

Eventually, it is essential to underline that one of the most remarkable results obtained by the developed framework is that it has been integrated into Apple's regression environment. Moreover, the tool has been presented to the engineers, who understood and appreciated the improvements that this tool can introduce.

5.1 Updated User Workflow

Before the development of the checker library framework, the user workflow to verify specific signals behavior was the following: a Tcl file containing many *Compare* commands was created by the user, which then had to execute the simulations to verify that the behavior was correct. It is crucial to notice that the Tcl file was not universal and reusable for each IP. Indeed, in the Tcl file, the user had to specify the path of the signal that is compared. Dealing with signals' paths implies that they must be retrieved (which can become quite a time-consuming task) and that the Tcl files are not reusable since every IP has different paths. Furthermore, for specific cases, the user had to perform a waveform review of the simulation to ensure no errors were present. It is clear that creating multiple Tcl files, retrieving hundreds of signals' paths, and manually reviewing dozens of waveforms can require days and can become a huge bottleneck in the DV process.

The user workflow with the integration of the checker library framework has undergone some improvements. When a user must define a new checker, she has to create the JSON file containing the meta-information of the checker. Then she has to generate the logic of the checker, written in SystemVerilog, which supports many powerful features and is not limited to a simple comparison. Moreover, the checker can be written in a way to be generic and reusable for multiple IPs in parallel and does not require the user to have any knowledge of the signals' paths. Once the checker has been created, it can be applied to the needed IP modifying the CSV control file and launching a simulation using the traditional regression environment.
Even if a statistical and objective comparison between the time and effort required by the two workflows cannot be made, it is pretty evident how the grind needed from the user is reduced using the checker library.

5.2 Performance Impact

In this section, the performances of the UVM-based framework are compared with the ones of the original flow. In this way, it is possible to evaluate the impacts of UVM on the simulation execution. This crucial information needs to be analyzed

to understand the framework's usability. For example, suppose the increase of many factors in the simulation execution time or a considerable amount of memory is allocated. In that case, a user may prefer to avoid using the framework and keep using the traditional flow.

The *xprof* profiler (embedded into the Xcelium simulator) has been used to obtain relevant data from the simulations. The profiler allows for extrapolating the data relative to the following fields:

- Memory Usage (Megabytes)
- CPU Usage (seconds)
  - System Time
  - User Time
- Total Simulation Time (seconds)

Extracting statistically relevant data implies the execution of each simulation multiple times to avoid the data being affected by outliers. In this project, it has been decided to execute each simulation ten times. This number has been defined as a trade-off between gathering relevant information and a limited simulation time.
As will be analyzed in the following sections, outliers are possible due to how the simulation software manages simulations. Indeed, Xcelium is distributed among the various users, which implies that the environment in which the simulations are executed is not predefined. The performances depend on the load of the farm servers at a specific moment. The software has a scheduler to balance the allocation of resources for each simulation. Still, the resources are finite, and some simulations can have massive time due to a lack of resources at a given moment. Since there is no way to have an identical simulation environment, it has been decided to launch the ten instances per simulation type simultaneously, relying on the optimization made by the scheduler to balance the average execution time.

5.2.1 Checker Library Impact

The features of the checker library allow having a variety of different simulations. Indeed, the DUT could be a partition or a chip, defined as RTL or Netlist, and the different types of checkers can be applied at each simulation. To have a uniform and an exhaustive number of different simulations, ten simulations were performed for each combination of the following characteristics:

DUT Type	IP Type	Checker Type
RTL Partition	Megacell	Specific
RTL Chip	HardIP	Generic
Netlist Partition	HTOL	

Table 5.1: Possible simulation configurations

Memory The impact of the checker library on memory usage during the simulation is basically influential from the user's point of view. Executing the various simulations for the different combinations of RTL, checkers, and IPs, it is clear that the memory used during the UVM simulation is slightly higher than during the traditional simulation. An example is shown in the graph below, which refers to a simulation of an RTL at partition level applying a specific checker. The increase in memory usage is less than 60 Megabytes, and considering that the simulation with no UVM needs almost 1.4 Gigabytes, the UVM impact is quite negligible. As expected, the quantity of used memory in the ten simulations is constant since the loaded data is always the same and is not affected by external factors. Since the memory impact is equal to or lower for the other simulation configurations, their specific memory usage is not explicitly reported but will be shown in a summary table.

CPU Usage As stated previously, the CPU usage statistics are divided into two subgroups: system and user. The user's CPU usage refers to the amount of time the processor has spent in the memory's user space. Similarly, the system's CPU

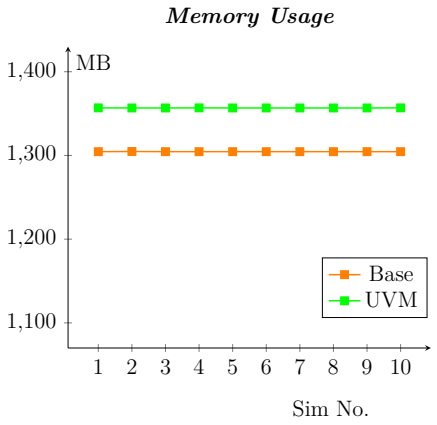

Figure 5.1: Memory usage of base and UVM simulations

usage indicates the number of seconds spent in the memory's system space in order to execute kernel-level operations.
As will be proved with the obtained statistics, the user CPU usage is typically higher since the user space contains the program and data information and, consequently, is the most used during the simulation.

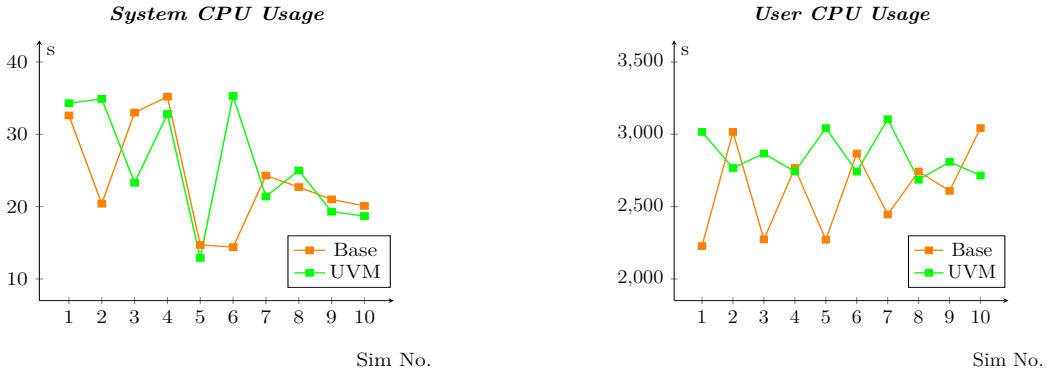

Figure 5.2: System and User CPU usage for both base and UVM version

As seen from the two charts above, the relation between CPU usage for the base and UVM versions is not as stable as the memory usage. Indeed, for both the user and system CPU usage, it is possible to notice the presence of some outliers.

In some simulations, the basic system requires more system CPU usage than the one with UVM. This probably depends on how the scheduler manages the various simulations under load. Since the needed time is relatively low (typically below 40 seconds), the impact of a reschedule may impact a lot on the total CPU usage. Different behavior can be noticed in the user's CPU usage. Indeed, in these cases, the time is, on average, higher (around 40 minutes), so the impact of the scheduler is typically lower. The time required for executing the UVM simulations is generally higher.

Total Simulation Time The total simulation time is the time elapsed between the launch of the command for executing the simulation until the termination of the simulation itself. Of course, some operations have to be executed before starting the simulation in this time frame. Indeed, the whole process can be divided into three phases:

- Compilation
- Elaboration
- Execution

The first one is the phase in which the various HDL modules are compiled, the elaboration is the process that links the various compiled elements, and the execution is the phase in which the simulation is run. All these phases require some time to be completed, and they compose the time that the user has to wait before seeing the simulation terminate.
Since this metric indicates how much an engineer has to wait before the simulation ends, it is fundamental that the UVM modules do not increase it too much. Indeed, if executing a simulation with the checker library takes much longer than the simulations without it, then the user would not use it. Given the importance of this information, the charts of the total simulation times for the possible configuration combinations are reported. The charts are organized in the following way: there are seven groups composed of three charts each. The first three represent groups describe the simulation times for specific, generic, and both checkers applied to a Megacell. The following three represent the simulation times with checkers applied to an HardIP. The last one is related to the execution of HTOL simulations that apply specific, generic, and checkers.

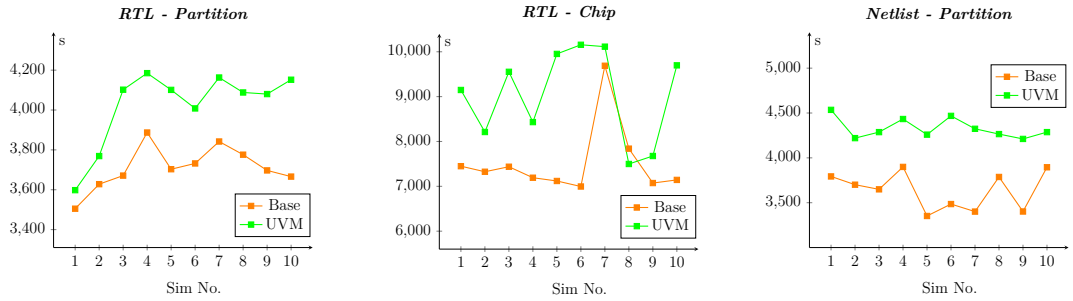

Figure 5.3: Simulation times with specific checker applied to Megacell

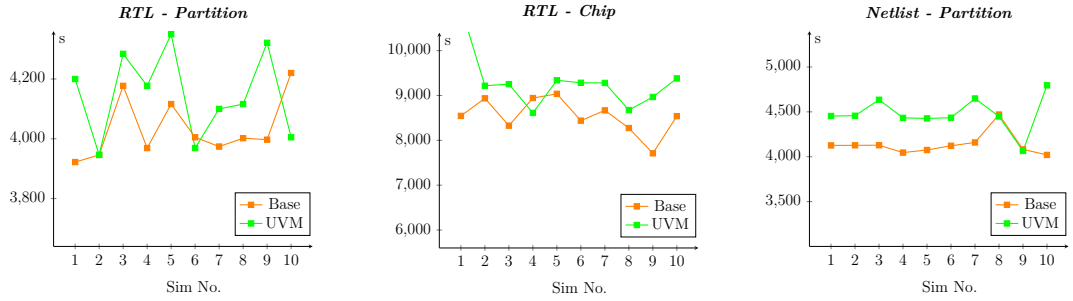

Figure 5.4: Simulation times with generic checker applied to Megacell

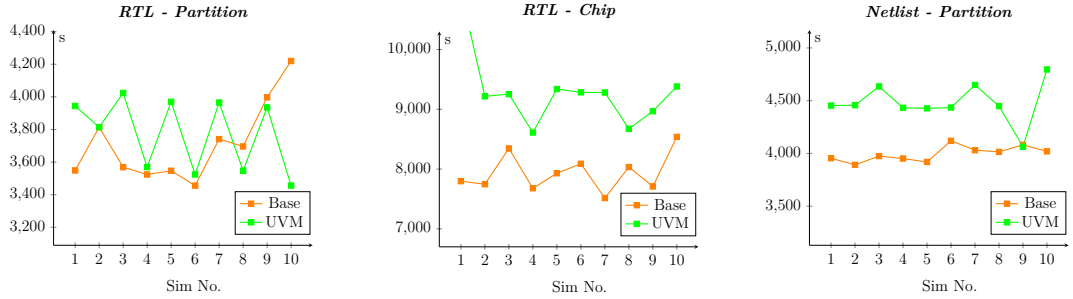

Figure 5.5: Simulation times with specific both a specific and generic checkers applied to Megacell

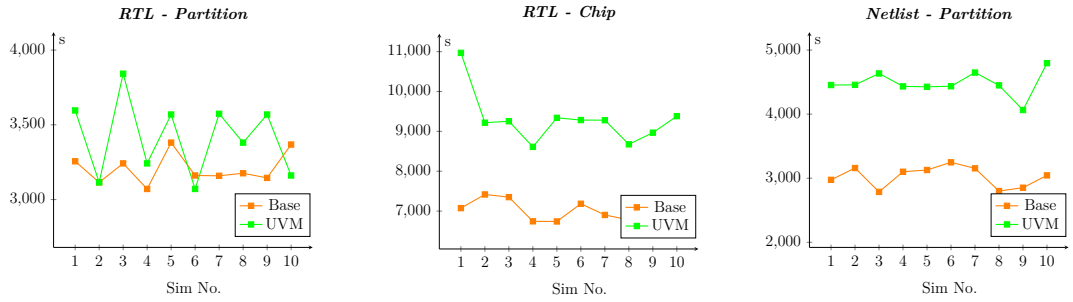

Figure 5.6: Simulation times with specific checker applied to HardIP

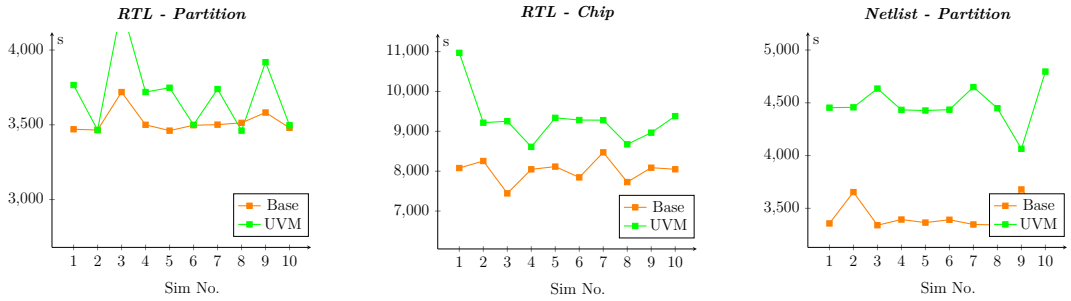

Figure 5.7: Simulation times with generic checker applied to HardIP

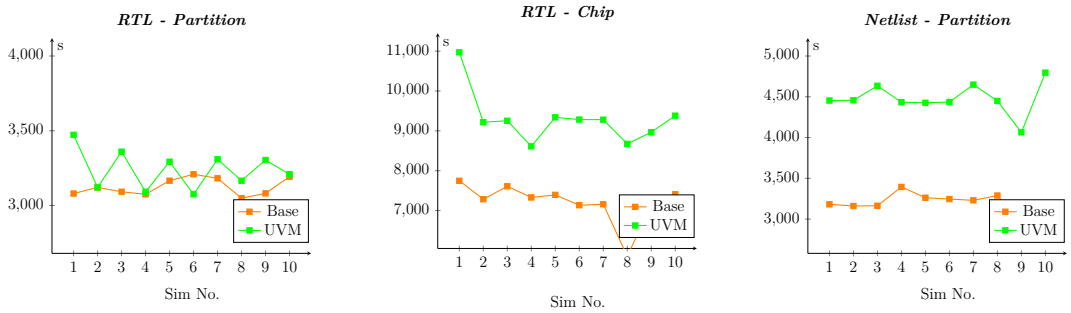

Figure 5.8: Simulation times with both specific and generic checkers applied to HardIP

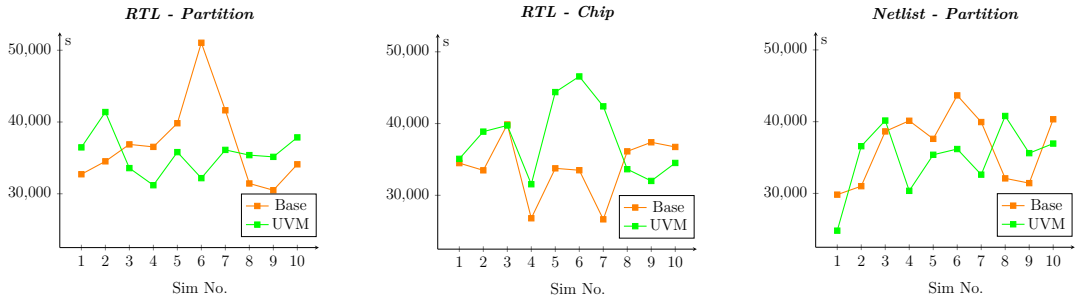

Figure 5.9: HTOL simulations with specific, generic and both checkers

As reported by the various charts, the simulation times when UVM is enabled are typically slightly higher than those in which it is not activated. A pattern that can be easily visualized is the irregularity of some simulations. Indeed, if some simulations have a very regular behavior of both the base and UVM cases, as the one shown in the first charts of Figure 5.1, others have a very unstable simulation time, as can be seen in the first chart of Figure 5.3.
These very diverse cases could be again due to a distributed environment and the management that the scheduler applies to a specific simulation. Since the simulation time lasts even more than an hour, the impact that the scheduler can have is considerable. A simulation could be stopped because of a resource shortage in a given time frame or because a higher-priority simulation was launched. Notice that the irregular behavior happens in both the base and UVM simulations and with entirely different configurations. Consequently, no correlation between some simulations' unstable behavior and the framework's use can be made.

Data Aggregation In this paragraph, the aggregation of the various data is made to have an insight into how the various simulation configurations behave on average. For example, the first data aggregation was made to see the impact of the checker type during the simulation. More specifically, the type of checkers, the IP under test, and the nature of the DUT are used to aggregate the needed values.

As can be seen from Table 5.2, the impact on memory does not depend on the checker type since it just depends on the size of the checker code that is loaded into the memory. Indeed, when two checkers have loaded into memory, the impact also increases. The CPU impacted the total simulation time are both around 10%, and there is no relationship depending on the type of the checkers. On average, the simulation time using a single specific checker is higher than when both the specific and the generic have been applied. This metric shows the absence of correlation between the checker types and the performance impact on the simulation.

Table 5.3 shows the performance impact of the checker library framework when different types of IPs are verified. As could be expected, this aspect does not impact all the performances. In both cases, the increase in memory is below the 3% for standard simulations, while the increase in time (both the CPU and simulation

	Memory	System CPU	User CPU	Sim Time
Specific	+2.68%	+8.01%	+11.11%	+12.41%
Generic	+2.66%	+4.89%	+6.03%	+7.84%
Both	+2.70%	+3.17%	+10.64%	+6.95%

Table 5.2: Performance impact of the framework when using different checker types

ones) is between 5 and 10% for both Megacells and HardIPs. Therefore, it can be deduced that the type of IP verified by a checker does not change the performance impact.

	Memory	System CPU	User CPU	Sim Time
Megacell	+2.58%	+7.04%	+13.21%	+10.53%
HardIP	+2.78%	+5.12%	+11.08%	+10.17%

Table 5.3: Performance impact of the framework when applied to different IP types

Another aspect that needs to be analyzed is the DUT nature's impact on the performances. Table 5.4 shows that the DUT impacts the library's performance. The memory is much more impacted when working at partition level for the chip level simulations since the checkers and UVM TB code composes a larger part of the compiled code. The partition is a subset of the whole chip so less code of the DUT is present when working at the partition level. Also, the use of the DUT's Netlist reduces the framework's impact since the partition's gate level description is much more complex than the RTL, which describes the DUT at the register level. This aspect is reflected in the CPU usage and simulation time. Indeed, the complexity of this type of simulation is clear, and using UVM with these DUT configurations is impacting the performances.

Eventually, the statistics about the HTOL simulations are reported. As shown in Table 5.5 the impact of the checkers on a very long and heavy simulation is almost

	Memory	System CPU	User CPU	Sim Time
RTL Partition	+4.18%	+2.72%	+7.02%	+8.79%
RTL Chip	+1.02%	+11.15%	+23.68%	+10.26%
Netlist Partition	+2.83%	+4.37%	+5.74%	+12.02%

Table 5.4: Performance impact of the framework depending on the DUT nature

irrelevant. The increase in memory usage is similar to the chip-level simulations since in both cases the UVM code is relatively small for the one of the whole chip. Interestingly, the total simulation time is increased on average by less than 1%. This is a very positive result because the execution time of the HTOL simulations is typically longer than a day and even a 10% increase in the run time would lead to many hours of waiting.

	Memory	System CPU	User CPU	Sim Time
HTOL	+1.05%	+3.71%	+1.16%	+0.99%

Table 5.5: Performance impact of the framework on HTOL simulations

Average Impact In this last paragraph, the summary table representing the average impact of the UVM framework is displayed. Table 5.6 indicates that the impact in terms of memory is completely negligible. Some Megabytes more are needed to use the UVM-based TB, which is completely negligible from the user and performance point of view. The simulation times are increased on average by about 10% and this can be considered a positive result. The simulations typically last less than a couple of hours and a simulation time increased by a few minutes is not a big trade-off, especially considering that this tool avoids many waveform reviews.

	Memory	System CPU	User CPU	Sim Time
Framework Impact	+2.44%	+6.04%	+10.64%	+9.21%

Table 5.6: Average performance impact of the framework on the simulations metrics

5.2.2 Analog Tool Impact

As has been done for the checker library framework, the impact on the simulation performances is also measured and reported for the tool used to manage the force and measure of analog signals during the playback simulations. Also, for this part of the Master's Thesis project, ten simulations have been executed for each possible configuration to reduce the impact of outliers caused by the distributed nature of the simulation software. Since the tools allow applying any combination of force and measure operations during the simulations, three different simulation types will be analyzed:

- only force operations are done
- only measurements on analog signals are performed
- both force and measure are executed

Memory The results related to the memory usage presented in Section 5.2.1 are comparable with the ones obtained from this part of the project. The UVM libraries and the UVM test bench used for the force and measure of analog signals have a negligible impact on memory usage. As shown in the example reported in Figure 5.10, the increase in memory usage when using the UVM TB module is limited to a maximum of 100 Megabyte increase making the impact negligible to the user who is utilizing the tool.

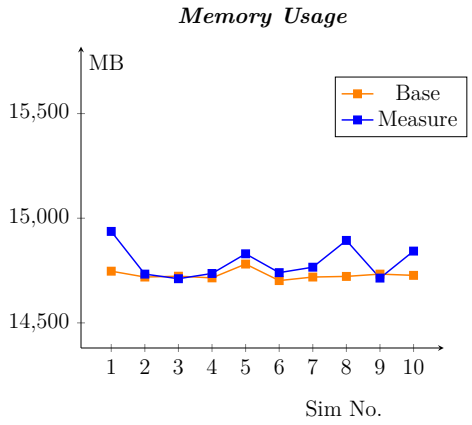

Figure 5.10: Memory usage of the base simulations and the ones supporting the analog tool

CPU Usage The impact of the analog tool on the simulation performances from the CPU usage perspective reflects the pattern described in the previous section. Indeed, the increase in user CPU usage is around 10%. The system CPU usage instead impacts slightly differently since it increases the time spent in system space between 5% and 15%. This metric is related to the time used by the CPU to execute instructions at the kernel level. This increase in time may be due to more complex UVM modules and UVM libraries to deal with analog signals, which may require significant system space usage. In the tables reported below, the user and system CPU usage behavior is reported when forcing analog signals.

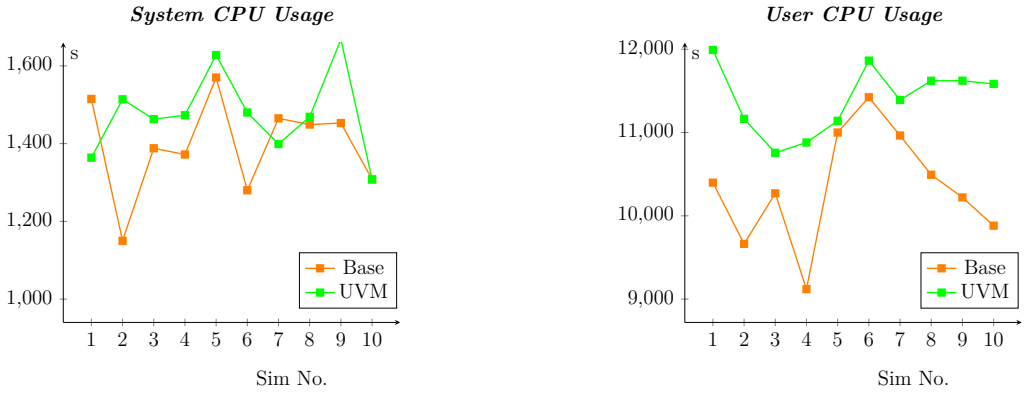

Figure 5.11: System and user CPU usage of during base and UVM-based simulations

Total Simulation Time As stated previously, the total simulation time is the most critical metric since it indicates if the engineer executing the simulations will experience a bad experience using the tool or if the increase in simulation time will not be noticeable. The charts of Figure 5.12 show that the simulation time impact of the tool is quite regular and is around a 10% of increase. Moreover, it shows that it does not depend on the type of analog operation performed during the simulation since the behavior is quite similar in all three cases. The increase in the simulation time is not entirely negligible. Indeed, the playback simulation requires many hours (or even days) to be completed, and even a 10% increase may implicate that the simulation could require even one hour more than if the UVM test bench was not present. This drawback is balanced by the fact that the tool allows doing operations that were not possible before and that the benefit of this solution could overcome the issue of having longer simulation times.

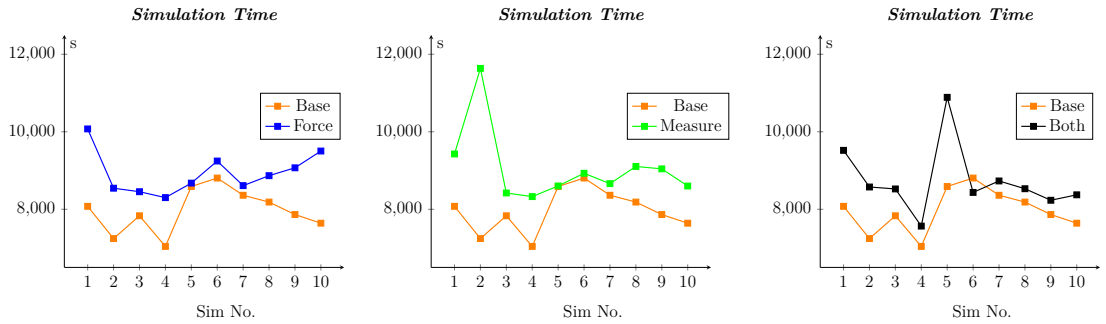

Figure 5.12: Simulation time when force, measure and both operations are executed

Summary This last paragraph aims to report the average impact of the tool on the simulation performances. The values of Table 5.7 give a glance of the influence of the UVM components on the simulations.

	Memory	System CPU	User CPU	Sim Time
Tool Impact	+0.86%	+10.27%	+9.93%	+11.90%

Table 5.7: Average performance impact of the analog tool on the simulations metrics

Chapter 6

Conclusion

This Master Thesis presents the processes that led to developing a UVM-based framework to enhance Apple's DFT DV flows. Specifically, the creation of an automated checker library for verifying signals' behavior during DV simulations has been presented. The developed framework provides a solution fully integrated into the pre-existing regression environment, which supports the combination of partition-level and chip-level simulations with DUTs defined as RTL and Netlists. Furthermore, the proposed solution is fully scalable since it does not limit the number of checkers applied to the chip's IPs.
The checker library framework could be extended to support the force and measure of analog signals during playback simulations. This can be done by developing an analog checker to be added to the library and a UVM driver, which can force the value of analog signals. A prototype has been designed to prove that the UVM-based approach was feasible for managing analog operations.

The data obtained by the profiler indicate the low impact of the framework on the simulation performances. The effects of the UVM test bench on the computer's memory are practically negligible since it increases the usage by 2% to 4%, which does not affect the usability of the regression environment from the user's point of view.
The impact of the frameworks on the total simulation time is considerable since it is increased on average by 9%. It must be highlighted that the typical simulation time without using the framework varies from a few minutes to more than an hour.

Consequently, the UVM TB module will slow the simulation time by just a few minutes. Furthermore, the increase in simulation time must then be compared with the time required to manually review the waveform of many IPs, which could require a lot of time and effort from the user.

Regarding future development, the main goal is to integrate the tool supporting the analog operations during the playback simulation into the checker library framework. Then, the tool must be made project independent, its triggering mechanism should be improved, and the python script that generates the UVM TB module must be integrated into the scripts of the checker library framework. Regarding the checker library, a possible refinement is to develop a system to allow the user to selectively activate or deactivate the error and warning messages generated by the UVM Scoreboard. This feature is helpful because once the engineer is notified about a possible event, it may want not to be advised during the following simulations. For example, an IP may not have all the signals a checker requires. Nevertheless, on the other hand, the user may be aware of the situation and may not want to be notified every time about this case.

Appendix A

Example structure.json File

```
{
  "name": "Project_X",
  "structure": {
    "partition_A": {
      "sub_partition_1": {
        "MC": {
          "cluster_mc_1": [{"mc_1": 1}],
          "cluster_mc_2": [
                  {"mc_20": 1},
                  {"mc_21": 1},
                  {"mc_22": 1}
          ],
          "cluster_mc_3": [{"mc_3": 4}]
        },
        "HardIp": {
          "cluster_hp_1": [{"hp_1": 1}],
          "cluster_hp_2": [{"hp_2": 1}],
          "cluster_hp_3": [{"hp_3": 1}]
        }
      }
    }
}}
```

Appendix B

Example cluster.json File

```
[
  {
    "name": "memory_1",
    "rtl_path": "DUT.chip.partition_A.sub_partition_1.cluster_mc_1.
   memory_1",
    "gls_path": "DUT.chip.\\partition_A/sub_partition_1./
   cluster_mc_1.memory_1 ",
    "pins": {
      "Address": {
        "properties": ["address"],
        "direction": "in",
        "verifOnly": false,
        "polarity": "unknown",
        "pins_number": 10
      },
      "Clock": {
        "properties": ["clk", "check_default"],
        "direction": "in",
        "verifOnly": false,
        "polarity": "high",
        "pins_number": 1
      },
      "Data": {
```

```
        "properties": ["data","check_default"],
        "direction": "in",
        "verifOnly": false,
        "polarity": "high",
        "pins_number": 128
      }
    }
]
```

Acronyms

ATE

Automated Test Equipment

ATP

Automatic Test Pattern

BIST

built-in self-test

BSR

boundary scan register

CSV

comma-separated values

DFT

Design For Testability

DUT

Device Under Test

DV

Design Verification

FSM

finite-state machine

HDL

Hardware Description Language

HTOL

High-temperature operating life

IC

Integrated Circuit

IP

Intellectual Property

JSON

JavaScript Object Notation

JTAG

Joint Test Action Group

NDA

non-disclosure agreement

OOP

object oriented programming

RTL

register-transfer level

SoC

System-on-Chip

STIL

Standard Test Interface Language

SV

SystemVerilog

TAP

Test Access Port

TB

test bench

UVM

Universal Verification Methodology

WSI

Wrapper Serial Input

WSO

Wrapper Serial Output

Bibliography

[1] L.T. Wang et al. *VLSI Test Principles and Architectures: Design for Testability*. Elsevier Science, 2006. ISBN: 9780123705976. URL: https://books.google.it/books?id=5NDAngEACAAJ.

[2] *About JTAG Technologies*. Feb. 2021. URL: https://www.jtag.com/about-jtag-technologies-2/.

[3] Embedded Staff. *Introduction to JTAG*. Oct. 2002. URL: https://www.embedded.com/introduction-to-jtag/.

[4] *JTAG Daisy-Chaining*. URL: https://onlinedocs.microchip.com/pr/GUID-DDB0017E-84E3-4E77-AAE9-7AC4290E5E8B-en-US-4/index.html?GUID-22309B26-88EF-4EC4-98F6-74C0BC5C80D5.

[5] Peng Zhang. «Industrial Control System Operation Routines». In: *Advanced Industrial Control Technology* (2010), pp. 735–745. DOI: 10.1016/b978-1-4377-7807-6.10018-x.

[6] «IEEE Standard for Verilog Hardware Description Language». In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[7] «IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language». In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[8] *UVM Testbench Architecture*. Mar. 2020. URL: https://verificationguide.com/uvm/uvm-testbench-architecture/.

[9] *UVM phases*. URL: https://www.chipverify.com/uvm/uvm-phases.