UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Automatic UVM-based Testbenches Generation, starting from FSM Specifications

Diploma Thesis

Eleni Dimkou

Supervisor: Nikolaos Mpellas

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Automatic UVM-based Testbenches Generation, starting from FSM Specifications

Diploma Thesis

Eleni Dimkou

Supervisor: Nikolaos Mpellas

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτόματη δημιουργία Testbenches από μηχανές πεπερασμένων καταστάσεων, βασισμένη στη UVM

Διπλωματική Εργασία

Ελένη Δήμκου

Επιβλέπων/πουσα: Νικόλαος Μπέλλας

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Nikolaos Mpellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

To begin with, I would like to thank my supervisor, Prof. Nikolaos Bellas, for his influence on my academic interests and career. In a similar manner, I would like to thank my managers Darko Vukicevic and Danilo Costa Oliveira, my mentor Paolo Capogreco for his time and support from start to finish of this project, as well as the rest of the Digital Verification team members of Infineon Technologies AG, in Villach, for their guidance.

Next, I would like to acknowledge the sacrifices, the encouragement, and the power my family gives me to continue and to put my best effort to fulfill my dreams.

I also would like to express my thanks to the rest of my friends, whose support, laughter, and presence have made this journey much more enjoyable.

Last but not least, I could not have persevered through the challenges of high school and my academic journey without the patience, support, and love of my partner Georgios Tsoulis and my best friend Vasileia Chantzara. For that, I would like to express my gratitude to them.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Eleni Dimkou

Diploma Thesis

Automatic UVM-based Testbenches Generation, starting from FSM Specifications

Eleni Dimkou

Abstract

The increasing complexity of mixed-signal Integrated Circuits (ICs) poses significant challenges for verification engineers, particularly in ensuring consistency and accuracy throughout the design verification process [1].

Finite State Machines (FSMs) provide a structured way to define system behavior, making them a natural choice for streamlining the verification process. This thesis introduces an FSM-based methodology to automate the generation of UVM-compliant verification components, such as reference models, monitors, and sequence items. The core idea lies in adopting FSMs as the main source of information, ensuring consistency between design specifications and verification frameworks.

Validating the proposed methodology involved testing FSMs of varying sizes and complexities, including a real-world example. The results showed that the generated verification components correctly validated functional designs and reliably identified defects in faulty ones, highlighting the robustness and effectiveness of the approach.

In summary, this work contributes a scalable and efficient solution to the challenges of mixed-signal verification by reducing manual effort, simplifying the workflow, and enhancing the efficiency of the verification process.

Keywords:

Mixed Signal Verification, Universal Verification Methodology, Finite State Machine

Διπλωματική Εργασία

Αυτόματη δημιουργία Testbenches από μηχανές πεπερασμένων καταστάσεων, βασισμένη στη UVM

Ελένη Δήμκου

Περίληψη

Η αυξανόμενη πολυπλοκότητα των Ολοκληρωμένων Κυκλωμάτων (ICs) μεικτού σήματος θέτει σημαντικές προκλήσεις για τους μηχανικούς επαλήθευσης, ιδιαίτερα όσον αφορά τη διασφάλιση της συνέπειας και της ακρίβειας σε όλη τη διαδικασία επαλήθευσης τους [1].

Οι Μηχανές Πεπερασμένων Καταστάσεων (Finite State Machines - FSMs) παρέχουν μια δομημένη μέθοδο για τον ορισμό της συμπεριφοράς ενός συστήματος, καθιστώντας τες ιδανική επιλογή για την απλοποίηση της διαδικασίας επαλήθευσης. Αυτή η διπλωματική εργασία παρουσιάζει μια μεθοδολογία βασισμένη σε FSM για την αυτοματοποίηση της δημιουργίας συνιστωσών επαλήθευσης συμβατών με το UVM, όπως μοντέλα αναφοράς, monitors και sequence items. Η βασική ιδέα έγκειται στην υιοθέτηση των FSMs ως την κύρια πηγή πληροφορίας, εξασφαλίζοντας συνέπεια μεταξύ των προδιαγραφών του σχεδιασμού και επαλήθευσης.

Η επικύρωση της προτεινόμενης μεθοδολογίας πραγματοποιήθηκε μέσω δοκιμών FSMs διάφορων μεγεθών και επιπέδων πολυπλοκότητας, συμπεριλαμβανομένου ενός πραγματικού παραδείγματος. Τα αποτελέσματα έδειξαν ότι οι παραγόμενες συνιστώσες επαλήθευσης επικύρωσαν σωστά τα λειτουργικά σχέδια και εντόπισαν τα ελαττώματα σε προβληματικά σχέδια, αναδεικνύοντας την αποτελεσματικότητα της προτεινόμενης προσέγγισης.

Συνοπτικά, η παρόν εργασία συμβάλλει με μια επεκτάσιμη και αποδοτική λύση στις προκλήσεις της επαλήθευσης μεικτού σήματος, μειώνοντας τη χειροκίνητη εργασία, απλοποιώντας τη διαδικασία και ενισχύοντας την αποτελεσματικότητα της διαδικασίας επαλήθευσης.

Λέξεις-κλειδιά:

Mixed Signal Verification, Universal Verification Methodology, Μηχανές Πεπερασμένων καταστάσεων

Table of contents

List of figures

Abbreviations

VE	Verification Environment
FSM	Finite State Machine
iVC	interface Verification Component
mVC	module Verification Component
SV	SystemVerilog
UVM	Universal Verification Methodology
AMS	Analog Mixed Signals
IP	Intellectual Property
DUT	Design Under Test
IC	Integrated Circuit
PLL	Phase Locked Loop
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
BFM	Bus Functional Model

Chapter 1

Introduction

1.1 Motivation

1.1.1 Mixed Signal Integration

The design of integrated circuits has historically been divided between analog and digital approaches, each with distinct advantages and limitations. Analog circuits, while capable of direct signal processing, are inherently susceptible to noise and distortion. Digital circuits, by contrast, offer superior noise immunity but require significantly higher bandwidth to transmit equivalent information. These fundamental trade-offs have traditionally dictated the choice between analog or digital implementation based on application requirements [7].

The modern electronics industry demands low-cost, highly integrated applications such as computers, automotive, wireless, and more. Consequently, innovative advancements made possible the production of analog and Radio Frequency circuits on a very small scale (up to 45nm). These advancements and implementations have transformed System-on-Chip (SoC) devices, which were made of mostly digital circuitry, to now contain significant analog and mixed-signal content. Additionally, analog blocks now include significant amounts of digital logic either to increase functionality or to assist the analog portions in achieving target performance [1].

Increasing SoC integration requires more complex mixed-signal IP blocks, many of which are as complex as traditional ICs. **But what is an Analog Mixed-Signal (AMS) block or AMS Intellectual Property (IP)?** An AMS IP is defined as a design that contains both analog circuits and digital gates, requiring a combination of analog and digital methodologies. Today,

this implementation continues to grow due to the increased needs for mobility, higher performance, and integration of interfaces. Examples of IPs with mixed-signal blocks include: Phase-Locked Loops (PLLs), Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs), High-speed Input/Output (I/O) interfaces, RF transceivers, Memory interfaces, and others [1].

1.1.2 Challenges in digital verification and design

The implementation of AMS blocks presents both significant opportunities and substantial challenges. Although they enable more powerful and integrated designs, they also introduce significant verification complexities. Some of the notable challenges are as follows:

- High complexity of the design. The integration of analog and digital components within the same system creates unique challenges, as these domains often rely on different design methodologies and tools [8].

- Mixed-signal IC verification inherently exceeds the complexity of purely digital or analog IC testing, as it requires testing in both analog and digital parts. This dual domain requirement often requires specialized test methodologies to ensure a comprehensive functional verification of all components. However, this increased complexity frequently introduces accuracy issues that must be carefully addressed [9], [1].

- Behavioral modeling is essential for mixed-signal verification, as it enables efficient simulation of AMS blocks by abstracting their behavior at a higher level. Its effective implementation requires a well-defined scope and purpose for the model, precise architecture selection, and expertise in analog / mixed signal design, coding, debugging, and simulation algorithms [1].

- The lack of clear and reliable specifications is one of the most critical challenges. Specifications often rely on documents that under specify or omit critical details about the design behavior in certain scenarios, leaving engineers with vague or ambiguous cases [10].Developing a reliable verification environment from insufficient specification is a painful task, as designers and verification engineers frequently encounter test failures caused by undocumented or undefined behavior.

These are just a few of the key challenges that need to be carefully addressed through improved methodologies and tools. By introducing innovations to simplify the work of engineers, these complexities can be minimized, enabling more efficient and accurate verification processes.

1.2 Thesis

Verification of mixed-signal IPs faces ongoing challenges, many of which arise due to unclear or inconsistent design specifications. These challenges not only delay development cycles, but also compromise the quality and reliability of the final design. In current workflows, critical requirements are often scattered across multiple sources (e-mails, informal discussions, tools), leading to inconsistencies and misinterpretations. This approach usually leads to confusion and errors, especially when engineers have to manually interpret and implement these evolving specifications into verification models. The situation becomes even more difficult when late-stage design changes require rushed updates, often resulting in additional rework, longer development times, and in the worst cases, preventable failures that were missed in the final silicon.

This thesis aims to address these issues by rethinking how design specifications are translated into verification models. By using Finite State Machines (FSMs) as the central source of truth for IP functionality, we can mitigate the risks associated with traditional, manual processes. Automating the generation of UVM-compliant reference models from FSMs eliminates manual coding tasks and reduces the potential for human-induced errors, and ensuring greater accuracy and efficiency in the verification process. Additionally, when specifications change, this FSM-based approach allows for quick and seamless updates, bypassing the inefficiencies of manual workflows.

By automating a process that is usually time-consuming and prone to mistakes, engineers can focus their efforts on uncovering corner-case defects or analyzing complex analog-digital interactions. This is especially critical for mixed-signal IPs, such as data converters or clocking systems, which inherently involve complex failure modes and can be hard to detect. The approach presented in this thesis creates a streamlined path from precise specifications to reliable verification, setting a higher standard for trustworthy and efficient mixed-signal design.

1.3 Thesis Contributions

The contributions of this thesis can be summarized as follows:

- An in-depth analysis was conducted on the common issues in mixed-signal IP verification workflows, including specification fragmentation, manual errors during model creation, and inefficiencies in adapting to specification changes.
- A systematic methodology was proposed to use Finite State Machines (FSMs) as the main source of defining IP's functionality.
- A methodology was implemented to automatically generate reference models in SystemVerilog using Universal Verification Methodology (UVM). This process eliminates manual coding tasks, reducing human-induced errors and saving engineering time.
- The workflow was evaluated using multiple designs, including intentionally faulty ones to test its error detection capabilities. The results confirmed that the automated workflow successfully identified issues in defective designs while accurately validating the functionality of the correct designs.

1.4 Outline

This thesis is structured into the following chapters.

- Chapter 2 provides the necessary background, presents the SystemVerilog fundamentals, details the UVM methodology, and explains FSM basics.
- Chapter 3 describes the architecture of the generated verification components and details the development process.
- Chapter 4 discusses the testing and validation process of the proposed FSM-based methodology.
- Chapter 5 concludes the thesis and points to future improvements in the methodology.

Chapter 2

Theory

2.1 Intoduction

In this chapter, you will be introduced to the theoretical foundations and key concepts of the languages and methodologies essential to this thesis. To develop a tool capable of building effective verification components that can be easily adapted to different scalable environments, a thorough understanding of the following components was required:

- **SystemVerilog:** The hardware description and verification language (HDVL) used for modeling and testing digital systems.

- **Universal Verification Methodology (UVM):** The industry standard framework for building modular, reusable test benches.

- **Finite State Machine (FSM):** The mathematical model that can form the design and verification of sequential logic.

The study of these elements was critical to achieving the thesis' goal: creating a UVM-based verification framework to validate FSM-driven designs against their specifications. Below, we detail their theoretical principles, relevance, and application in this work

2.2 SystemVerilog

2.2.1 What is SystemVerilog?

The journey of hardware design and verification languages began around in the mid-1980s with the introduction of the Verilog language. By 1995, Verilog, developed by Phil Moorby at Gateway Design Automation, had gained recognition as an IEEE standard. Initially, Verilog aligned more with hardware design but contained sufficient behavioral elements for testbench development. In the late 1980s, designs ranged from 10,000 to 100,000 gates, involving substantial gate-level verification and some Register Transfer Level (RTL) verification. At that stage, Verilog's capabilities adequately addressed design and verification needs [11]. The Verilog-2001 update was significant, introducing features like multi-dimensional arrays, auto variables, and enhanced design constructs. As designs progressed to multi-million gates, it became clear that Verilog required substantial improvements in functional verification capabilities. The limitations in advanced verification constructs within Verilog led to the emergence of other languages like Vera and "e", supplementing Verilog with Object-Oriented Programming (OOP) subsets. However, this necessitated a multi-language environment supported by various EDA vendors, which proved complex, and time-consuming [11].

SystemVerilog 3.0, introduced in June 2002, marked a significant advancement extending the capabilities of traditional Verilog HDL to address the growing complexity of modern digital systems. Developed through Accellera Systems Initiative and later standardized by IEEE (IEEE 1800), SystemVerilog serves as both a design modeling and verification language, addressing the growing complexity of digital systems [11]. What makes SystemVerilog particularly valuable is its dual nature - it maintains backward compatibility with Verilog for design while adding powerful constructs specifically for verification [11].

The evolution continued with SystemVerilog 3.1a in May 2004 by Accellera, working closely with major EDA companies [11].

This release introduced some general-purpose object-oriented programming features such as [11], [12]:

- SystemVerilog classes, enabling the creation of complex, reusable components.
- Data types, including various built-in and user-defined types.

- Checkers, which can be used to monitor and report design behavior.
- Interfaces, providing structured connections between design and verification components.
- Processes and procedures, which define concurrent and sequential behavior.
- Methods, enabling encapsulation of behavior within classes.
- Specific data structures such as arrays, queues, structures, unions, and packages, which are crucial for managing verification data and they are not applicable using Verilog.

2.2.2 Why is SystemVerilog a good choice for verification?

SystemVerilog provides features specifically designed to support the verification of digital hardware. These are integrated into the core language because they are difficult to implement efficiently as mere library functions. Some of the language's key features that stand out are:

Random Stimuli: Provides the ability to give random input to the signals of the design under test. This feature has changed the verification approach by examining some edge cases that might otherwise be missed in more depth.

Constrained Random Generation: Constrained randomization allows engineers to automatically generate complex test scenarios within specific ranges for the parameters, significantly improving test coverage. It also prevents the test from covering areas of the design that were not originally intended [2].

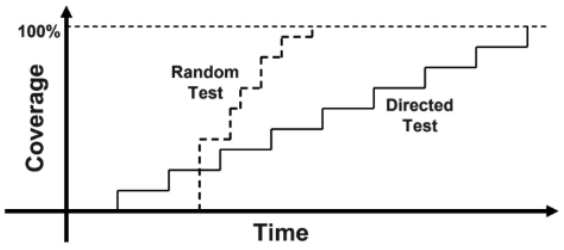

Figure 2.1: Random tests cover a wider scope compared to direct tests [2].

Assertions: Enable the means to formally specify temporal relationships and expected behaviors. While providing the capability to continuously monitor design behavior over time by automatically triggering violations of expected properties.

Functional Coverage: Through constructs like covergroups and coverpoints, verification engineers can measure which parts of the design have been tested, providing objective metrics for the completeness of the verification. It helps verification engineers assess the thoroughness of their verification efforts and identify untested areas. [11]

Figure 2.2: Functional Coverage [2].

While SystemVerilog provides these powerful features we discussed earlier, engineers develop inconsistent verification approaches and the verification environments become hard to maintain, reuse, and debug. Eventually, this challenge between SystemVerilog's capabilities and real-world application led to the creation of standardized verification methodologies.

2.3 Universal Verification Methodology

Pre-silicon digital design verification process represents one of the most critical and time consuming phases during the implementation of the design. Verification engineers must verify all possible functional scenarios while meeting the project's timelines. Although SystemVerilog provides the necessary constructs for this process through its assertion and coverage capabilities, its features alone have been proven insufficient for large-scale verification. This limitation resulted to the development of the Universal Verification Methodology (UVM).

2.3.1 Timeline

Following the timeline of how UVM was invented, starting in 2000, Verisity Design (now part of Cadence Design Systems, Inc.) introduced the Verification Advisor (vAdvisor), a collection of best practices aimed at the 'e' user community. Furthermore, vAdvisor provided practical guidance for stimulus generation, coverage model creation, and self-checking testbenches [13].

In 2002, Verisity launched the first verification library, the e Reuse Methodology (eRM), which quickly gained traction. It defined packaging guidelines, architecture for environments, agents, and monitors, as well as sequences, messaging, and more that strongly influenced modern methodologies like OVM. Later, eRM was extended to support module-to-system reuse, add system and software verification components, and include the first commercial register package, vr_ad [13].

In 2003, Synopsys responded with the Reuse Verification Methodology (RVM) for the Vera language. RVM included base classes, messaging, and packaging guidelines, but it lacked architecture guidelines, sequences, and objection handling, making it a subset of eRM in the eyes of many. Its key innovation was the callback mechanism, adapted from software design to enhance procedural extensions in Vera and other object-oriented languages. RVM was later converted to the SystemVerilog (SV) Verification Methodology Manual (VMM), a proprietary library developed by Synopsys to support the evolving SystemVerilog standard [13].

Mentor Graphics's Advanced Verification Methodology (AVM) arrived in 2006. Based primarily on the OSCI SystemC Transaction-Level Modeling (TLM) standard, it addressed low-level communication but left gaps in higher-level features such as test classes, complex stimulus generation, and factory-based configuration. However, AVM was a milestone as the first open-source verification solution [13].

In 2005, Cadence acquired Verisity and began adapting eRM to SystemVerilog. The result, launched in 2007, was Universal Reuse Methodology (URM), an open-source framework that uses TLM and brings proven eRM concepts to the SystemVerilog community. URM also introduced new features such as an abstract factory, configuration mechanisms, test classes, and class automation (copy, print, compare, and more) [13].

In early 2008, Cadence and Mentor merged URM and AVM to create the Open Verification Methodology (OVM). Since both already used TLM, the integration was seamless. OVM adopted the high-level methodology of URM while adding multi-vendor simulator compatibility, a critical advantage at a time when the implementation of SystemVerilog varied between tools. As a result, many companies adopted OVM as their verification methodology, and Cadence extended it to support multi-language testbenches in e, SystemVerilog, and System [13].

By 2010, OVM 2.1.1 was selected as the foundation for the Universal Verification Method-

ology (UVM), an emerging Accellera standard tested by all major vendors. With UVM, the industry moved past OVM–VMM comparisons and overcame reuse barriers. The portal was launched to serve both UVM and OVM communities, supporting the ongoing evolution of features to meet future verification needs [13].

2.3.2 Fundamentals about UVM

The Universal Verification Methodology (UVM) is a comprehensive, standardized approach that combines the most effective practices for efficient verification. One of the core principles of UVM is the creation and reuse of UVM Verification Components (UVCs), reusable elements that form the foundation of a structured and efficient verification environment. In addition, UVM is open-source as an Accellera standard (with IEEE standardization in progress), and it is tested on all major commercial simulators for compatibility and portability, making it equally suitable for small-scale designs as well as complex, IP-based, high–gate-count system-on-chip (SoC) architectures.

From a technical perspective, UVM delivers a consistent object-oriented usage model for all UVCs, ensuring seamless interoperability regardless of their source or implementation language [13]. The primary capabilities of UVM are:

- **Data Design:** Enables a clean separation of data and components within the verification environment. Its built-in utilities simplify the printing and visualization of objects, hierarchical data access, and routine operations (such as copying, comparing, and packing). This allows engineers to focus on the content and behavior of objects rather than low-level support code [13].

- **Stimulus Generation:** Provides classes and infrastructure for precise control of sequential data flows at both module and system levels. Supports randomization of data based on the current environment state, including DUT status, interfaces, and previously generated transactions, and allows customization with user-defined hierarchical transactions [13].

- **Building and Running the Testbench**: Streamlines testbench creation for complex SoCs by using base classes that automate and structure the build process. Supports hierarchical, reusable environments and includes a standard configuration interface for

customizing runtime behavior and testbench topology without modifying the source code [13].

- **Coverage Model Design and Checking Strategies:** Supports best practices for functional coverage integration, along with physical, temporal, protocol, and data checks, all within reusable UVCs [13].

Additional features, including the use of factory, macros, and the `uvm_phases` mechanism, will be examined in the following subsections.

2.3.3 Factory

The Universal Verification Methodology (UVM) Factory is a core utility within the UVM Class Library that enables the dynamic construction and management of verification objects.

It provides a mechanism to simplify the development of adaptable objects and components, enhancing re-usability across testbenches. More importantly, it enables direct substitution of components without requiring engineers to create new classes for each different component [5]. Using the built-in UVM factory implementation, verification engineers avoid the overhead of building a custom factory or embedding factory methods in class definitions [14],[5].

2.3.4 Macros

Another advantage of the UVM methodology lies in its comprehensive macro system. The UVM library incorporates an extensive set of predefined macros specifically designed to minimize coding overhead by enabling combined definitions (e.g., fields, tasks) in single statements. Although optional, these macros enable faster development and easier maintenance through built-in, standardized implementations of common verification tasks like object registration in the factory using utility macros (uvm*utils*, *callbacks, constants, etc.), sequence generation (`uvm_do`) and field operations (`uvm_field_`) [5],[15].

It is important to note that the use of UVM macros may impact performance differently between tool vendors, make debugging more challenging, and provide less control over execution order compared to hand-written implementations. Verification engineers must carefully evaluate whether the project should prioritize development efficiency and code consis-

tency (using macros) or require fine-grained control and optimization (hand-coded solutions) [5].

2.3.5 UVM Phases

The UVM execution model follows a strictly ordered sequence of phases to ensure proper testbench initialization and operation, serving as a synchronizing mechanism throughout the simulation lifecycle.

In more detail, each component follows a pre-defined set of phases, and it cannot proceed to the next phase until all components finish their execution in the current phase.

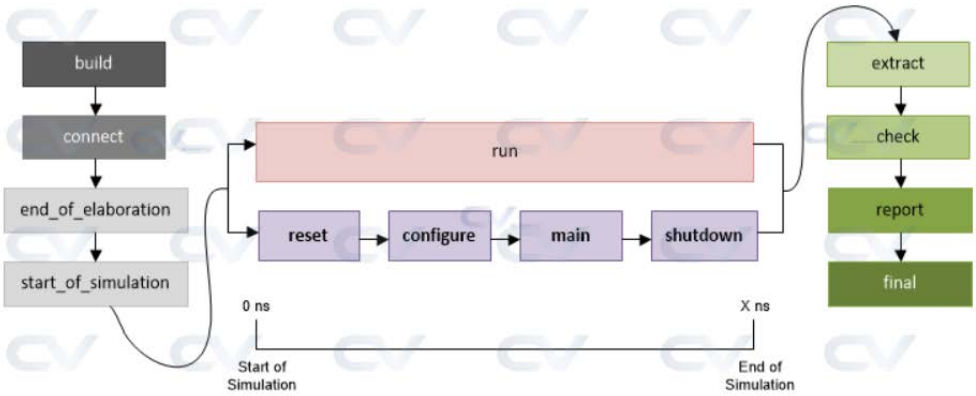

Figure 2.3: UVM Phases Diagram [3].

The initial phase is the `build_phase` which constructs all the components, objects hierarchically, establishing the complete testbench structure, and this deliberate prevents connection attempts to components during their construction phase [3].

Following successful instantiation of all the components, the `connect_phase` establishes all the communication channels and references between the components. Subsequent phases (`end_of_elaboration_phase` and `start_of_simulation_phase`) serve primarily for structural verification and diagnostic reporting of the assembled testbench hierarchy [3].

The primary execution phase, termed `run_phase`, is where the actual test stimulus is generated. This phase consumes simulation time while operating concurrently with other UVM run-time phases [3].

The final phase is the clean phase, which begins with data extraction and computation of the expected results (`extract_phase`). It then performs error-checking tasks between

expected and actual design values (`check_phase`). The subsequent `report_phase` displays the results from the checkers or provides a summary of the test objectives. Finally, the `final_phase` executes last-minute operations before simulation termination. [3]

2.3.6 Verification Component Overview

UVM provides a set of base classes from which more complex classes can be built by inheritance and adding onto them any methods that are required for verification environment.

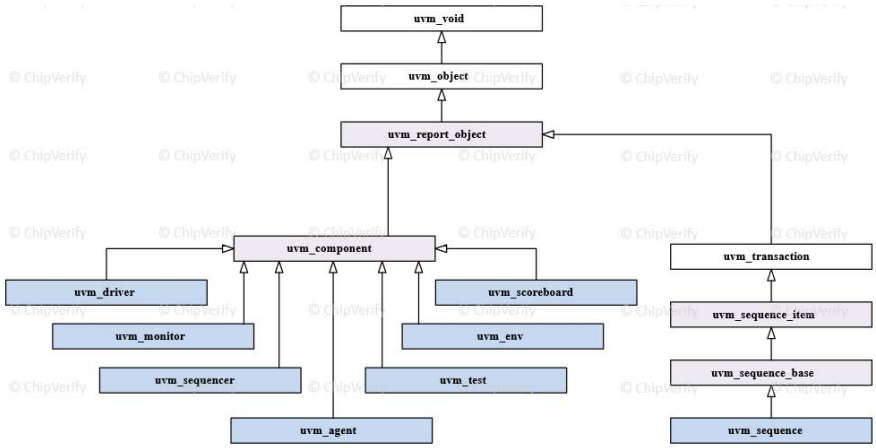

Figure 2.4: UVM Hierarchy [4].

As shown in the picture 2.4, the base class for almost everything in UVM is the uvm_object class. It provides a set of common utility functions such as `print(), copy(), compare(),` and `record()` that will be inherited by any other class in a UVM testbench. This prevents developers from repeatedly rewriting these common functions, and from creating configuration settings for parameters and components [4].

The `uvm_component` class serves as the foundational base class for all UVM components. Building upon the core functionality inherited from `uvm_object` and `uvm_report_object`, the `uvm_component` base class introduces several key interfaces essential for a structured verification environment:

- **Hierarchy Management:** Enables component hierarchy traversal and search operations [16].
- **Phasing:** Defines a standardized phased execution flow, including built-in phase methods (e.g., build_phase, run_phase) [16].

- **Configuration Methods:** Facilitates pre-construction and runtime configuration of component topology and parameters [16].

- **Reporting:** The `uvm_report_object` provides an interface to the UVM reporting facility. Components use this interface to access the reporting system, where all simulation messages (e.g. warnings, errors, and debug information) are generated and can be handled [16].

- **Transaction Recording:** Supports transaction recording through dedicated methods, allowing components to record activity to tool-specific databases for analysis and debugging [16].

- **Factory Integration:** Offers simplified access to the uvm_factory for dynamic object and component creation, supporting type and instance overrides [16].

Having established the foundational UVM classes, in the following subsections, we will examine the fundamental architecture of a testbench using a top-down approach.

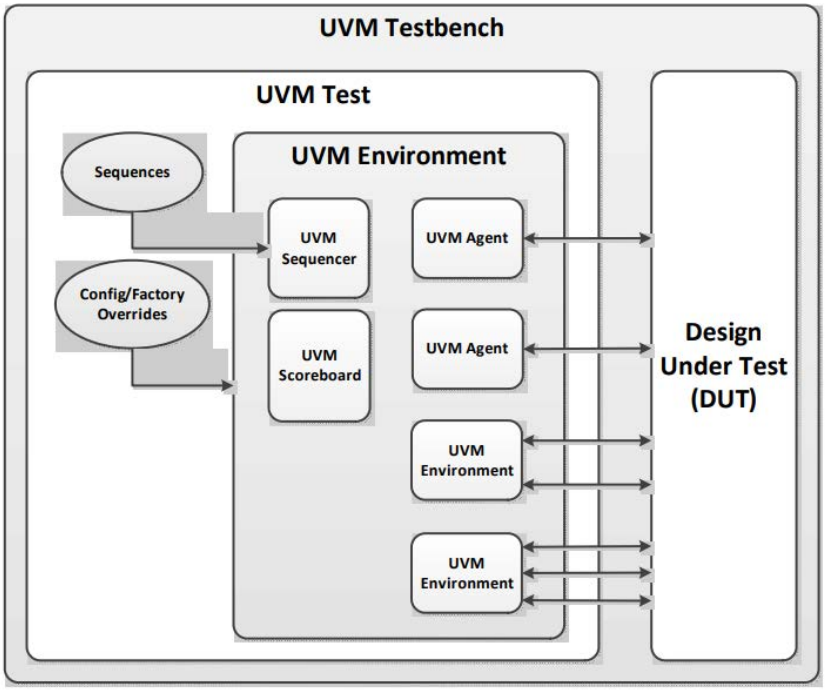

Figure 2.5: Typical UVM Testbench Architecture [5].

Testbench Top

In the top-level module known as testbench, the Design under Test (DUT) module, and the UVM Test class, interfaces, and the rest of the verification module-based components are instantiated [5]. It is a static container that holds everything that is required to be simulated and becomes the root node in the hierarchy [17].

Interface

An interface provides a structured approach to encapsulating signals in a single block, simplifying the management of the connection between the DUT and the verification components. By grouping related signals together within an interface block, the same interface becomes reusable across multiple projects [18].

Environment

The UVM Environment contains and groups together other reusable verification components and also defines their default configuration as required by the application. Some of the components which are part of the environment are UVM Agents, UVM Scoreboards, or even other UVM Environments [5].

Agent

A UVM Agent is a structural component that encapsulates and connects other verification elements using TLM interfaces. An agent also supports two configuration options: Active or Passive.

An active agent instantiates three key components: a sequencer (which is a stimulus generator that manages transaction items that are provided to the driver for execution), at least one driver (responsible for converting received transactions into DUT signals), and a monitor (for observing interface activity).

Conversely, for scenarios which require only signal observation or coverage collection (without DUT stimulation), the agent operates in a passive mode. This configuration includes only monitoring components like the monitor, checkers, etc, which also matches our implementation's approach [19].

Monitor

A UVM monitor is a passive component that observes DUT signals without driving them. Its primary role involves collecting transactions by extracting signal information from the bus and converting them into transaction objects that can be used throughout other components. The monitor detects when new transactions occur, structures the data appropriately, and notifies other components by emitting events. It performs protocol and data checking and coverage collection. Monitors may also generate trace information for debugging [19].

Scoreboard

The main function of the UVM Scoreboard is to verify the behavior of a certain design. The UVM Scoreboard usually receives transaction-level objects (carrying inputs and outputs) from an interface of the DUT through TLM Analysis Ports. It processes input transactions through a reference model (also known as the predictor which mimics the functionality of the design) to produce expected transactions and then compares these expected outputs with the actual outputs from the DUT [5].

2.3.7 Summary

In this chapter, we presented the core aspects of the Universal Verification Methodology (UVM) and its important role in modern verification flows. Focusing on block-level verification, we provided an overview of how UVM's factory pattern, macros and phased execution proved to be game changers. Finally, we summarized the fundamental UVM components and their hierarchical relationships and their specialized roles within a verification environment. Together, these features make UVM effective by providing verification engineers with a solid framework to build verification environments that are efficient, scalable, and easy to adapt.

2.4 FSM

As computer technology advances, systems are becoming larger to address more sophisticated demands, making reliability harder to ensure. Consequently, testing has become a crucial part of the design and implementation process, and yet it has proved to be a significant challenge, particularly for modern digital systems [20].

Since the goal of this master thesis is to automatically generate verification components that replicate some of the functionality of the intended IP (the reference model of the DUT), we thought about modeling the core behavior of the digital circuits using finite state machines (FSM). This approach not only ensures accurate verification of the DUT's operation but also provides deeper insights into its behavioral characteristics.

2.4.1 What is an FSM?

A finite-state machine (FSM) is a mathematical model of computation, and it is used for modeling and designing sequential circuits. This abstract machine can be in exactly one of a finite number of possible states at any given time [21]. The FSM's behavior is defined by the initial state, its current state, and the transition between states that are triggered by specific inputs. For each state, there is exactly one transition to another state that is triggered by a specific input.

There are two types of finite state machines: Mealy and Moore machines. The theory is very similar for the two types. An FSM is called a Moore machine when the output state depends on the current state of the machine, and the input can only affect its next state. On the other hand, the output of the Mealy machines can be affected by the inputs and the current state.

Moreover, an FSM (Finite State Machine) can be represented using a state transition diagram, a type of directed graph where the nodes correspond to the machine's states and the edges / arrows represent transitions between them. Each transition is typically labeled with the input that triggers it. If it is a Mealy machine, the corresponding output is also labeled on the transition [20].

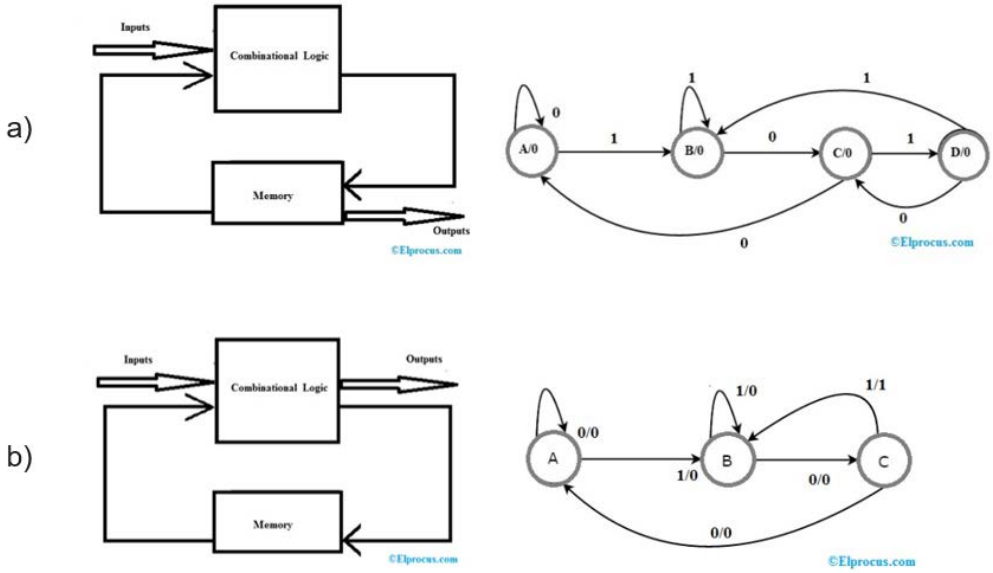

Figure 2.6: Example block and state diagram of Moore and Mealy FSMs [6].

State diagrams can be designed and described using various tools, where one of the most widely used standard is the UML (Unified Modeling Language) [22]. For this thesis, we decided to use PlantUML, a format that offers several advantages, including rapid definition through simple plain text, easy visualization of states and transitions without manual drawing, integration with documentation platforms, and support for multiple output formats [23].

2.4.2 Why Finite State Machines are fundamental to this Thesis

The study of finite state machines is essential for this thesis because it addresses our core verification challenge: a digital verification engineer receives an FSM specification (with its transition diagram), but they must verify a black-box implementation where only input/output behavior is observable. Since internal state transitions cannot be directly monitored, this work dynamically generates UVM-based components to verify whether the implementation's behavior matches the FSM specification [20].

Chapter 3

Implementation

3.1 Implementation Overview

The implementation of this thesis is centered on the automatic generation of two key verification components: the interface verification component, and the module verification component. Each of these components consists of structured elements such as agents, environments, monitors, scoreboard, and reference model, and they are generated based on a user-provided configuration. The most critical source of information for the generation process is the PlantUML file, which encodes the behavior of a finite state machine (FSM) and serves as the primary input for producing the reference model of the design under test (DUT).

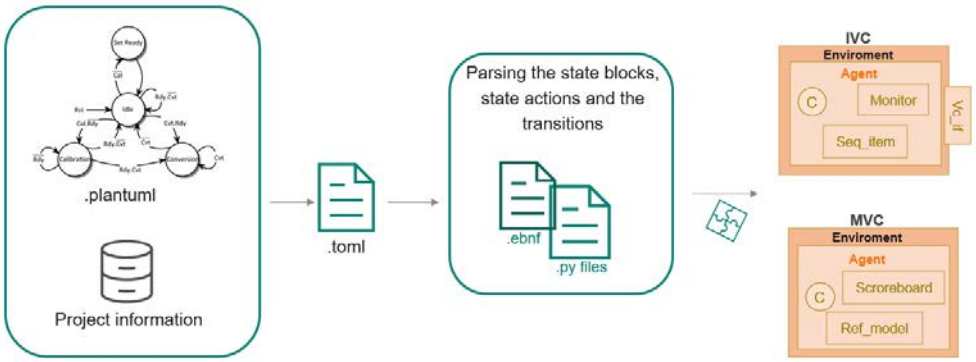

Figure 3.1: Implementation Overview

3.2 Input Specification and FSM Processing

3.2.1 User Input and Configuration

The first step in the generation process involves collecting the input data required for generating both the Interface Verification Component (IVC) and the Module Verification Component (MVC). As part of the setup, the user provides the required information in a TOML configuration file. This file includes project related parameters such as the project name, the path to FSM described in PlantUML format, and other fields essential for guiding the automated generation process.

```toml
# Specify the informations of the FSM
[fsm]
project_name = "example"
check_fsm_consistency = false

# plantuml_file = specify the full path to the plant uml file
plantuml_file = "edimkou/.../inputs/example.plantuml"

# output_dir = specify the full path where the generated components will be placed
output_dir = ""

# state_enums = specify the enum name
state_enums = "test_0_state_t"

[fsm.ref_class]
# if exists, specify the reset signal and if the reset is enabled the predicted state to a default state
reset_signal = "rst"
default_state = "S0"

# ************* Analysis Ports *************
[[fsm.ref_class.analysis_ports]] # nested array of tables
name = "example_ap1"
arg_type = "example_seq_item"
identifier = "example_aimp"

# ************* Covergroups *************
[[fsm.ref_class.covergroup]]
covergroup_identifier = "example_cg"
coverpoints = [
        {coverpoint_label = "NUM_OF_CALS", coverpoint = "Numbe_of_Cycles"},
        {coverpoint_label = "NUM_OF_SAMPLE_CLOCKS", coverpoint = "zero_clocks" },
        ]
covercrosses = [
    {cross_point = "example_CROSS", coverpoints = "NUM_OF_CALS, NUM_OF_SAMPLE_CLOCKS"}
    ]
```

Figure 3.2: Configuration file example

3.2.2 FSM Parsing and Validation

Once the input has been provided, the next technical challenge is to parse the user-provided FSM. A detailed analysis of the PlantUML specification was performed to identify a subset of features that the implementation would support. These include multiple state declarations, inline comments, basic and directional transitions, and input-driven transitions. To formally define this subset, an Extended Backus-Naur Form (EBNF) grammar was de-

veloped to capture the syntax of the supported PlantUML structures, and recognize the legal tokens.

To interpret the PlantUML file, a dedicated parser class was implemented to convert the FSM into structured data. The parser instantiates a field named fsm, which stores key information such as the state graph (using a networkx.DiGraph structure), the initial and final states, and transition signals. Furthermore, the supported transition inputs include functions, accessing fields, events, integers, and real numbers, as well as their logical combinations via Boolean expressions.

```
@startuml example

skinparam state {
    BackgroundColor LightBlue
    BorderColor Black
    FontColor Black
}

STATE S0
STATE S1
STATE S2
STATE S3


[*] --> S0 : Reset (rst)

S0 --> S1 : in0 == 10
S0 --> S2 : in1 == 1 || in2 != 2
S0 --> S0

S1 --> S3 : (insert_num(2) && rst == 0) || ( in3 && rst == 0)
S1 --> S2 : in0 && in1 == 1
S1 --> S1

S2 --> S1 : in4 == 10
S2 --> S0 : chl.idle_cycles = 7

S3 --> S0 : rst
S3 --> S3 : chl.continue()
S3 --> [*]

@enduml
```

Figure 3.3: Basic PlantUML code example

To ensure the accuracy and reliability of the generated reference model, the user has the option to enable a set of consistency checks to validate the structure of the FSM. These checks are applied after parsing and before any code generation. Some of the validations are to detect an empty FSM, and the existence of transitions of a specific state that have the same inputs and lead to different states, which may result in conflicting behavior. Moreover, the implementation verifies that all states are properly connected, identifying isolated states that are not reachable or do not link to any other state. These consistency checks contribute to the reliability of the tool by ensuring that only valid and well-defined FSMs are used as input for

the generation.

3.3 SystemVerilog Code Generation

After the extraction of the FSM model is complete, the necessary components must be built to ensure that the verification environment will reflect the behavior of the FSM. We created tools to generate language constructs, such as if-else blocks, class definitions, and functions. These have been hierarchically combined to create each of the necessary components.

3.3.1 If-Else statement Builder

The If-Else statement Builder is a fundamental component of the SystemVerilog code generation process. It provides a way to generate if-else statements with a given condition, body, and else clause. It also allows for the generation of nested if-else statements and conditional logic.

3.3.2 Function Builder

The Function Builder is responsible for generating SystemVerilog functions. This includes defining the function signatures, bodies, and any necessary logic. Additionally, it allows for the generation of functions with specific attributes, such as virtual functions, functions that call the superclass, and functions with specific return types. This flexibility enables the creation of a wide range of functions that can be used to implement complex behavior in a verification environment.

3.3.3 Class Builder

The Class Builder is responsible for generating SystemVerilog classes that represent a verification component. This includes defining the class structure, methods, properties, inheritance, and configuration. It also allows for the generation of the constructor function, which is used to initialize the class. Additionally, the Class Builder can generate the *build* and *connect* UVM functions, depending on the user's selection. These functions are used to construct and connect the verification component, respectively.

In addition, the Class Builder allows for the declaration of class fields, which can be used to store data and configuration options. Furthermore, it can generate covergroups, which are used to specify the coverage of the verification component. Covergroups can include coverpoints, coverpoint bins, and covercrosses, which specify the coverage of specific signals and conditions. Finally, the Class Builder uses the Function and If-else builders to generate the necessary methods and conditional logic for the class.

3.4 Generated components

The generated components are the result of the SystemVerilog code generation process. These components are integrated in a higher-level verification environment.

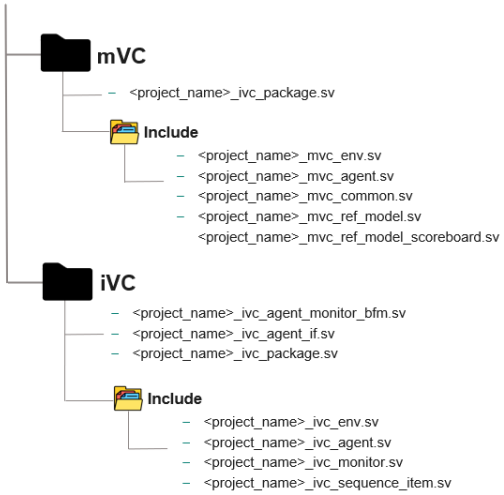

Figure 3.4: The structure of the generated verification components.

3.4.1 Interface Verification Component

The interface Verification Component is a generic layered component constructed from several smaller components, including at least one interface, which ensures the connectivity between the verification environment and the DUT.

Following the top-down hierarchical structure of the iVC, the environment class is the highest component in the hierarchy, which instantiates the rest of the components. For every component in the hierarchy, there is a configuration file that specifies its behavior, allowing flexible and customizable testing.

The Agent creates a TLM (Transaction Level Modeling) interface port of a specific data type, and instantiates the monitor and its configuration object. Moreover, it consists of a *connect_phase* method, which establishes connections between components, specifically connecting the monitor's analysis port to the agent's analysis port.

The monitor class, as mentioned in the previous section, is responsible for observing and collecting data from the DUT. In particular, the monitor is dedicated to convert the pin-level activity into a sequence item object and transmit it to its TLM port. The basic monitor structure includes the *build_phase* and the *connect_phase*. The build phase functionality is to construct the monitor, while the *connect_phase* function establishes connections between other components. Lastly, other monitoring mechanisms include coverage collection and interface protocol functionality checks.

Each iVC is dedicated with one interface that defines a set of signals and their direction. In our tool-generated interface, a clocking block is defined, which is used to collect signals synchronized with a specific clock and to describe the timing relationships between the clock and the signals. The interface also includes a modport, which imposes certain restrictions on interface access within a module by specifying the direction of the signals (input, output, or inout).

The sequence item is a class that includes all related data packages that will be received from or transmitted to the DUT through the interface. In addition, the data are declared as random in the sequence item to allow the generation of stimuli from the higher-level testbench. These data will reach the reference model and the scoreboard, where they will be compared with the expected results.

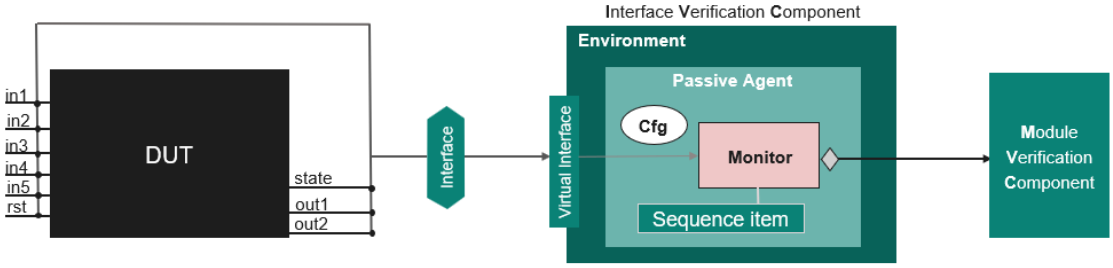

Figure 3.5: Example of a generated Interface Verification Component

3.4.2 Module Verification Component

The module Verification component is also a generic layered component constructed from several components, but the difference compared to the previous one is that it is used to verify the functionality of a specific module within a digital design. The environment class follows a similar approach to the iVC's environment hierarchy. However, the agent class introduces some notable differences. In particular, besides creating a TLM port, it also instantiates the reference model and the scoreboard. Additionally, in its *connect_phase* function, the agent establishes connections between the reference model and the scoreboard through analysis ports.

The reference model class includes an analysis port, which is used to receive data objects from other components. The data transmission is established through a *write* method which is called when a data object is received via the analysis port. This ensures synchronization between the DUT and the reference model. In our scope, this function updates the current state of the FSM based on the input data and the current state of the DUT. This function determines the next expected state of the specified FSM. Additionally, it checks if the input data are invalid or if the FSM is in an invalid state and reports an error using the *uvm_error* macro.

Another component in the mVC is the scoreboard, which is used to verify the behavior of a DUT with the corresponding specification. The main functionality of this scoreboard is to examine the expected state with the actual one. To guarantee synchronization and data cross-checking, there is a write function which receives the "live" value of the DUT, and then calls the *match_states* function. The *match_states()* uses an assertion mechanism to check that the actual state matches the predicted state and reports an error in all other cases.

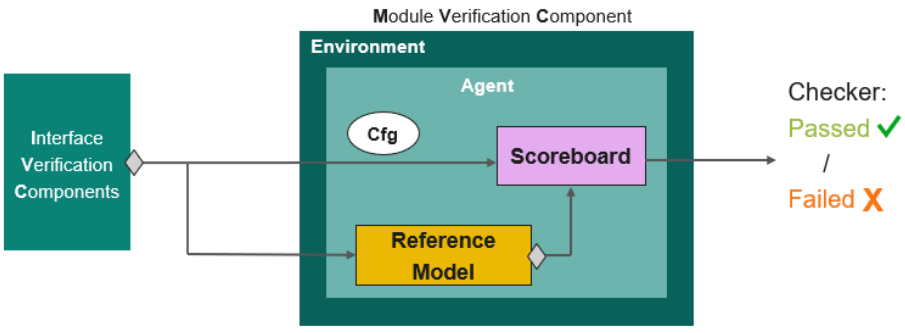

Figure 3.6: Example of a generated Module Verification Component

3.5 Example

In this section, we will present an example usage of the tool, and illustrate the practical application of the concepts and techniques discussed in the preceding sections.

Let us take the scenario where the following FSM is given.

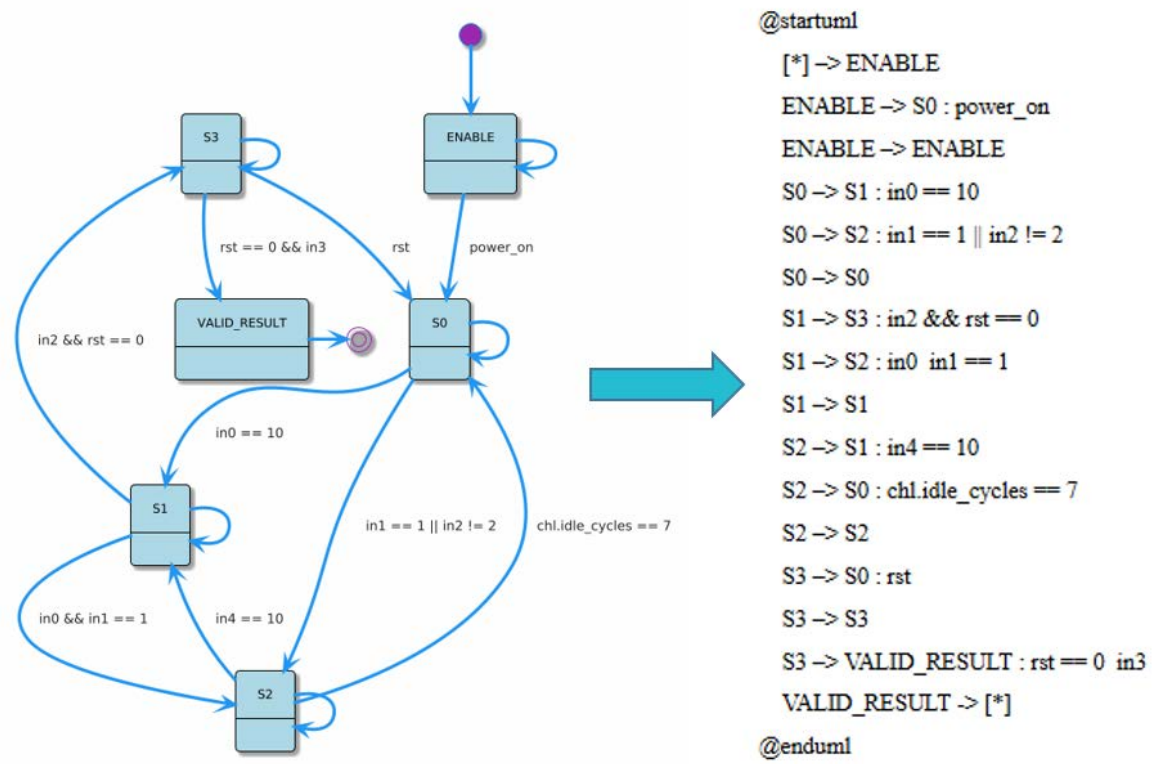

Figure 3.7: FSM in PlantUML Format

First, we need to specify some configuration options such as the project name, the path of the plantuml file, etc.

The configuration we will use for this example is the following:

```
# Specify the informations of the FSM
[fsm]
project_name = "example"
check_fsm_consistency = false

# plantuml_file = specify the full path to the plant uml file
plantuml_file = "/edimkou/.../inputs/example.plantuml"

# output_dir = specify the full path where the generated components will
    be placed
```

```
output_dir = "/edimkou/.../source/sv/tb/"

# state_enums = specify the enum name
state_enums = "test_0_state_t"

[fsm.ref_class]
# if exists, specify the reset signal and if the reset is enabled the
    predicted state to a default state
reset_signal = "rst"
default_state = "S0"

# ************* Analysis Ports *************
[[fsm.ref_class.analysis_ports]] # nested array of tables
name = "example_ap1"
arg_type = "example_seq_item"
identifier = "example_aimp"

# ************* Covergroups *************
[[fsm.ref_class.covergroup]]
covergroup_identifier = "example_cg"
coverpoints = [
    {coverpoint_label = "TRANSACTIONS", coverpoint = "curr_ref_state"}
]
covercrosses = []
```

Listing 3.1: FSM Configuration File example

After specifying the configuration file, the next and final step for the sv generation files is to run the script:

```
rf_generation -i ./inputs/example.toml
```

Now that the mVC and the iVC have been generated, the next step is to connect them to the testbench.

To connect them to the testbench, we should do the following modifications/adaptations in the top-level component:

```
class example_tb_env extends uvm_env;

  example_tb_config  cfg;
  example_tb_virtual_sequencer virtual_sequencer;
```

```
  example_ivc_env example_ivc;
  example_mvc_env example_mvc;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    uvm_config_db #(example_ivc_env_cfg)::set(this, "example_ivc", "cfg",
    cfg.ex_ivc_config);
    example_ivc = example_ivc_env::type_id::create("example_ivc", this);

    uvm_config_db #(example_mvc_env_cfg)::set(this, "example_mvc", "cfg",
    cfg.ex_mvc_config);
    example_mvc = example_mvc_env::type_id::create("example_mvc", this);
 endfunction

 virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //connect the iVC with the mVC
    example_ivc.agent.monitor.example_ap.connect(example_mvc.agent.
   example_ap);
 endfunction
endclass
```

Listing 3.2: Testbench Enviroment

```
class example_tb_config extends uvm_object;

    rand example_ivc_cfg ex_ivc_config;
    rand example_mvc_cfg ex_mvc_config;
    ...
    function new( string name = "example_tb_config" );
    super.new(name);
    ex_ivc_config=example_ivc_cfg::type_id::create("ex_ivc_config");
    ex_mvc_config=example_mvc_cfg::type_id::create("ex_mvc_config");
  endfunction
endclass
```

Listing 3.3: Testbench Enviroment's Configuration

Interface connection

```
module example_tb_hvl;
  import uvm_pkg::*;
  import example_tb_test_pkg::*;

 // register the monitor interface into factory
  initial begin
    uvm_config_db #(virtual example_ivc_agent_monitor_bfm)::set(
    null,
    "uvm_test_top",
    "ex_ivc_agent_mon",
    example_tb_hdl.ex_ivc_agent_mon);
  end

  initial begin
    #0;
    run_test();
  end
endmodule
```

Listing 3.4: Testbench Enviroment's Interface

```
class example_tb_test extends uvm_test;
  example_tb_config cfg;
  example_tb_env    tb_env;
  basic_seq_random m_random_seq;

  `uvm_component_utils_begin(example_tb_test)
  `uvm_component_utils_end

  function new( string name = "example_tb_test", uvm_component parent =
    null );
    super.new(name, parent);
  endfunction

  virtual protected function void connect_interfaces();
    ...
    if(!uvm_config_db #(virtual example_ivc_agent_monitor_bfm)::get(
        this,
        "",
        "ex_ivc_agent_mon",
```

```
19          cfg.ex_ivc_config.agent_agent_config.monitor_cfg.mon_bfm)) begin
20              `uvm_fatal(get_full_name(),"Could not get virtual interface
    ex_ivc_agent_mon from configuration database.")
21      end
22    endfunction
23
24    virtual function void build_phase(uvm_phase phase);
25      super.build_phase(phase);
26      // create top configuration object
27      create_config();
28      // configure the configuration objects
29      post_randomize_config();
30      // create the TB env
31      create_tb_env();
32      // virtual interfaces
33      connect_interfaces();
34    endfunction
35
36    virtual task run_phase(uvm_phase phase);
37     m_random_seq = basic_seq_random::type_id::create("m_random_seq");
38     if (!m_random_seq.randomize()) begin
39          `uvm_fatal(get_type_name(), "Top Sequence Randomization Failed.")
40      end
41      `uvm_info(get_type_name(), $psprintf("Printing object\n %s",
    m_random_seq.sprint()), UVM_NONE)
42      m_random_seq.start(tb_env.virtual_sequencer);
43      post_run_phase(phase);
44    endtask: run_phase
45
46  endclass
```

Listing 3.5: UVM test Example

In order to enable stimuli, a random sequence has been developed which will drive random input values. This will be repeated a set number of times in the range [0,100].

After running an example test, the simulation looks like this:

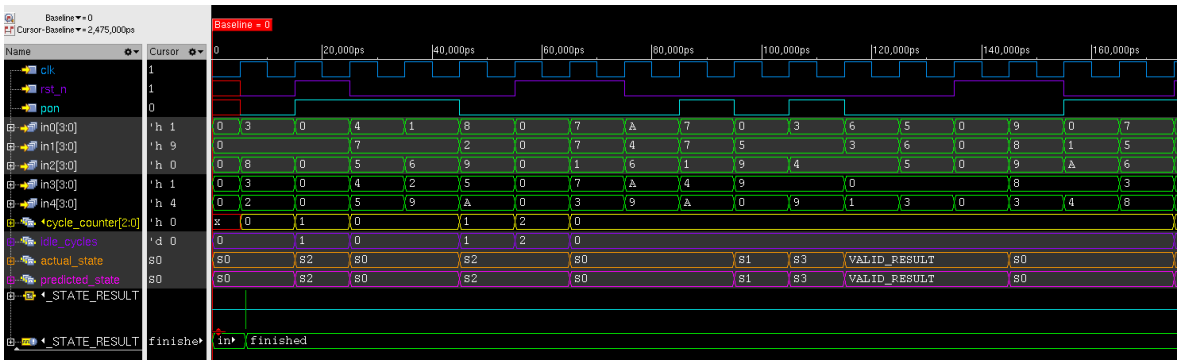

Figure 3.8: Simulation Example Result

Verification Scenario: Detection of the intended design mismatch.

To evaluate the effectiveness of the verification environment in capturing potential design errors, we intentionally introduce an RTL implementation error to assess the verification environment error detection capabilities, simulating real-world scenarios where design mistakes may originate from either engineer misinterpretation or ambiguous specifications. Specifically, we examine a discrepancy in idle cycle counter behavior between the verification reference model and RTL implementation.

Specific Test Case: Idle cycle calculation mismatch.

The *verification engineer* implemented the idle cycle counter behavior as follows: Increment the counter on every positive clock edge and reset to zero upon system reset.

However, the *RTL designer* implemented the logic as shown in Listing 3.6, where the counter additionally resets when transitioning to any state other than S2.

This creates a conflict between the two different implementations.

```
module dut(...);
  ...
 // Idle cycle counter logic
  always_ff @(posedge clk) begin
      if (rst_n || state != S2) begin
          idle_cycle_counter <= 0;
      end else begin
          idle_cycle_counter <= idle_cycle_counter + 1;
      end
  end
endmodule
```

Listing 3.6: RTL Implementation of Idle Cycle Counter

After running the simulation, we check whether the VE is able to detect the mismatch. As we can see in the figure 3.9 below, the assertion mechanism failed, and we were able to catch the inconsistency.

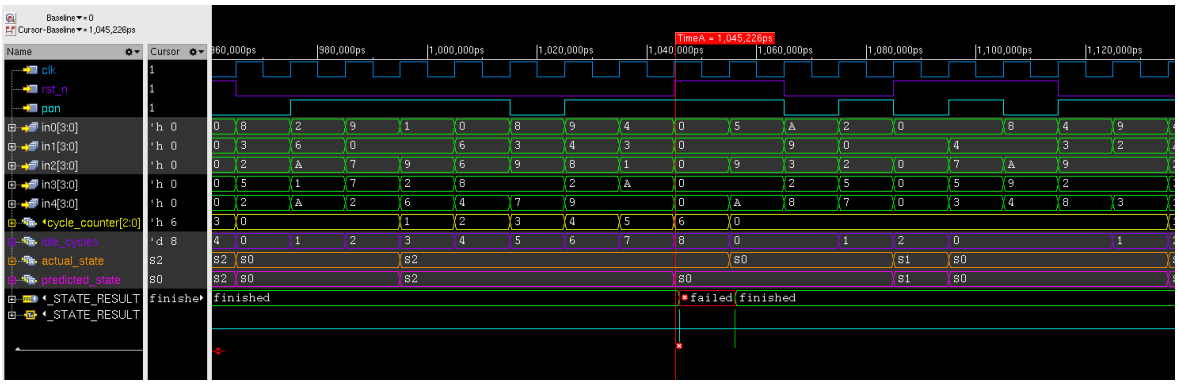

Figure 3.9: Simulation Results Showing Idle Cycle Counter Implementation Discrepancy

Chapter 4

Results

4.1 Overview

This section presents the results of applying the proposed FSM-based methodology to generate UVM-compliant reference models for verification. The functionality of the tool was tested by integrating the generated reference models into a verification environment and running simulations with various IP designs. These designs included functional designs as well as intentionally faulty ones to validate the tool's ability to identify issues. The results confirm the practicality and reliability of the automated workflow.

4.2 Testing Flow

The testing process followed these steps:

1. **FSM Creation:** The testing process began with the creation of various FSMs of different sizes (empty, small, large, etc.) that served as the single source of truth, defining the expected behavior of the designs under test.

2. **Code Generation:** The tool automatically generates UVM-compliant reference models and verification components based on the provided FSM.

3. **Integration:** The generated components were then integrated into a manually built verification environment to simulate a realistic workflow.

4. **Simulation:** The simulations were executed using the Xcelium Cadence tool.

5. **Analysis:** The simulation results were analyzed to evaluate the correctness and detect functional errors.

6. **Regression Tests:** Regression tests were performed using Verisium Manager to confirm that the design aligned the verification environment in multiple scenarios and across multiple iterations.

4.3 Test Cases

Several test cases were executed to validate the generated components and assess the effectiveness of the tool. One notable test case involved replicating the reference model of a real-world, production-ready ADC (Analog-to-Digital Converter). The tool-generated components achieved excellent results, reliably validating the ADC design and proving that the approach is practical for real-world applications.

In another set of tests, colleagues provided example designs along with FSMs that, in a real-world scenario, would form part of the IP specification. These FSMs were used to generate the iVC and mVC, including the reference models for their respective DUT. The results revealed that when the FSM matched the behavior of the DUT, the simulations were successful and all tests were passed. In contrast, when the DUT deviated from the FSM specifications, the generated components correctly identified functional errors and detected the faults during simulation and regression analysis.

4.4 Key Observations

- UVM-compliant reference models and other verification components were generated for all tested designs.

- Faulty designs were accurately identified during verification, demonstrating the tool's ability to detect design errors.

- The generated code is entirely dependent on the provided FSM as it describes the intended functionality and behavior of the design.

- The tool worked seamlessly across various types and scales of design sources, demonstrating its versatility.

These results highlight the practical value of the proposed methodology. By automating the generation of reference models, the workflow significantly reduces manual effort and ensures consistent, error-free verification setups. The successful detection of faults in defective designs further emphasizes the reliability of the tool, making it a valuable asset in the verification of mixed-signal IPs.

Chapter 5

Conclusion

5.1 Summary

This thesis focused on the development of an FSM-based methodology for automating the generation of UVM-compliant verification components. The study was motivated by the challenges faced in the mixed-signal IP verification workflow, such as specification ambiguity, manual errors in model creation, and inefficiencies in traditional workflows. By centralizing FSMs as the single source for describing the main functionality of the designs, the proposed methodology aims to streamline the verification process, improve accuracy, and reduce the manual effort required to create verification environments.

The key contribution of this work lies in its ability to automate the generation of critical verification components (such as reference models, monitors, agents, sequence items, etc.) directly from an FSM description. This approach provides a structured and reliable way to bridge the gap between design specifications and verification workflows. By integrating with industry-standard tools like Xcelium Cadence and Verisium Manager, the methodology aligns with existing verification practices, ensuring a smooth adoption process for verification. However, the effectiveness of this approach relies heavily on the accuracy of the provided FSM, as any errors in the FSM are directly propagated into the generated components. This dependency highlights the importance of creating precise and error-free FSMs, as they serve as the single source of truth within the proposed methodology.

5.2 Future Work

Although this work has shown promising results, there are several areas for future development that could further enhance its capabilities and applicability, such as:

- **Automated Stimulus Generation:**
  To ensure thorough testing, the tool could be improved to automatically generate stimuli that trigger all possible FSM transitions. This would improve test coverage and simplify the process of identifying edge cases and corner conditions during verification.

- **UVM Event-Based Approach:**
  Incorporating a UVM event-based approach into the methodology could enable more flexible and dynamic verification scenarios. Using UVM events, it would be possible to implement sophisticated synchronization mechanisms, and to further improve the efficiency of the verification environment.

Bibliography

[1] Jess Chen, Michael Henrie, Monte F. Mar, Mladen Nizic, and Brian Bailey. In *Mixed-Signal Methodology Guide: Mixed-Signal Design Trends and Challenges*, Chapter 1 2012.

[2] Chris Spear. Systemverilog for verification: A guide to learning the testbench language features. United States of America, 2006.

[3] Chipverify: Uvm phases. `https://www.chipverify.com/uvm/uvm-phases`. Accessed year: 2025.

[4] Chipverify: Uvm introduction. `https://www.chipverify.com/uvm/uvm-introduction`. Accessed year: 2025.

[5] Accellera Systems Initiative (Accellera). In *Universal Verification Methodology (UVM) 1.2 User's Guide*, `https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf`, October 2015.

[6] Finite state machine: Mealy state machine and moore state machine. `https://www.elprocus.com/finite-state-machine-mealy-state-machine-and-moore-state-machine/`. Accessed year: 2025.

[7] Monolithic Power Systems. In *Analog Signals vs. Digital Signals*, `https://www.monolithicpower.com/en/learning/resources/analog-vs-digital-signal`, Accessed year: 2025.

[8] Quarktwin. In *Challenges and Opportunities in Mixed-Signal IC Design*, `https://www.quarktwin.com/blogs/other/challenges-and-opportunities-in-mixed-signal-ic-design/508`, June 2024.

[9] Niranjana Gurushankar. In *Functional Verification Methodologies for Mixed Signal Designs*, https://www.researchgate.net/publication/387487185_Functional_Verification_Methodologies_for_Mixed_Signal_Designs, June 2022.

[10] Wen Chen, Sandip Ray, Magdy Abadir, Jayanta Bhadra, and Li-C Wang. In *Challenges and Trends in Modern SoC Design Verification*, https://www.researchgate.net/publication/318890041_Challenges_and_Trends_in_Modern_SoC_Design_Verification, August 2017.

[11] Ashok B. Mehta. In *Introduction to SystemVerilog*, Introduction to System Verilog, Springer Nature Switzerland AG, pp. 2-4, July 2021.

[12] J. Bromley. In *If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language*, Paris, France, pp. 1-7 2013.

[13] Kathleen A Meade Sharon Rosenberg. In *A Practical Guide to Adopting the Universal Verification Methodology (UVM)*, pp. 1-7 2012.

[14] Ahhyung Shin, Yungi Um, Youngsik Kim, and Seonil Brian Choi. In *Saving and Restoring Simulation Methodology using UVM Factory Overriding to Reduce Simulation Turnaround Time*, Samsung Electronics Co., Ltd., Seoul, Korea, 2012.

[15] Chipverify: Uvm base classes. https://www.chipverify.com/uvm/uvm-utility-field-macros. Accessed year: 2025.

[16] Siemens verification academy: Verification methodology reference. https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/base/uvm_component-svh.html. Accessed year: 2025.

[17] Chipverify: Testbench structure. https://www.chipverify.com/uvm/uvm-testbench-top. Accessed year: 2025.

[18] Chipverify: Systemverilog-interface. https://www.chipverify.com/systemverilog/systemverilog-interface. Accessed year: 2025.

[19] Accellera Systems Initiative (Accellera). In *Universal Verification Methodology (UVM) 1.1 User's Guide*, https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.1.pdf, May 18 2011.

[20] Mihalis Yannakakis David Lee. a survey. In *Principles and Methods of Testing Finite State Machines*, https://www.researchgate.net/publication/2985061_Principles_and_Methods_of_Testing_Finite_State_Machines-a_Survey, September 1996.

[21] Wikimedia Foundation. Finite-state machine. https://en.wikipedia.org/wiki/Finite-state_machine, Accessed year: August 2025.

[22] State machine diagrams. https://www.uml-diagrams.org/state-machine-diagrams.html. Accessed year: 2025.

[23] Plantuml. https://plantuml.com/. Accessed year: 2025.

APPENDICES

Appendix A

Example of Generated Files

This appendix provides examples of the files generated by the methodology proposed in this thesis. These files demonstrate the automated creation of UVM-compliant verification components based on the provided FSM descriptions, enabling consistent and efficient verification workflows.

The first example demonstrates the mVC (module verification component), which represents the verification logic for a specific module in the Design Under Test (DUT). This is followed by the iVC (interface Verification Component), which specifies the interface that enables communication between the DUT and the surrounding verification environment.

These generated files serve as representative examples of the output produced by the workflow, illustrating the tool's ability to translate functional specifications into practical verification components automatically.

A.1 Module Verification Component (mVC) Example

```
1  package example_mvc_pkg;
2    import uvm_pkg::*;
3    `include "uvm_macros.svh"
4
5    import example_ivc_pkg::*;
6    `include "include/example_mvc_common.sv"
7    `include "include/example_mvc_ref_model_config.sv"
8    `include "include/example_mvc_ref_model.sv"
9    `include "include/example_mvc_ref_scoreboard_base.sv"
10   `include "include/example_mvc_ref_scoreboard_base_config.sv"
```

```
  `include "include/example_mvc_agent.sv"
  `include "include/example_mvc_agent_config.sv"
  `include "include/example_mvc_env.sv"
  `include "include/example_mvc_config.sv"
endpackage
```

Listing A.1: example_mvc_pkg.sv

```
typedef example_mvc_config;
typedef class example_mvc_agent;

class example_mvc_env extends uvm_env;
    //****** Member Variables Declaration ******
    example_mvc_config cfg;
    example_mvc_agent agent;

    `uvm_component_utils_begin(example_mvc_env)
        `uvm_field_object(cfg, UVM_ALL_ON)
    `uvm_component_utils_end

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        `uvm_info(get_type_name(), "Build phase.", UVM_LOW)

        if(cfg == null) begin
            if(!uvm_config_db #(example_mvc_config)::get(this, "", "cfg",
    cfg)) begin
                `uvm_fatal(get_type_name(), "Failed to get cfg from
   config db.")
            end
        end

        uvm_config_db #(example_mvc_agent_config)::set(this,"agent", "cfg
    ", cfg.agent_agent_config);
        agent = example_mvc_agent::type_id::create("agent", this);
    endfunction : build_phase
```

```
30
31      virtual function void connect_phase(uvm_phase phase);
32          super.connect_phase(phase);
33          `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
34      endfunction : connect_phase
35  endclass
```

Listing A.2: example_mvc_env.sv

```
1  class example_mvc_config extends uvm_object;
2      //****** Member Variables Declaration ******
3      example_mvc_agent_config agent_agent_config;
4
5      `uvm_object_utils_begin(example_mvc_config)
6          `uvm_field_object(agent_agent_config, UVM_ALL_ON)
7      `uvm_object_utils_end
8
9      function new(string name = "example_mvc_config");
10         super.new(name);
11         agent_agent_config = example_mvc_agent_config::type_id::create("
       agent_agent_config");
12     endfunction : new
13 endclass
```

Listing A.3: example_mvc_config.sv

```
1  typedef example_mvc_agent_config;
2  typedef class example_mvc_ref_scoreboard_base;
3  typedef class example_mvc_ref_model;
4
5  class example_mvc_agent extends uvm_agent;
6    //****** Member Variables Declaration ******
7    example_mvc_agent_config cfg;
8    example_mvc_ref_model ref_model;
9    example_mvc_ref_scoreboard_base ref_state_scoreboard;
10
11   `uvm_component_utils_begin(example_mvc_agent)
12     `uvm_field_object(cfg, UVM_ALL_ON)
13   `uvm_component_utils_end
14
```

```
15  //****** Analysis Ports Declaration ******
16  // Declare and create a TLM Analysis Port to receive or to send data
      objects to other components
17  uvm_analysis_port #(example_seq_item) example_aimp;
18
19  function new(string name, uvm_component parent);
20    super.new(name, parent);
21    example_aimp = new("example_aimp", this);
22  endfunction : new
23
24  virtual function void build_phase(uvm_phase phase);
25    super.build_phase(phase);
26    `uvm_info(get_type_name(), "Build phase.", UVM_LOW)
27
28    if(cfg == null) begin
29      if(!uvm_config_db #(example_mvc_agent_config)::get(this, "", "cfg",
      cfg)) begin
30        `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
31      end
32    end
33
34    uvm_config_db #(example_mvc_ref_model_config)::set(this,"ref_model",
     "cfg", cfg.ref_model_cfg);
35    ref_model = example_mvc_ref_model::type_id::create("ref_model", this)
     ;
36
37    uvm_config_db #(example_mvc_ref_scoreboard_base_config)::set(this,"
     ref_state_scoreboard", "cfg", cfg.ref_scoreboard_cfg);
38    ref_state_scoreboard = example_mvc_ref_scoreboard_base::type_id::
     create("ref_state_scoreboard", this);
39
40  endfunction : build_phase
41
42  virtual function void connect_phase(uvm_phase phase);
43    super.connect_phase(phase);
44    `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
45    ref_state_scoreboard.ref_model = ref_model;
46
47    // Connections to ref_model
```

```
48      example_aimp.connect(ref_model.example_aimp);
49
50      // Connections to ref_state_scoreboard
51      example_aimp.connect(ref_state_scoreboard.example_aimp);
52    endfunction : connect_phase
53  endclass
```

Listing A.4: example_mvc_agent.sv

```
1  class example_mvc_agent_config extends uvm_object;
2    //****** Member Variables Declaration ******
3    example_mvc_ref_model_config ref_model_cfg;
4    example_mvc_ref_scoreboard_base_config ref_scoreboard_cfg;
5
6    `uvm_object_utils_begin(example_mvc_agent_config)
7      `uvm_field_object(ref_model_cfg, UVM_ALL_ON)
8      `uvm_field_object(ref_scoreboard_cfg, UVM_ALL_ON)
9    `uvm_object_utils_end
10
11   function new(string name = "example_mvc_agent_config");
12     super.new(name);
13     ref_model_cfg = example_mvc_ref_model_config::type_id::create("
       ref_model_cfg");
14     ref_scoreboard_cfg = example_mvc_ref_scoreboard_base_config::type_id
       ::create("ref_scoreboard_cfg");
15   endfunction : new
16 endclass
```

Listing A.5: example_mvc_agent_config.sv

```
1  typedef example_mvc_ref_scoreboard_base_config;
2
3  class example_mvc_ref_scoreboard_base extends uvm_scoreboard;
4    //****** Member Variables Declaration ******
5    example_mvc_ref_scoreboard_base_config cfg;
6    example_mvc_ref_model ref_model;
7    test_0_state_t predicted_state, actual_state;
8
9    `uvm_component_utils_begin(example_mvc_ref_scoreboard_base)
10     `uvm_field_object(cfg, UVM_ALL_ON)
```

```
11  `uvm_component_utils_end
12
13  //****** Analysis Ports Declaration ******
14  // Declare and create a TLM Analysis Port to receive or to send data
     objects to other components
15  `uvm_analysis_imp_decl(_example_ap1)
16  uvm_analysis_imp_example_ap1 #(example_seq_item,
     example_mvc_ref_scoreboard_base) example_aimp;
17
18  function new(string name, uvm_component parent);
19    super.new(name, parent);
20    example_aimp = new("example_aimp", this);
21  endfunction : new
22
23  virtual function void build_phase(uvm_phase phase);
24    super.build_phase(phase);
25    `uvm_info(get_type_name(), "Build phase.", UVM_LOW)
26
27    if(cfg == null) begin
28      if(!uvm_config_db #(example_mvc_ref_scoreboard_base_config)::get(
     this, "", "cfg", cfg)) begin
29        `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
30      end
31    end
32  endfunction : build_phase
33
34  virtual function void connect_phase(uvm_phase phase);
35    super.connect_phase(phase);
36    `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
37  endfunction : connect_phase
38
39  //****** Write functions for every Analysis Port ******
40  // Define action to be taken when a packet is received via the declared
     analysis port
41  virtual function void write_example_ap1(example_seq_item item);
42    `uvm_info(get_type_name(), "example_ap1 trans received", UVM_HIGH);
43
44    predicted_state = ref_model.curr_ref_state;
45    actual_state = item.state;
```

```
46      match_states(predicted_state, actual_state);
47    endfunction : write_example_ap1
48
49    function void match_states(test_0_state_t predicted_ref_state,
      test_0_state_t actual_state);
50      `uvm_info(get_type_name(), $sformatf("Match the states checker:
      actual -> %s and predicted -> %s", actual_state.name(),
      predicted_ref_state.name()), UVM_LOW);
51      fork
52        #1 // add 1 simulation unit delay
53
54        CHK_STATE_RESULT : assert(actual_state == predicted_ref_state)
55        else begin
56          `uvm_error("REF_STATE_RESULT", {{"Unexpected reference State
      Result: ",
57            $sformatf("predicted state: %s", predicted_ref_state.name()),
58            $sformatf("actual state: %s", actual_state.name())}});
59          end
60      join_none
61    endfunction : match_states
62  endclass
```

Listing A.6: example_mvc_ref_scoreboard_base.sv

```
1  class example_mvc_ref_scoreboard_base_config extends uvm_object;
2    `uvm_object_utils_begin(example_mvc_ref_scoreboard_base_config)
3    `uvm_object_utils_end
4
5    function new(string name = "example_mvc_ref_scoreboard_base_config");
6      super.new(name);
7    endfunction : new
8  endclass
```

Listing A.7: example_mvc_ref_scoreboard_base_config.sv

```
1  class example_mvc_ref_model extends uvm_component;
2    //****** Member Variables Declaration ******
3    example_mvc_ref_model_config cfg;
4    test_0_state_t next_ref_state;
5    test_0_state_t curr_ref_state;
```

```
6   test_0_state_t pre_ref_state;
7   `uvm_component_utils_begin(example_mvc_ref_model)
8     `uvm_field_object(cfg, UVM_ALL_ON)
9   `uvm_component_utils_end
10
11  //****** Analysis Ports Declaration ******
12  // Declare and create a TLM Analysis Port to receive or to send data
      objects to other components
13  `uvm_analysis_imp_decl(_example_ap1)
14  uvm_analysis_imp_example_ap1 #(example_seq_item, example_mvc_ref_model)
      example_aimp;
15
16  //****** Covergorups ******
17  covergroup example_cg;
18    option.per_instance = 1;
19
20    TRANSACTIONS: coverpoint curr_ref_state;
21    VALID_DATA: coverpoint curr_ref_state;
22    EXAMPLE_CROSS: cross VALID_DATA, TRANSACTIONS;
23  endgroup : example_cg
24
25  function new(string name, uvm_component parent);
26    super.new(name, parent);
27    example_aimp = new("example_aimp", this);
28  endfunction : new
29
30  virtual function void build_phase(uvm_phase phase);
31    super.build_phase(phase);
32    `uvm_info(get_type_name(), "Build phase.", UVM_LOW)
33
34    if(cfg == null) begin
35      if(!uvm_config_db #(example_mvc_ref_model_config)::get(this, "", "
        cfg", cfg)) begin
36        `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
37      end
38    end
39  endfunction : build_phase
40
41  virtual function void connect_phase(uvm_phase phase);
```

```
42      super.connect_phase(phase);
43      `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
44
45    endfunction : connect_phase
46
47    //****** Write functions for every Analysis Port ******
48    // Define action to be taken when a packet is received via the declared
        analysis port
49    virtual function void write_example_ap1(example_seq_item item);
50      `uvm_info(get_type_name(), "example_ap1 trans received", UVM_HIGH);
51      pre_ref_state = curr_ref_state;
52
53      if(item.rst) begin
54        curr_ref_state = S0;
55        next_ref_state = S0;
56        return;
57      end
58      else begin
59        curr_ref_state = next_ref_state;
60      end
61
62      case (item.state)
63        ENABLE: begin
64          if(item.power_on) begin
65            next_ref_state = S0;
66          end
67          else begin
68            next_ref_state = ENABLE;
69          end
70        end
71        S0: begin
72          if(item.in0==10) begin
73            next_ref_state = S1;
74          end
75          else if(item.in1==1 || item.in2!=2) begin
76            next_ref_state = S2;
77          end
78          else begin
79            next_ref_state = S0;
```

```
80          end
81        end
82        S1: begin
83          if(item.in2 && item.rst==0) begin
84            next_ref_state = S3;
85          end
86          else if(item.in0 && item.in1==1) begin
87            next_ref_state = S2;
88          end
89          else begin
90            next_ref_state = S1;
91          end
92        end
93        S2: begin
94          if(item.in4==10) begin
95            next_ref_state = S1;
96          end
97          else if(chl.idle_cycles==7) begin
98            next_ref_state = S0;
99          end
100         else begin
101           next_ref_state = S2;
102         end
103       end
104       S3: begin
105         if(item.rst) begin
106           next_ref_state = S0;
107         end
108         else if(item.rst==0 && item.in3) begin
109           next_ref_state = VALID_RESULT;
110         end
111         else begin
112           next_ref_state = S3;
113         end
114       end
115       VALID_RESULT: begin
116       end
117       default: `uvm_error(get_type_name(), $sformatf("Wrong data %s",
    item.state.name()))
```

```
118     endcase
119
120   endfunction : write_example_ap1
121 endclass
```

Listing A.8: example_mvc_ref_model.sv

```
1 typedef enum {
2   // Client states:
3   S0,
4   S1,
5   S2,
6   S3,
7   VALID_RESULT,
8   ENABLE
9 } test_0_state_t;
```

Listing A.9: example_mvc_common.sv

A.2 Example of an Interface Verification Component (iVC)

```
1 package example_ivc_pkg;
2   import uvm_pkg::*;
3   `include "uvm_macros.svh"
4
5   `include "../../example_mvc/include/example_mvc_common.sv"
6   `include "include/example_seq_item.sv"
7   `include "include/example_ivc_monitor.sv"
8   `include "include/example_ivc_monitor_config.sv"
9   `include "include/example_ivc_agent.sv"
10  `include "include/example_ivc_agent_config.sv"
11  `include "include/example_ivc_env.sv"
12  `include "include/example_ivc_config.sv"
13 endpackage
```

Listing A.10: example_ivc_pkg.sv

```
1 interface example_ivc_agent_if();
2   //****** Member Variables Declaration ******
```

```
3   logic clk;
4   logic power_on;
5   logic [3:0] in0;          // Input 0
6   logic [3:0] in1;          // Input 1
7   logic [3:0] in2;          // Input 2
8   logic [3:0] in3;          // Input 3
9   logic [3:0] in4;          // Input 4
10  logic rst;
11
12  // ********************************
13  // Add the currect state recieved from the DUT.
14  // ********************************
15  logic [3:0] current_state;
16
17  clocking cb_mon @(posedge clk);
18    default input #1step;
19    input power_on;
20    input in0;
21    input in1;
22    input in2;
23    input rst;
24    input in4;
25    input in3;
26    input current_state;
27  endclocking : cb_mon
28
29  modport monitor (
30    clocking cb_mon,
31    input power_on,
32    input in0,
33    input in1,
34    input in2,
35    input rst,
36    input in4,
37    input in3,
38    input current_state
39  );
40
41 endinterface
```

Listing A.11: example_ivc_agent_if.sv

```
interface example_ivc_agent_monitor_bfm(example_ivc_agent_if vc_if);
  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import example_ivc_pkg::*;
  example_ivc_monitor proxy;

  task collect(example_seq_item item);
    `uvm_error("In example_ivc/example_ivc_agent_monitor_bfm", $sformatf(
    "Check if the signals are collected properly. Please remove after you
    checked!"))

    forever begin
      @(vc_if.cb_mon);
      `uvm_info("In example_ivc/example_ivc_agent_monitor_bfm", $psprintf
    ("Printing object\n %s",item.sprint()), UVM_NONE)

      if($cast(item.state, vc_if.cb_mon.current_state)) begin
        `uvm_error("In example_ivc/example_ivc_agent_monitor_bfm",
    $sformatf("The casting filed!"))
      end
      item.power_on = vc_if.cb_mon.power_on;
      item.in0 = vc_if.cb_mon.in0;
      item.in1 = vc_if.cb_mon.in1;
      item.in2 = vc_if.cb_mon.in2;
      item.rst = vc_if.cb_mon.rst;
      item.in4 = vc_if.cb_mon.in4;
      item.in3 = vc_if.cb_mon.in3;
      proxy.notify_item(item);
    end
  endtask
endinterface
```

Listing A.12: example_ivc_agent_monitor_bfm.sv

```
typedef example_ivc_config;
typedef class example_ivc_agent;

```

```
class example_ivc_env extends uvm_env;

  //****** Member Variables Declaration ******
  example_ivc_config cfg;
  example_ivc_agent agent;

  `uvm_component_utils_begin(example_ivc_env)
    `uvm_field_object(cfg, UVM_ALL_ON)
  `uvm_component_utils_end

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_type_name(), "Build phase.", UVM_LOW)

    if(cfg == null) begin
      if(!uvm_config_db #(example_ivc_config)::get(this, "", "cfg", cfg))
    begin
        `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
      end
    end

    uvm_config_db #(example_ivc_agent_config)::set(this,"agent", "cfg",
   cfg.agent_agent_config);
    agent = example_ivc_agent::type_id::create("agent", this);
  endfunction : build_phase

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
  endfunction : connect_phase
endclass
```

Listing A.13: example_ivc_env.sv

```
class example_ivc_config extends uvm_object;

```

```
3   //****** Member Variables Declaration ******
4   example_ivc_agent_config agent_agent_config;
5
6   `uvm_object_utils_begin(example_ivc_config)
7     `uvm_field_object(agent_agent_config, UVM_ALL_ON)
8   `uvm_object_utils_end
9
10  function new(string name = "example_ivc_config");
11    super.new(name);
12    agent_agent_config = example_ivc_agent_config::type_id::create("
      agent_agent_config");
13  endfunction : new
14
15 endclass
```

Listing A.14: example_ivc_env_config.sv

```
1  typedef example_ivc_agent_config;
2  typedef class example_ivc_monitor;
3
4  class example_ivc_agent extends uvm_agent;
5
6    //****** Member Variables Declaration ******
7    example_ivc_agent_config cfg;
8    example_ivc_monitor monitor;
9
10   `uvm_component_utils_begin(example_ivc_agent)
11     `uvm_field_object(cfg, UVM_ALL_ON)
12   `uvm_component_utils_end
13
14   //****** Analysis Ports Declaration ******
15   // Declare and create a TLM Analysis Port to receive or to send data
       objects to other components
16   uvm_analysis_port #(example_seq_item) example_aimp;
17
18   function new(string name, uvm_component parent);
19     super.new(name, parent);
20     example_aimp = new("example_aimp", this);
21   endfunction : new
22
```

```
23   virtual function void build_phase(uvm_phase phase);
24     super.build_phase(phase);
25     `uvm_info(get_type_name(), "Build phase.", UVM_LOW)
26
27     if(cfg == null) begin
28       if(!uvm_config_db #(example_ivc_agent_config)::get(this, "", "cfg",
       cfg)) begin
29         `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
30       end
31     end
32
33     uvm_config_db #(example_ivc_monitor_config)::set(this,"monitor", "cfg
       ", cfg.monitor_cfg);
34     monitor = example_ivc_monitor::type_id::create("monitor", this);
35   endfunction : build_phase
36
37   virtual function void connect_phase(uvm_phase phase);
38     super.connect_phase(phase);
39     `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
40     monitor.example_aimp = example_aimp;
41   endfunction : connect_phase
42 endclass
```

Listing A.15: example_ivc_agent.sv

```
1  class example_ivc_agent_config extends uvm_object;
2
3    //****** Member Variables Declaration ******
4    example_ivc_monitor_config monitor_cfg;
5
6    uvm_active_passive_enum is_active = UVM_PASSIVE;
7    bit has_functional_coverage = 0;
8    bit has_scoreboard = 0;
9
10   `uvm_object_utils_begin(example_ivc_agent_config)
11     `uvm_field_object(monitor_cfg, UVM_ALL_ON)
12     `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
13     `uvm_field_int(has_functional_coverage, UVM_ALL_ON)
14     `uvm_field_int(has_scoreboard, UVM_ALL_ON)
15   `uvm_object_utils_end
```

```
16
17   function new(string name = "example_ivc_agent_config");
18     super.new(name);
19     monitor_cfg = example_ivc_monitor_config::type_id::create("
     monitor_cfg");
20   endfunction : new
21 endclass
```

Listing A.16: example_ivc_agent_config.sv

```
1  typedef example_ivc_monitor_config;
2
3  class example_ivc_monitor extends uvm_monitor;
4    //****** Member Variables Declaration ******
5    example_ivc_monitor_config cfg;
6    virtual example_ivc_agent_monitor_bfm bfm_m;
7    bit checks_enable;
8    bit coverage_enable;
9
10   uvm_analysis_port #(example_seq_item) example_aimp;
11
12   `uvm_component_utils_begin(example_ivc_monitor)
13     `uvm_field_object(cfg, UVM_ALL_ON)
14     `uvm_field_int(checks_enable, UVM_ALL_ON)
15     `uvm_field_int(coverage_enable, UVM_ALL_ON)
16   `uvm_component_utils_end
17
18   function new(string name, uvm_component parent);
19     super.new(name, parent);
20   endfunction : new
21
22   virtual function void build_phase(uvm_phase phase);
23     super.build_phase(phase);
24     `uvm_info(get_type_name(), "Build phase.", UVM_LOW)
25
26     if(cfg == null) begin
27       if(!uvm_config_db #(example_ivc_monitor_config)::get(this, "", "cfg
     ", cfg)) begin
28         `uvm_fatal(get_type_name(), "Failed to get cfg from config db.")
29       end
```

```
30      end
31    endfunction : build_phase
32
33    virtual function void connect_phase(uvm_phase phase);
34      super.connect_phase(phase);
35      `uvm_info(get_type_name(), "Connect phase.", UVM_LOW)
36      bfm_m = cfg.mon_bfm;
37      bfm_m.proxy = this;
38    endfunction : connect_phase
39
40    virtual task collect(example_seq_item item);
41      bfm_m.collect(item);
42    endtask
43
44    virtual task perform_checks(example_seq_item item);
45      `uvm_fatal(get_type_name(),"This virtual function must be overwritten
      ");
46    endtask
47
48    virtual task collect_transactions(uvm_phase phase);
49
50       forever begin
51        example_seq_item example_aimp_item;
52        example_aimp_item = example_seq_item::type_id::create("
      example_aimp_item", this);
53        collect(example_aimp_item);
54
55        if(checks_enable) begin
56          perform_checks(example_aimp_item);
57        end
58
59        example_aimp.write(example_aimp_item);
60      end
61    endtask
62
63    virtual task run_phase(uvm_phase phase);
64      super.run_phase(phase);
65      `uvm_info(get_type_name(), "Run phase in monitor. Ready to collect
      transactions.", UVM_LOW)
```

```
66      collect_transactions(phase);
67    endtask
68
69    task notify_item(example_seq_item item);
70
71      if(checks_enable) begin
72        perform_checks(item);
73      end
74      example_aimp.write(item);
75    endtask : notify_item
76  endclass
```

Listing A.17: example_ivc_monitor.sv

```
1   class example_ivc_monitor_config extends uvm_object;
2     //****** Member Variables Declaration ******
3     virtual example_ivc_agent_monitor_bfm mon_bfm;
4
5     `uvm_object_utils_begin(example_ivc_monitor_config)
6     `uvm_object_utils_end
7
8     function new(string name = "example_ivc_monitor_config");
9       super.new(name);
10    endfunction : new
11  endclass
```

Listing A.18: example_ivc_monitor_config.sv

```
1   class example_seq_item extends uvm_sequence_item;
2     //****** Member Variables Declaration ******
3     rand logic power_on;
4     rand logic rst;
5     rand logic [3:0] in0;          // Input 0
6     rand logic [3:0] in1;          // Input 1
7     rand logic [3:0] in2;          // Input 2
8     rand logic [3:0] in3;          // Input 3
9     rand logic [3:0] in4;          // Input 4
10    rand test_0_state_t state;
11
12    `uvm_object_utils_begin(example_seq_item)
```

```
13      `uvm_field_int(power_on, UVM_ALL_ON)
14      `uvm_field_int(in0, UVM_ALL_ON)
15      `uvm_field_int(in1, UVM_ALL_ON)
16      `uvm_field_int(in2, UVM_ALL_ON)
17      `uvm_field_int(rst, UVM_ALL_ON)
18      `uvm_field_int(in4, UVM_ALL_ON)
19      `uvm_field_int(in3, UVM_ALL_ON)
20      `uvm_field_enum(test_0_state_t, state, UVM_ALL_ON)
21    `uvm_object_utils_end
22
23    function new(string name = "example_seq_item");
24      super.new(name);
25    endfunction : new
26
27    virtual function void reset();
28      `uvm_fatal(get_type_name(),"This virtual function reset() must be
       overwritten in the example_seq_item");
29
30      power_on = 0;
31      in0 = 0;
32      in1 = 0;
33      in2 = 0;
34      rst = 0;
35      in4 = 0;
36      in3 = 0;
37      state = S0;
38    endfunction : reset
39
40    // TODO: add constraint
41    // constraint rst_const { soft rst inside {[0:1]};}
42  endclass
```

Listing A.19: example_seq_item.sv