INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2026, VOL. 72, NO. 2, PP. 1–6
Manuscript received February 20, 2026; revised April 2026. doi: 10.24425/ijet.2026.157943

Cocotb in modern functional verification – a critical Review and comparative analysis with SystemVerilog/UVM

Kamil Piekoszewski, and Mariusz Rawski

***Abstract*—This paper presents a review of recent scientific literature concerning the use of cocotb as a Python-based framework for functional verification of digital systems. The study categorizes existing works into three groups: design verification case studies, tool enhancement methodologies, and comparative analyses between cocotb and SystemVerilog/UVM environments. The strengths and limitations of cocotb are evaluated with respect to accessibility, ecosystem maturity, constrained random verification capabilities, and industrial applicability. The analysis reveals that cocotb provides a flexible, cost-effective solution, particularly suited to early-stage RTL verification and open-source workflows, while SystemVerilog/UVM remains advantageous for large-scale industrial projects due to its mature ecosystem and commercial tool integration. The paper identifies current gaps in methodology evaluation, coverage analysis consistency, and reproducibility in existing research, and outlines directions for future development of hybrid verification flows.**

***Keywords*—digital systems; functional verification; cocotb; hybrid verification flows**

I. Introduction

FUNCTIONAL verification has undergone significant transformations over the past few decades, driven by the increasing complexity of both hardware and software systems. Early approaches relied heavily on direct testing methods, where predefined input stimuli were applied to a design-under-test (DUT) to validate its behavior. While effective for simpler designs, this methodology struggled to scale with the advent of sophisticated digital systems, which are described using Hardware Description Languages (HDLs) such as Verilog and VHDL.

SystemVerilog, standardized as IEEE 1800 [1], extends Verilog with advanced constructs for object-oriented programming, constrained random verification, assertions, and functional coverage. It integrates features for both hardware description and verification. Its object-oriented programming capabilities, advanced data types, constrained randomization, and functional coverage mechanisms make it a powerful language for creating robust verification environments. By providing a unified framework for design and verification, SystemVerilog streamlines workflows and reduces development time.

Building on SystemVerilog's strengths, UVM (Universal Verification Methodology) defines a reusable class library and architectural framework for scalable verification environments [2]. Developed by Accellera and released in 2011, it introduces a standardized methodology for functional verification. UVM was derived from earlier methodologies, including OVM (Open Verification Methodology) and eRM (e Reuse Methodology). UVM introduces structured components such as agents, sequencers, drivers, monitors, scoreboards, and a register abstraction layer (RAL), enabling hierarchical and reusable verification environments. Its methodology is widely documented in both the official UVM User Guide and established verification literature [3], and has become the dominant industrial approach for constrained-random functional verification of complex digital systems. It provides a modular framework that promotes reusability and scalability. Its class library automates key verification tasks, such as sequences, data packing, and comparisons, while enabling engineers to create hierarchical testbenches tailored to complex designs.

UVM has become the gold standard for functional verification, providing a standardized framework for verifying complex digital systems. This framework offers numerous advantages, making it widely adopted in the industry. Unfortunately, not every golden standard is a silver bullet, and UVM has several notable drawbacks that can limit its effectiveness in certain scenarios. One of the biggest challenges associated with this methodology is its steep learning curve and its dependence on SystemVerilog as the base verification language. This creates a barrier to adoption, as companies need to hire highly trained engineers with specialized skills sets. Additionally, for smaller companies, the requirement to purchase expensive licenses for commercial EDA tools can pose a significant financial burden. As a result of these limitations, alternative open-source approaches to functional verification, such as *cocotb*, have emerged, offering a more accessible and cost-effective path for developing verification environments.

Although a number of recent publications describe individual cocotb-based verification case studies or limited comparisons with SystemVerilog/UVM environments, a structured

K Piekoszewski is with Faculty of Electronics and Information Technology, Warsaw University of Technology, Warsaw, Poland (e-mail: kamil.piekoszewski.dokt@pw.edu.pl).

M. Rawski is with Faculty of Electronics and Information Technology, Warsaw University of Technology, Warsaw, Poland (e-mail: mariusz.rawski@pw.edu.pl).

© The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/), which permits use, distribution, and reproduction in any medium, provided that the Article is properly cited.

synthesis of these findings remains limited. This paper contributes by organizing and analyzing existing literature according to verification scope, methodological features, and reported tool capabilities. In addition, it provides a structured qualitative comparison between cocotb and SystemVerilog/UVM across selected technical dimensions, including constrained random verification support, coverage mechanisms, architectural structure, and ecosystem maturity. Based on the reviewed evidence, the paper outlines a conceptual hybrid verification model intended to clarify practical scenarios in which cocotb, UVM, or a combination of both may be appropriate. Rather than positioning cocotb as a direct replacement for established industrial methodologies, this work aims to provide a balanced assessment of its current capabilities, limitations, and areas requiring further development.

This article is organized as follows. Section 2 describes the *cocotb* environment. Section 3 presents a review of scientific articles divided into three categories: Design Verification, Enhancing Verification Tools, and Cocotb–SV Comparison. Finally, Section 4 provides a summary and discusses the conclusions.

II. Cocotb

Cocotb [4], a coroutine-based co-simulation testbench framework for verifying digital designs, is one of the solutions that aims to address the main UVM problems. Over the last few years, it has gained popularity, especially among smaller companies that lack the resources to implement the full UVM flow. Its growth can be seen in [5], where it reached over 120,000 downloads last month. It offers several advantages over UVM, particularly in terms of accessibility, flexibility, and productivity. One of its primary benefits is ease of use, as cocotb leverages Python, a widely known and simpler language than SystemVerilog, significantly lowering the learning curve for hardware verification.

Additionally, it provides access to Python's extensive ecosystem of libraries for tasks like data analysis, randomization, and networking. Cocotb is an Open Source project, and its community has created a wide range of libraries enhancing its verification capabilities, like PyUVM [6], introducing a UVM-like structure of the testbench, PyVSC [7], which adds coverage and constrained randomization of variables or dedicated interface modules like AXI, I2C, etc. These additional libraries significantly reduce the development of verification testbench, especially compared to UVM, which does not provide such an open database for extension. The closed nature of SystemVerilog requires companies adopting UVM to rely heavily on closed Verification IPs (VIPs) from major EDA companies, which can substantially reduce verification environment development efforts at the expense of additional costs.

Cocotb works by co-simulating with a hardware description language (HDL) simulator via the Direct Programming Interface (DPI), a standard mechanism that enables communication between hardware simulators and programming languages such as C or Python. Cocotb leverages this interface to connect Python test logic directly to the simulated hardware. At its core, cocotb uses coroutines, a Python language feature that allows concurrent test routines to be written in a natural, readable way. Coroutines enable the testbench to wait for and react to specific events in simulation time, such as clock edges or signal changes, without blocking other operations. This approach enables clean, event-driven testbench code that can easily stimulate the design, monitor its behavior, and verify correctness.

III. Cocotb in literature

This section provides an overview of scientific articles on the use of cocotb in functional verification. The collected articles were sorted and assigned to categories describing the subject of each work, as shown in Table I.

TABLE I
Categories of cocotb articles

Article	Type
[8] [9] [10] [11] [12]	Design Verification
[13] [14] [15]	Enhancing Verification Tools
[16] [17] [18] [19]	Cocotb - SV Comparison

All the articles presented in the following sections of this paper describe the process of digital circuit verification, which uses cocotb to a greater or lesser extent. This is particularly evident in works classified as Design Verification type. Although less numerous, there are also articles proposing new verification methods based on cocotb, as well as works focusing on comparisons between cocotb testbenches and analogous verification environments written in SystemVerilog.

A. Design Verification

Contemporary formal verification provides a wide range of tools that enable the development of complex test environments for thoroughly validating digital circuits, with verification engineers as the primary target group. However, it should be noted that functional verification is also one of the fundamental tools used by engineers involved in digital circuit design (RTL designers). Due to differences in skills and responsibilities within the design process, the requirements they place on test environments are significantly different. For RTL designers, development time and ease of testbench implementation are of paramount importance, as they require efficient mechanisms to validate basic design functionality at very early stages of development. As a result, the structural complexity introduced by constrained-random verification (CRV) and Universal Verification Methodology (UVM) is often perceived as a drawback in this context. Consequently, direct tests implemented in hardware description languages remain the predominant verification approach among RTL designers.

In [10], the concept of using cocotb as a unit testing framework is presented. The term originates from software testing, and its underlying assumptions correspond to direct testing in functional verification. Under this approach, each test case validates a specific functionality of the design, while the engineer is responsible for developing the tests and

ensuring their correctness. In [10], cocotb was applied to verify a Floating Point Unit (FPU) that constitutes a component of a larger ASIC developed within the IDSoC project [20]. As noted at the beginning of this section, this method is particularly suitable for RTL designers who require a fast and lightweight approach to validating their designs during early development stages. Cocotb, as a Python-based library, enables rapid, efficient test environment construction and therefore represents a viable solution for this use case.

The authors of [10] followed a similar premise when verifying the design of a multi-master interconnection network supporting multiple AMBA protocols. Cocotb was used to develop a verification environment where direct tests targeted specific functionalities. The paper details five scenarios used to validate the design: QoS configuration, AHB Master access, AXI Master access, Cross-clock domain checking and Dynamic Arbitration.

A different approach, seen in works [9] and [12], involves using cocotb for system-level verification. Employing a Python-based verification environment appears logical. Access to external libraries and the overall ease of use should enable easier modeling of input signals and system-level functions, in contrast to SystemVerilog, which originates from hardware description languages.

In [9], this methodology was applied to verify a RISC-V processor. Load/Store memory access instructions were modeled in Python and used within the cocotb environment to test functionality of a Verilog implementation of a core supporting the RV32I instruction set.

A similar approach, but described in greater detail, is presented in work [12]. In this case, the entire process of verifying the RISC-V processor was divided into three stages, each employing different verification techniques. At the module level, where individual components were tested, UVM was used. At connectivity layer formal verification was applied, while cocotb was used to test chip-level functionality, such as interrupts, bus interconnection, and DMA transfers. The proposed solution was compared with the verification process of a similar processor based on the Cortex-M4F core. According to the authors, using the above multi-level verification method made it possible to reduce the team from 10 engineers to just 1. Unfortunately, the article lacks information about the team's experience as well as the methodology used to test the compared system, which significantly hinders the validation of these conclusions.

A significant issue in most of the articles analyzed in this text is the lack of information on whether the described design was put into production. Such information would confirm that the applied verification methodology is mature enough to validate circuits used in the industry. In this context, the work [8] stands out, where cocotb was used to create testbenches for several custom ASICs for the ATLAS ITk Strips detector. According to the author of the article, the first prototype units of the system were delivered in 2019, and they are ultimately planned to be installed in the Large Hadron Collider in 2026.

B. Enhancing Verification Tools

This chapter discusses papers that present new tools or methodologies based on cocotb, offering capabilities that go beyond the classical understanding of functional verification, which typically involves creating a testbench and preparing tests to validate RTL design.

In paper [13], an FPGA-based HW/SW co-verification environment is presented that provides fine-grained dataplane observability. The goal of the proposed solution is to create a test environment for modern reprogrammable network devices. Due to the specific nature of such systems, it is crucial to detect timing errors, which is not possible with the classical RTL testing approach. The presented solution aims to enable debugging and waveform analysis, which is significantly more difficult when the design is run on an FPGA.

Cocotb was used to create a testbench responsible for generating input data and simulating the model in either HDL or a functional language. The generated input data is also transmitted to the FPGA programmed with the HDL code. The output data from the FPGA platform is read and sent back to the cocotb environment, where it is compared with the simulation result. The solution allows synchronization of data between the simulation and the FPGA system at a 1 clock cycle level.

In theory, using a single test environment for both RTL model verification and FPGA implementation should minimize the possibility of errors related to physical implementation, particularly timing errors.

The integration of electronic devices into increasingly critical components of transportation systems, such as automobiles and aircraft, has led to the development of safety standards that define required reliability levels as early as the design phase. In [14], a solution based on cocotb is presented, which enables testing of systems to verify their compliance with such standards. The approach facilitates fault injection to evaluate system behavior in the presence of internal errors.

According to the authors, the primary advantage of the proposed solution is its ability to simulate multiple faults within a single process, in contrast to other methods that are limited to simulating one fault per process. However, this claim cannot be independently validated based solely on the references provided in the article, due to the closed nature of the software under discussion and the lack of information regarding the methodology against which this software was evaluated.

Last article [15] discussed in this chapter addresses the challenge of concurrent software and hardware development in the design of modern digital systems targeted for FPGA platforms. Traditional design methodologies typically involve an early-stage partitioning of system functionality into software and hardware domains. While this approach can simplify initial development, it may lead to significant challenges if redesign becomes necessary at later stages-particularly in cases where performance requirements are not met.

In [15], the authors introduce a novel methodology for the concurrent development of digital systems for FPGA platforms, built around the DUTILS framework. DUTILS is a Python-based toolkit developed by the authors that enables

early modeling of digital systems. This model supports the initiation of software development and the creation of a verification environment at an early stage of the project, while also allowing integration with the hardware described using a synthesizable hardware description language (HDL). The integration is achieved through the use of cocotb, which serves as the interface between DUTILS and the HDL-based hardware design.

Theoretically, the proposed methodology has the potential to significantly streamline the design of complex FPGA-based systems by enabling more flexible migration of functionality between software and hardware domains. However, as with several other approaches reviewed in this chapter, a major limitation lies in the lack of access to the source code of the DUTILS framework. This restricts the ability of independent researchers or practitioners to evaluate and adopt the proposed methodology. This limitation is particularly critical in this case, given that the primary contribution of the work is the introduction of a new design methodology-yet no testing tools or public implementations are provided to support its practical application.

The articles presented above demonstrate the potential of using cocotb to develop new tools that go beyond classical functional verification. As a Python-based library, cocotb provides free access to digital circuit simulation through an easy-to-use high-level programming language, significantly facilitating the development of new tools.

C. Cocotb – SV comparison

As the leading functional verification methodology, UVM serves as a benchmark for emerging techniques gaining popularity. Consequently, a number of research works focus on comparing verification approaches based on cocotb with the classical methodology employing SystemVerilog and the UVM framework.

The authors of [16] investigate whether it is possible to create a UVM-like constrained random verification (CRV) environment using the Python programming language. To this end, they developed a verification environment for a memory arbiter circuit using cocotb along with several Python libraries: pyuvm, cocotb-coverage, constraintrandom, and PyVSC. These libraries were thoroughly evaluated for their usefulness in constructing a CRV environment. The authors successfully verified the target design, highlighting the strengths and weaknesses of the chosen method and the employed libraries.

The pyuvm library enabled the creation of a UVM-like environment in Python. However, the study pointed out the absence of essential components such as Transaction-Level Modeling (TLM) 2.0 and a fully functional Register Abstraction Layer (RAL), both of which are critical parts of the standard UVM framework. It is worth noting that newer versions of pyuvm offer preliminary RAL support. However, not all classes are implemented, and there is currently no tooling available for automatic generation of the RAL layer from IPXACT or SystemRDL files.

The authors also identified limitations of open-source simulators commonly used with cocotb, such as Icarus Verilog and Verilator, particularly regarding incomplete SystemVerilog syntax support. One critical feature currently not supported is SystemVerilog Assertions, which play an important role in verifying communication interfaces within RTL designs.

The document also evaluates the capabilities of cocotb-coverage, PyVSC, and constraintrandom in terms of variable randomization and functional coverage collection. Regarding randomization, cocotb-coverage was excluded due to long execution times, its inability to randomize dynamic objects, and the significant boilerplate required for nested objects. The constraintrandom package exhibited the shortest randomization time for simple constraints and delivered an appropriate distribution. In the case of complex constraints, PyVSC's constraint solver was the fastest, but the uneven distribution of results makes it difficult to consider the tool mature enough for verifying complex designs.

Both cocotb-coverage and PyVSC were evaluated positively for collecting functional coverage during single simulation runs. However, the lack of support for merging coverage results from multiple simulations in cocotb-coverage, as well as incorrect merge results using dedicated tools in PyVSC, were identified as significant drawbacks.

The subsequent paper [18] focuses on the verification process of a simple master/slave SPI controller with register access via the APB protocol, using the cocotb framework, and compares it with the verification of an SPI controller performed in an SV/UVM environment as described in [21].

The authors provide a relatively detailed description of the entire verification process, presenting the structure of the cocotb testbench, the EDA tools employed, and the test plan. Commercial EDA tools from Synopsys were used for simulation and code coverage collection, while functional coverage was gathered using the cocotb-coverage library. However, the paper does not specify whether the drivers used in the verification environment are based on existing, publicly available cocotb libraries or whether they were developed specifically for the purposes of the project.

Code coverage was used as the primary metric for comparing the results with the controller described in [21]. Code coverage indicates the percentage of RTL code exercised during simulation. Nevertheless, this metric is difficult to consider fully representative, as it directly depends on the RTL implementation of the design under verification, which differs between the analyzed studies. Moreover, the functional coverage model includes only 56 coverpoints, indicating low design complexity. This raises concerns regarding the scalability of the reported results to significantly more complex designs.

An extension of the aforementioned work is presented in [19]. The paper presents the verification process of an AHB-SPI controller using three different environments: SystemVerilog/UVM, plain cocotb, and cocotb combined with the UVM-Python library. The AHB-SPI design is expected to exhibit higher complexity than an APB-SPI controller due to the use of the more sophisticated AHB protocol for register access. Unfortunately, the paper lacks information regarding the number of coverpoints as well as a description of the test plan, which significantly limits the ability to assess the complexity of both the design and the verification effort.

The SystemVerilog/UVM environment described in the article [19] was built using Verification Intellectual Property (VIP) for the SPI and AHB protocols provided by Synopsys. The second environment was developed using cocotb, leveraging the cocotbext-ahb and cocotbext-spi libraries as interface drivers. Since cocotb does not enforce a specific architecture for verification environments, it is likely that the authors adopted a structure similar to the example tests available in the cocotb GitHub repository [4]. The third environment under comparison is based on the UVM-Python library, which provides UVM-like classes for cocotb. Similar to the previous setup, the cocotbext-ahb and cocotbext-spi libraries were used as interface drivers.

The authors successfully verified the controller using all three environments, achieving identical coverage results in terms of both functional coverage and code coverage. A noteworthy conclusion from the article is that, thanks to the availability of a mature ecosystem, including VIPs from EDA tool vendors, the creation of an SV/UVM-based environment can be faster than one based on cocotb. However, when VIPs for specific interfaces do not exist or cannot be used due to cost constraints, cocotb offers a more accessible, faster path to building the environment, thanks to open-source libraries and the use of the Python programming language.

The final article [17] discussed in this section focuses on comparing verification environments built with cocotb and SV/UVM for testing an AES module. The authors claim that the cocotb-based environment has advantages over SV/UVM in terms of testbench construction efficiency and coverage analysis. Unfortunately, the article does not provide sufficient data to support such a strong claim, particularly with respect to the coverage analysis. The specific library used for collecting functional coverage is not mentioned, which is crucial given the variety of libraries available for cocotb-based environments. Furthermore, the article fails to explain the significant discrepancy-64 (sic!) versus 65,536-in the number of coverpoints between the UVM and cocotb environments, despite both testbenches allegedly being authored by the same team. These omissions cast doubt on the validity of the conclusions presented.

Analyzing the literature comparing verification using cocotb and SV/UVM environments, it can be concluded that cocotb offers significant advantages in the verification of smaller digital designs, especially where the cost of implementation and the availability of engineers skilled in SystemVerilog are limiting factors. For larger projects, the mature SV/UVM ecosystem and the availability of commercial EDA tools enable quicker development of a complete verification environment-provided that cost is not a critical constraint. However, even in such cases, cocotb remains a viable alternative. Access to libraries such as pyuvm, cocotb-constraint, PyVSC, and constraintrandom enables the construction of advanced CRV environments despite their current limitations.

IV. Summary

This document presented a review of the available scientific literature concerning cocotb. In the analyzed works, cocotb emerges as an increasingly popular framework for functional verification, whose open-source nature and foundation in the Python programming language address some of the key limitations of the widely adopted SystemVerilog/UVM-based approach. Although cocotb is a relatively young verification methodology, its maturity has been demonstrated across numerous projects of varying complexity, ranging from simple academic designs to production-grade silicon implementations.

Owing to its Python foundation, cocotb enables the development of tools that leverage digital circuit simulation, with applications extending beyond traditional functional verification. While cocotb does not natively provide essential features for constrained random verification-such as functional coverage, constraints, or a defined verification environment architecture-it benefits from integration with Python libraries that offer such capabilities. However, due to their open-source nature and early development stage, these libraries do not yet match the maturity and robustness of commercial EDA tools based on SystemVerilog/UVM.

An interesting hybrid approach involves the combined use of cocotb and SystemVerilog/UVM within a single verification flow, where different environments are employed at different levels of the verification process. Thanks to its ease of environment setup and the use of Python, cocotb can be effectively utilized by RTL designers for early-stage RTL module verification. In contrast, SV/UVM-being a more mature and structured methodology-can offer more comprehensive verification of complex components such as DMA engines or interface controllers. At the system level, cocotb can again be leveraged, this time for generating sophisticated traffic patterns with the help of Python libraries, making it a practical tool for high-level integration testing.

References

[1] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[2] "Ieee standard for universal verification methodology language reference manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, 2020.

[3] C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd ed. Springer Publishing Company, Incorporated, 2012.

[4] "GitHub - cocotb/cocotb: cocotb: Python-based chip (RTL) verification — github.com," https://github.com/cocotb/cocotb, [Accessed 05-04-2025].

[5] "PyPI Download Stats — pypistats.org," https://pypistats.org/packages/cocotb, [Accessed 05-04-2025].

[6] "GitHub - pyuvm/pyuvm: The UVM written in Python — github.com," https://github.com/pyuvm/pyuvm, [Accessed 05-04-2025].

[7] "GitHub - fvutils/pyvsc: Python packages providing a library for Verification Stimulus and Coverage — github.com," https://github.com/fvutils/pyvsc, [Accessed 05-04-2025].

[8] B. J. Rosser, "Verification of readout electronics in the atlas itk strips detector," *Proceedings of the 2019 Meeting of the Division of Particles and Fields of the American Physical Society, DPF 2019*, pp. 1–5, 2019.

[9] S. Guney and I. Cicek, "Verification of risc-v load and store instructions using a custom cocotb based testbench," *CIEES 2024 - IEEE International Conference on Communications, Information, Electronic and Energy Systems*, pp. 20–22, 2024.

[10] H. Yang, L. Wang, C. Xia, and B. Gao, "Cocotb-based verification of multi-protocol interconnection network," *2024 4th International Conference on Electronic Information Engineering and Computer Technology, EIECT 2024*, pp. 791–794, 2024.

[11] M. Ludwiniak and A. W. Łuczyk, "Using cocotb framework during different stages of ic design," *Mixed Design of Integrated Circuits and System, MIXDES 2024*, pp. 154–157, 2024.
[12] L. Li, Y. Ren, X. Liu, and N. Tan, "A multi-level verification method for a quad-core risc-v soc," *2024 9th International Conference on Integrated Circuits and Microsystems, ICICM 2024*, pp. 517–521, 2024.
[13] M. Su, J. P. David, Y. Savaria, B. Pontikakis, and T. Luinaud, "An fpga-based hw/sw co-verification environment for programmable network devices," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2022-May, pp. 2529–2533, 2022.
[14] M. H. Fayez, M. A. Eladawy, M. S. Sahyon, I. O. Ahmed, O. H. El-Din, M. A. Elshafie, M. A. Taha, and M. G. Talaat, "Fault simulation framework using pyuvm," *Proceedings of the International Conference on Microelectronics, ICM*, pp. 158–161, 2023.
[15] M. Trapaglia, R. Cayssials, L. D. Pasquale, and E. Ferro, "Flexible software to hardware migration methodology for fpga design and verification," *2019 10th Southern Conference on Programmable Logic, SPL 2019 - Proceedings*, pp. 39–44, 2019.
[16] C. Harvig, B. Kasper, A. Sohail, N. Jo, and J. Sander, "An implementation of a python-based verification environment using pyuvm and cocotb," 2024.
[17] N. Susithra, S. S. Kanna, K. S. Karthik, T. N. Raja, V. Yaswant, and K. Rajalakshmi, "Analyzing aes verification: A comparative study of uvm and cocotb approaches," *International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering, ICSSEEC 2024 - Proceedings*, pp. 478–482, 2024.
[18] H. Liang, N. Tan, Y. Ren, W. Hu, J. He, and J. Xia, "Python based testbench for coverage driven functional verification," *2022 7th International Conference on Integrated Circuits and Microsystems, ICICM 2022*, pp. 361–365, 2022.
[19] G. Wang, N. Tan, Y. Cheng, and P. Zhang, "A comparative study of different verification platforms of ahb-spi," *2024 9th International Conference on Integrated Circuits and Microsystems, ICICM 2024*, pp. 847–851, 2024.
[20] "IDSoC - Instytut Łączności - Portal Gov.pl — gov.pl," https://www.gov.pl/web/instytut-lacznosci/idsoc, [Accessed 05-04-2025].
[21] B. Xu and K. Mou, *Proceedings of 2020 IEEE International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA 2020) : November 6-8, 2020, Chongqing, China*. IEEE Press, 2020.