Enable Reuse of SystemVerilog Verification IPs in cocotb/pyuvm

Yilou Wang, Thorsten Dworzak and Johannes Grinschgl

September 27, 2024

Enable Reuse of SystemVerilog Verification IPs in cocotb/pyuvm

Yilou Wang, Munich, Germany

Thorsten Dworzak, Dr. Johannes Grinschgl, Infineon Technologies AG, Neubiberg, Germany

wangyilou123@gmail.com, Thorsten.Dworzak@infineon.com, Johannes.Grinschgl@infineon.com

***Abstract*—This paper presents a novel strategy for enhancing the Python verification ecosystem by integrating established SystemVerilog Verification IPs (SV-VIPs) utilizing the cocotb and pyuvm framework. Gradually gaining recognition within the verification community, Python-based environments are being explored for their potential to become mainstream in future verification processes. This approach taps into the established SystemVerilog ecosystem, enabling effective reuse of SV-VIPs within Python settings. By leveraging the Direct Programming Interface (DPI-C) and the ctypes library, our method ensures seamless integration between Python testbenches and SV-VIPs. This integration not only utilizes Python's simplicity and readability but also fortifies its capacity for handling sophisticated hardware verification tasks. The paper illustrates this methodology with two practical implementations. It shows Python's evolving significance as a powerful and adaptable verification language and bridges the current divides between software flexibility and hardware verification demands.**

Keywords—SystemVerilog; UVM; Python Verification; reuse; pyuvm; cocotb

I. Background and Introduction

As verification tasks proliferate, the exploration of the future of verification is intensifying. It is widely accepted that writing testbenches is a software problem. Therefore, Python, as a powerful software programming language, has naturally gained attention. Engineers and researchers are considering the possibility of Python becoming the future verification language after SystemVerilog. Python offers several inherent advantages as a verification language, such as:

- Enhanced Productivity and Widespread Use: Python's simplicity and efficiency speed up testbenches writing. Additionally, its popularity makes it more accessible and easier to learn compared to SystemVerilog or VHDL, broadening the pool of available talent and resources.
- Extensive Libraries and OpenSource Community: Python's vast libraries facilitate code reuse, reducing the need to write new code. The active OpenSource community provides significant momentum for Python verification, leading to the continuous emergence of more and richer IPs.
- Easy Interfacing: Python can seamlessly interface with other languages, increasing its flexibility and integration capabilities. This is especially beneficial for complex verification tasks, which require multiple interfaces.
- Interpretative Nature: Python's interpretative nature allows for quick test edits and reruns without the need for recompilation, significantly speeding up the testing and debugging process.

With continuous exploration into Python verification, it is now feasible and practical to write testbenches and run simulations using Python. Cocotb [2] uses coroutines to simulate hardware parallelism and Generalized Procedural Interface (GPI) to enable communication between Python testbenches and simulators, making it possible to write testbenches in Python. On top of cocotb, pyuvm [1] integrates the Universal Verification Methodology (UVM), which is highly popular in the industry, into the Python verification ecosystem. This allows users to enhance code reusability and improve testbench writing efficiency by using UVM within Python testbenches. Additionally, pyuvm and cocotb, combined with OpenSource simulators like Verilator, enable OpenSource UVM verification applications, further enhancing the benefits of using Python.

In this paper, we introduce a novel approach to reuse SystemVerilog-based Verification IPs (SV-VIPs) in a Python verification environment leveraging cocotb and pyuvm. This method represents a significant shift from the current research trends, which primarily focus on using a purely Python-based environment for verification [3] [4]. Unlike the current efforts that aim to create a Python verification infrastructure from scratch, this paper proposes the integration of existing commercial SV-VIPs, thus augmenting the Python verification ecosystem by repurposing these SV-VIPs. SystemVerilog has dominated the verification industry for over a decade, with the industry investing heavily in millions of SV-VIPs, which significantly enhance verification efficiency. These commercial SV-VIPs are almost indispensable to the current verification industry. Rewriting these in Python would undoubtedly be complex and inefficient. In contrast, if there were a method to reuse these SV-VIPs within a Python testbench, it would be a highly promising development. This approach would not only improve efficiency but also provide a pathway and interface for connecting the Python software world with the SystemVerilog hardware world.

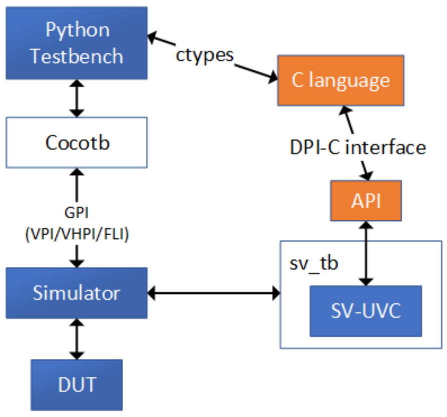

Figure 1. Pathway Diagram

The proposed methodology serves as a bridge between the established, hardware simulation oriented SystemVerilog environment and the versatile, high-level Python environment. This is achieved by establishing a pathway shown in Figure 1 where SV-VIPs are abstracted to the C language level via the Direct Programming Interface (DPI-C) and connected to Python through the ctypes library. Since we are discussing reuse, we assume we cannot make any changes to the internals of the SV-VIPs. Instead, we create an application programming interface (API) as a wrapper around the SV-VIPs and use DPI-C to achieve transaction-level communication via C language. This connection facilitates seamless interaction between Python testbenches and SV-VIPs, enabling the execution of verification tasks that leverage the strengths of both programming environments.

II. IMPLEMENTATION

The paper details two implementations that demonstrate the practicality and effectiveness of this approach. By discussing these implementations, readers can gain a comprehensive understanding of how communication and synchronization work, as well as how transaction-level data is transferred.

A. Controlling SV-VIPs in Python testbench

The first implementation allows a Python testbench to control the initiation and termination of an SV-VIP, integrating it as part of the Python testbench to complete verification tasks. As shown in Figure 2, the Python testbench connects to C language to control the SV-VIP by activating its internal sequence for initiation. The details of this process involve the design of a communication protocol, implementation of synchronization, and the addition of a user-defined task phase to allow the Python testbench to regain control.

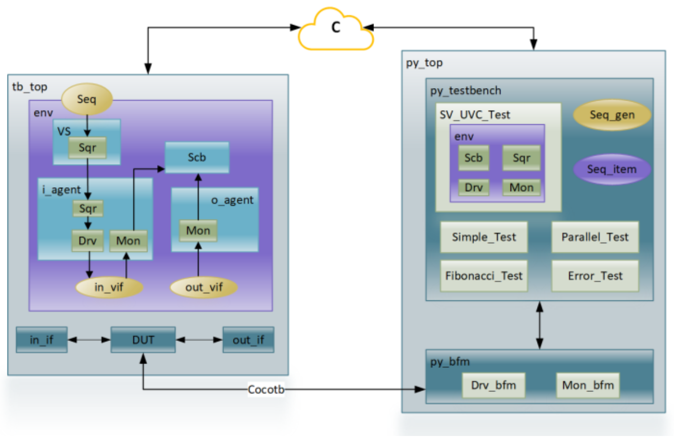

Figure 2. Testbench Framework of controlling SV-VIP test

The communication between the Python testbench and the SV-VIP follows the classic master-slave non-blocking communication model. In this setup shown in the left part of Figure 3, the Python testbench acts as the master, while the SV-VIP acts as the slave. The slave continuously performs non-blocking wait and request operations until it receives a release signal from the master. Once the Python testbench releases control, the SV-VIP executes its task. Meanwhile, the master performs non-blocking wait and query operations to monitor the state of the SV-VIP. Upon receiving a notification from the slave, the master regains control. The blue and green boxes in the figure represent which side currently has control. This non-blocking communication model ensures synchronization between the SV-VIP and the Python testbench, avoiding conflicts on the design side, and it also supports a master-to-multiple-slave configuration.

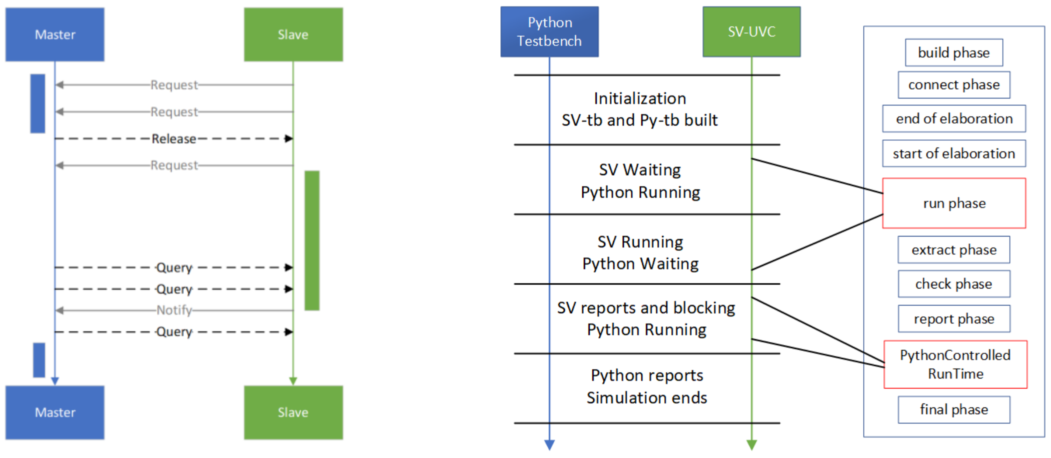

Figure 3. Communication Model (left) and Addition of user-defined task phase (right)

However, since the SV-VIP needs to return control to the Python testbench after completing its task and then continue in a non-blocking wait state, we need to create an additional task phase for the SV-VIP. This user-defined task phase, called the PythonControlledRunTime, operates within the UVM phase mechanism framework [5], positioned between the report phase and the final phase. The whole simulation process of two sides and the addition of the user-defined task phase are shown in the right part of Figure 3. To preserve all the original functionalities of the SV-VIP as much as possible, we moved the internal function report_summarize() from the UVM's final phase to the PythonControlledRunTime. This adjustment allows the SV-VIP to maintain its full range of functions while integrating seamlessly with the Python testbench.

This first implementation illustrates how SV-VIPs can operate independently within a Python-driven environment, performing their internal tasks.

B. SV-VIPs utilizing Python-generated Sequence

The second implementation further explores the dynamic capabilities of this framework by enabling the Python testbench to generate sequences and send sequence descriptors to an SV-VIP via the established pathway. The SV-VIP, upon receiving these descriptors, generates and dispatches sequences to the Device Under Test (DUT), monitors the output from the DUT, and sends the results back to the Python testbench, which are checked and compared in a Python scoreboard, as shown in Figure 4. This implementation showcases the bidirectional freedom of the pathway and supports dynamic transmission of high-level (transaction level) data. Utilizing this feature, Python can fully leverage its software capabilities, laying a solid foundation for future extensions.

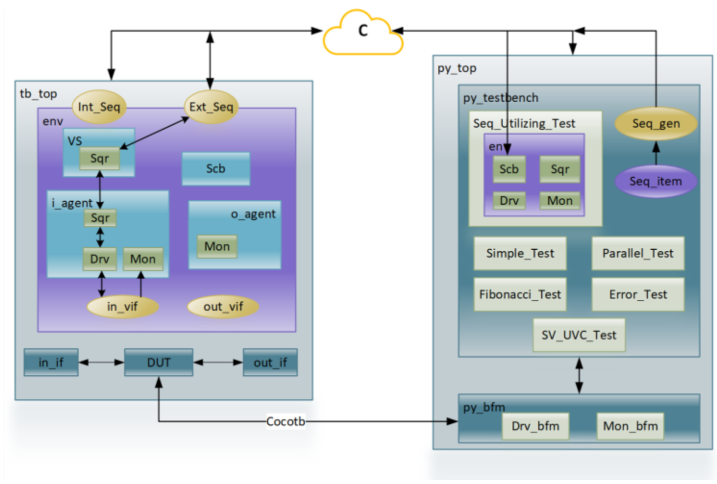

Figure 4. Testbench Framework of SV-VIP utilizing Python-generated sequence

One common concern when using Python, a software programming language, in the hardware world is the simulation wall-clock time. Therefore, in this implementation, paper employs two methods to reduce the delay in transaction transferring. Firstly, the transaction-level data transferring process is dynamic. Figure 5 shows how Python dispatches sequence descriptors to SystemVerilog: the Python testbench continuously sends descriptors to the C language, which are temporarily stored in a shared queue. When the C language receives data from Python, it activates a fork-join on the SystemVerilog side. One part receives the descriptor while the other processes the descriptor, generating the external sequence and sending it to the corresponding sequencer. The C language component includes two queues to continuously receive data from both the Python testbench and the SV-VIP. When the Python testbench dispatches sequence descriptors, the SV-VIP simultaneously sends back the monitored responses to the Python testbench.

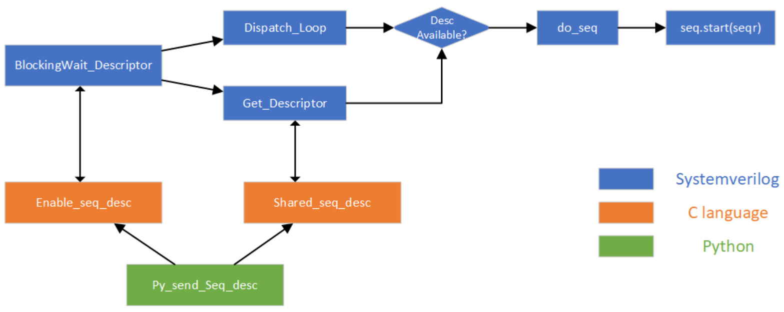

Figure 5. Sequence descriptor dispatch

Additionally, it is important to note that the data type being transferred between Python testbench and SV-VIP is a descriptor. The Python testbench only needs to send the necessary information to the SV-VIP, which then generates the external sequence based on the descriptor content. This eliminates the need to send redundant information to the SV-VIP, thereby optimizing the data transfer delay.

This second implementation illustrates how the Python testbench can freely control and interact with the SV-VIPs, enhancing their tasks. These two implementations not only validate the functionality of the pathway but also illustrate a future verification model where software languages like Python can handle the software aspects of verification, with SV-VIPs acting as the bridge to the hardware. This dual approach leverages the strengths of both domains—Python's flexibility and ease of use in software tasks, and SystemVerilog's specialization in handling hardware interactions.

III. Results Analysis and Encapsulation

Through two implementations, we have demonstrated that the pathway successfully connects the Python testbench and the SV-VIP. The Python testbench can effectively control the SV-VIP while also performing high-level data transfers with it. This capability is crucial for simulation time efficiency, as our tests reveal that the primary factor affecting simulation performance is the communication frequency between Python and SystemVerilog.

Based on our experimental results, we analyzed the combined outcomes of two implementations and the results from pyuvm example tests, as shown in Table I. In the pyuvm example tests (including random_test, parallel_test, fibonacci_test and error_test), where all verification tasks occur within the Python environment, a single transaction data requires 2.98ms of wall-clock time. In the SV-VIP test, where Python controls the SV-VIP and subsequent verification tasks are completed in SystemVerilog, the wall-clock time for a single sequence item is 0.8ms. Lastly, in the test utilizing Python-generated sequences, where Python generates the sequence items, the SV-VIP sends these to the design, and then the response is sent back to Python's scoreboard, the wall-clock time for a single transaction data is 72ms.

Table I. Simulation performance analysis

Test	**Simulation Performance**		
	Total Real Time/ms	***Total # of Sequence Items***	***Time for Single Sequence item/ms***
Pyuvm_Example_Test	170	57	2.98
SV_VIP_Test	240	300	0.8
Utilizing_Python-generated_Sequence_Test	1440	20	72

Although experimental results may be influenced by various factors, the primary impact is clear: frequent interactions between Python and SystemVerilog increase the simulation time greatly. This indirectly supports the feasibility of our pathway, as it enables transaction-level data transferring. In contrast, using existing cocotb's GPI interface, like VPI (Verilog Procedural Interface) or VHPI (VHDL Procedural Interface), would limit the transferring to the signal level, requiring significantly more simulation time. Higher-level data transfer reduces interaction frequency, thus decreasing simulation time while leveraging Python's flexibility in handling complex data as a software language.

Finally, we encapsulated the entire pathway to further enhance its standardization, code portability, and reusability. For the encapsulated package, we utilized virtual class [8] and interface class [7] at the code level to separate user-dependent and user-customizable components. This approach facilitates user integration and simplifies future maintenance and extensions by developers.

We named this pathway BSHL (Bridge-path for SystemVerilog and High-Level Language). This bridge-path is not limited to Python, as it is fundamentally independent of pyuvm and cocotb. We envision it as a language-independent pathway, serving as a window connecting SystemVerilog with any languages that can connect the C language.

The package has been successfully implemented in a practical project at Infineon, demonstrating its applicability and effectiveness in real-world scenarios. The integration diagram is illustrated in Figure 6 and the whole implementation in Infineon’s project demonstrates the package's ease of portability and the functionality of the pathway, making it capable of supporting complex projects.

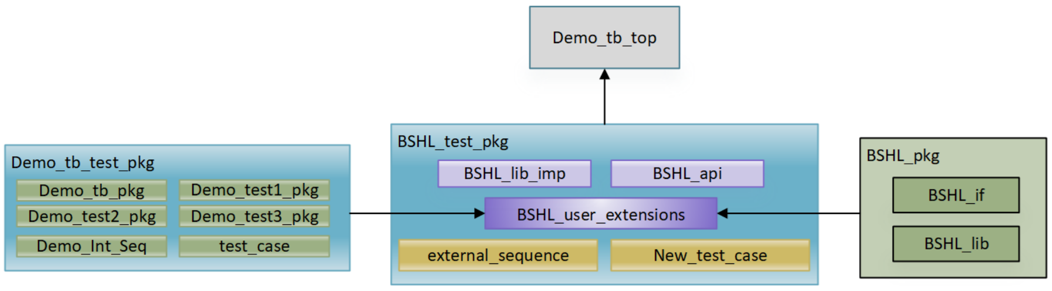

Figure 6. Integration of BSHL in a demo project

To summarize the results, we tested the functionality of the entire pathway with two implementations in both a simple alu-design demo and a practical project at Infineon, and the results proved several points:

- Through this pathway, a Python testbench can control SV-VIPs.
- Python testbench and SV-VIPs can interact at the transaction level through the pathway.
- The encapsulated pathway offers good portability and code reusability.
- Compared to signal-level integration, this pathway requires lower simulation effort.

IV. Future Prediction and Conclusion

For the future research directions of this pathway, we see two main areas of focus. First, further leveraging Python's software characteristics to enhance its utilization of SV-VIPs. This could involve applying machine learning for adaptive sequence generation or optimizing coverage. Second, refining the entire pathway to optimize data transfer delays and enrich the interactions between SV-VIPs and the Python testbench. During the implementation, we recognize that, about Python as a future mainstream verification language, there is still a long way to go. When dealing with complex projects, the scarcity of IP and the slow execution speed of Python can pose challenges. However, we are optimistic about the future of this pathway as it provides a new possibility of involving Python in the verification world more deeply, not just as a scripting language. We believe that Python's simplicity and open-source nature will attract more people to the field and bring a new dynamism to the verification world.

In conclusion, this paper presents a novel method for reusing SystemVerilog VIPs within a Python environment, a capability previously unexplored in research. This approach not only demonstrates the possibility of Python to complement the verification toolbox but also enhances the Python verification ecosystem. Moreover, it provides a conceptual framework for future integrations between Python environments and SystemVerilog, paving the way for more cohesive and efficient verification strategies. This innovative method ensures that Python can continue to expand its role in the field of hardware verification, bridging traditional and modern verification techniques. Finally, this paper predicts future related work and points out a significant issue with using Python as a verification language and its integration with SystemVerilog: simulation performance. However, given the prevailing trend of software-based verification tasks, Python, as a powerful programming language, is certain to have an impact on the field of verification.

References

[1] R. Salemi, "Python for RTL Verification: A Complete Course in Python cocotb and pyuvm", 2021.

[2] Welcome to cocotb’s documentation! cocotb 1.8.1 documentation, [online] Available: https://docs.cocotb.org/en/stable/.

[3] 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), Xi'an, China, 2022, pp. 361-365, doi: 10.1109/ICICM56102.2022.10011364.

[4] M. H. Fayez et al., "Fault simulation Framework using pyuvm," 2023 International Conference on Microelectronics (ICM), Abu Dhabi, United Arab Emirates, 2023, pp. 158-161, doi: 10.1109/ICM60448.2023.10378910.

[5] Qiang Zhang, UVM Combat, Machinery Industry Press, 2014.

[6] Liu, Bin, A Walking Guide to SoC Verification, Publishing House of Electronics Industry, Beijing, 2018.

[7] Sokorac, S., SystemVerilog interface classes – more useful than you thought, DVCon, USA, 2016.

[8] Spear, Chris, System Verilog for Verification, 2 Aufl., Springer Publishing Company, Incorporated, 2008.