2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

# Offloading Complex Mathematical Computations in SystemVerilog Testbenches

## Asyncronous verification of ethernet Reed-Solomon forward error correction using third-party Python packages

Simon Coulter, Cadence Design Systems, Cork, Ireland (*scoulter@cadence.com*)

***Abstract*— Mathematically complex RTL blocks can pose an issue with verification. Applying standard SystemVerilog/UVM verification to mathematically complex RTL blocks is challenging, as developing verification code often requires reimplanting the full mathematical function. This paper will present a methodology for offloading these calculations to a 3rd party software library with established software models that can be used to verify against the Device-Under-Test (DUT) Module. This approach is particularly beneficial when applied to functionality such as Reed-Solomon Forward Error Correction (RS-FEC), where implementing a verification model would consume a disproportionate amount of the design and verification resources. The methodology allows for verification of multiple different DUT elements simultaneously, scales to thousands of parallel tests for large regressions run across a server farm and runs independent of other test frameworks; it can be seamlessly integrated into existing testbenches as plug-in functionality. This solution reduces verification implementation times for mathematically complex functionality without compromising thoroughness or reliability.**

***Keywords—Verification; Mathematically Complex; Direct Programming Interface; TCP; Universal Verification Methodology; Reed-Solomon; Python***

## I. INTRODUCTION

The verification of complex mathematical functions in Register-Transfer-Level (RTL) designs presents a persistent challenge in hardware development. As digital systems increasingly incorporate error correction, cryptography, and signal processing algorithms, traditional verification approaches using independently created validation models in SystemVerilog, using SystemVerilog/UVM testbenches [1] struggle to verify the expected behaviour without duplicating the development effort of the original design.

This is particularly true of functionality such as Reed-Solomon Forward Error Correction (RS-FEC), where the algorithm involves operations in finite fields, polynomial arithmetic and computational algebra, among others. Creating a verification environment for such functionality traditionally requires substantial expertise in both the mathematical domain and hardware description languages, often resulting in verification effort comparable to the original RTL implementation effort.

Several software languages, on the other hand, have existing libraries with robust software models for these functionalities. Notably Python and C/C++ have many freely available packages or libraries (Python: ReedSolo, C: Shifra, libcorrect, etc.) A library such as one of these can be used in conjunction with a standard SystemVerilog/UVM simulation by connecting the applications via a Transmission Control Protocol (TCP) socket, as shown in Figure 1.

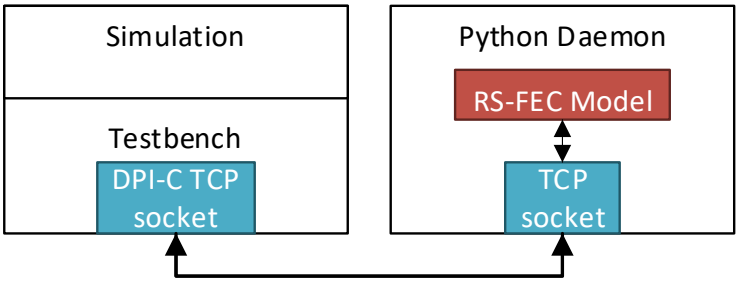


Figure 1. Conceptual high-level architecture

By combining the use of pre-existing software libraries with robust implementations of complex mathematical algorithms and a TCP socket infrastructure designed to handle hundreds or thousands of simultaneous simulation instances, this paper demonstrates a verification approach that significantly reduces the implementation effort required while maintaining confidence in the hardware implementation. This paper will focus on an implementation of this framework for RS-FEC and Python, though its applicability is not limited to this example.

## II. Related Work

There are several existing successful efforts in this space. Previous research [2] demonstrated SystemVerilog's Direct Programming Interface (DPI) functionality for C/C++ integration. However, Python integration requires an additional C intermediary layer, creating differing architectural considerations to this approach. Another verification approach [3] implemented compiled MATLAB executables called directly from the SystemVerilog testbench. A significant limitation of this method is the requirement for the executable to start up and complete execution before simulation can continue, causing increased simulation times. Direct SystemC models through the DPI have been utilised in [4] to simulate complex blocks. While this approach shares the goal of reducing implementation time, it offers less flexibility for verification across multiple toolchains and environments. A TCP socket-based framework similar to this work was developed in [5]. However, their primary objective was co-simulation with FPGA hardware alongside software models, rather than software replacement of verification blocks, as in our approach.

The methodology also differs from frameworks like cocotb, which require the entire testbench to be implemented in Python. Instead, we maintain the existing verification environment while supplementing it with external mathematical models, preserving investment in established infrastructure. The implementation of the methodology discussed in this work is substantially less than that of re-implementation or porting an existing testbench to Python.

While this work shares conceptual similarities with these existing methodologies, it distinguishes itself by providing a framework where any software co-verification package can be seamlessly integrated into an existing verification testbench. It also provides a pathway for an engineer to validate a complex block without having to spend time understanding the complexities of the underlying math, instead focusing functionality, coverage, and more thorough testing. While this work was based on validation using Python, C/C++/SystemC or other languages could be substituted.

## III. Architecture

The verification system can be thought of as having three parts:

1. The Testbench Code, responsible for compiling data to be verified, and packaging it for transmission.
2. The transmission code – consisting of a TCP socket on each side of the link.
3. The verifier application – written in a software language that has a reference library with the complex mathematics already handled, such as Python or C/C++. This paper focuses on a Python implementation.

Furthermore, for this approach to fit seamlessly into existing testbenches, it should:

1. Be scalable to 1000s of simultaneous simulations run in parallel.
2. Be flexible to adaptation for other test needs - including anything that is difficult to verify in a traditional SV/UVM testbench.
3. Not be overly consumptive of server resources – this covers several fronts, including RAM usage, CPU time, server farm “slots”, file/socket handle limits, and ethernet bandwidth.

While some of these, such as ethernet bandwidth within the server farm, are relatively trivial to optimise, others are more challenging.

### A. *Testbench code*

The primary role of the testbench code is to compile data across multiple clock ticks, package it into a structured binary data blob, and send it to the TCP socket. “Compiling” the data in this context refers to packaging a meaningful block of data such as an RS-FEC codeword that passes through a module interface over multiple clock ticks. RS-FEC (544, 514, m=10) for instance, as used in the 802.3 Ethernet Physical Coding Sublayer (PCS) for lane rates of 50G or greater, uses a 5440-bit block of data of 544 10-bit symbols, referred to as a codeword (Figure 2) [6]. The first 514 symbols consist of data, and the remaining 30 are parity bits - the decoding of which is a particularly complex process. Due to its size the codeword arrives over multiple clock cycles, in “data stripes” in the range of hundreds of bits per cycle. The test bench code tracks the start and end of these codewords, captures the data, and holds it in a buffer. It simultaneously tracks the progress of the codeword through the module and captures it on exit again several clock ticks later, after the parity calculations are performed. The two are then packaged together and sent to the socket. As each set of codewords are fully independent from all other codewords, this forms a complete and independent transaction. By taking this approach and storing the codewords in the testbench code until the matching exit codeword and other relevant signals are ready, the task of synchronising the data is greatly simplified.

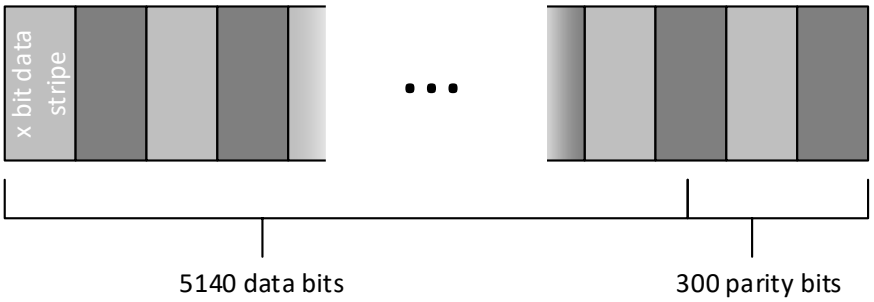


Figure 2. RS-FEC (544, 514) Codeword

The code required to pre-process the data is encapsulated in an “observer” module along with any other required functionality – for example, inverting bit/symbol orders where necessary, or packaging of the data into a byte-stream for the TCP socket. This is shown in Figure 3, where the “observer” module monitors the I/O of the DUT module using a SystemVerilog bind construct. A single simulation instance might have several DUT

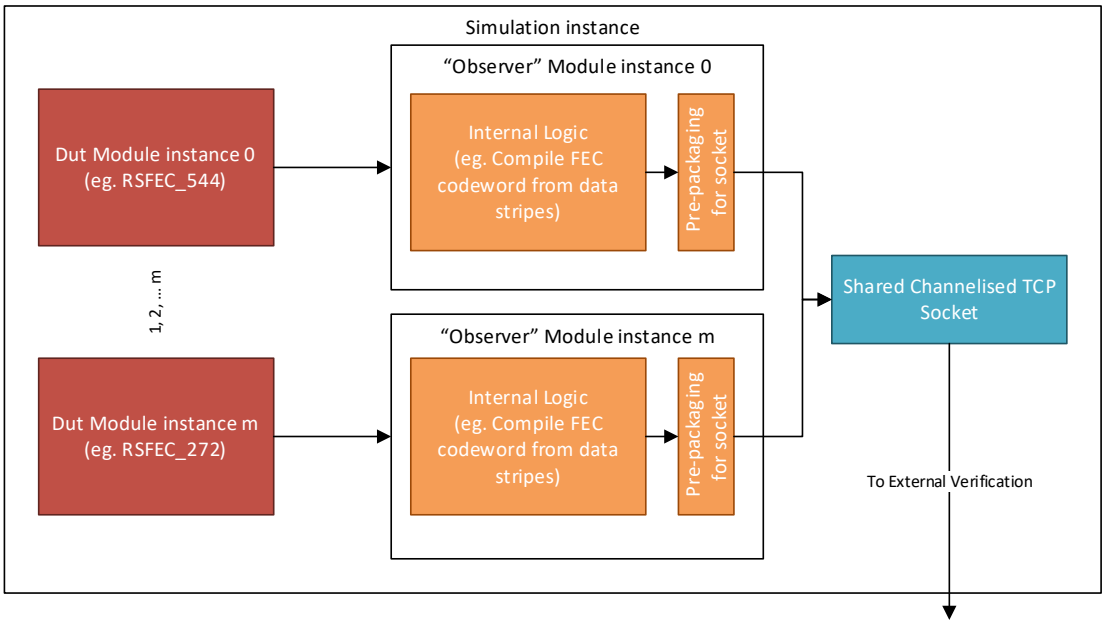


Figure 3. Simulation side infrastructure

modules in need of validation of varying types; for example, different RS-FEC types ((544, 514), (272, 257+1), (528, 514)). Each individual DUT module has its own observer module instance, and typically each *type* of DUT module being observed will require a different type of observer module. The use of the bind construct makes the observer module independent of the testbench driving the stimulus – it can sit adjacent to UVM or pure SV testbenches with equal ease.

When results are returned across the TCP link from the external verification application, they can be interpreted by the observer module, or a separate code unit. Pass/Fail information can be assigned to an assertion or plugged into any testbench as required (UVM, SystemVerilog, or other). Coverage information is gathered for successful comparisons, while failed comparisons can be dumped to debug logs (the simulation timestamp metadata is particularly useful for this). This section of the framework is extremely flexible and can be adapted to the needs of the testbench.

### *B. TCP socket infrastructure*

There are several ways two independent applications can communicate, but IP/TCP sockets provide advantages that cannot easily be replicated. Simple programs might use file handles, reading and writing out to files in known locations – however with a requirement to support hundreds or thousands of simulations simultaneously, a file handle per simulation quickly becomes unwieldy, and a drain on system resources. Additionally, to enable two-way communication, as is necessary in this case for communicating results, either a second file handle or a very carefully orchestrated protocol is required. Local Unix sockets step past much of this and are very efficient. They are bidirectional and can operate on a packet or streaming interface basis, which is ideal for this use case, but are crucially limited to a single physical machine.

IP/TCP sockets on the other hand are limited to one system/server, which is extremely useful in a distributed server farm environment. They are however less efficient than local sockets and operate on a streaming interface basis rather than packet based, requiring extra code to manage. The use of sockets also allows an easy N-to-1 connection, where multiple clients (simulation instances) can connect to a single server (Python, etc.) socket [7].

To reduce the usage of system resources, each simulation instance uses a single socket instance, shared between the observer modules. The packaged data has a standard header prepended to it, including a channel ID that associates it back to its relevant observer module. The packaged data contains other information, such as a simulation timestamp for debug, as well as module signals (modes, rates, clock frequency) in addition to the codeword data. The actual socket code in SystemVerilog must be via a DPI-C call, as there is no native socket implementation; many libraries exist to implement this.

The python application side is implemented using the asyncio package and standard socket calls. The asyncio package [8] allows the application to handle multiple socket connections simultaneously, setting the socket read commands to sleep while waiting for activity. This causes the system to behave as if multi-threaded even when running on a single thread – ideal for this use case, as each pair of codewords is completely independent from all others. Additionally, if the load for a single thread becomes too high to keep up with the influx of data the asyncio package is designed to allow handing off its asynchronous tasks off to subprocesses[1], allowing true parallelisation across multiple cores, with minimum additional implementation effort.

### *C. Python verification application*

The verification app consists of two parts – a TCP server wrapped in an asynchronous handler, and a wrapper around the RS-FEC library. The TCP server has mostly been discussed in the section above – the asynchronous wrapper allows the application to scale easily from a single client to many (Figure 4). If the load becomes so heavy that several subprocesses are necessary, a dispatcher that assigns socket connections to a pool of worker

[1] Python subprocesses are somewhat analogous to threads in most other languages. Threads in Python are in some senses still single threaded, as they must all access the same Global Interpreter, which is single threaded. Subprocesses step around this by spawning a new instance of the Global Interpreter, effectively launching a new instance of Python in a controllable fashion. Multiple “threads” – or asyncio tasks - can run on each subprocess.

threads/processes will also be necessary [9], along with a process-safe communication and management system for managing logging and load balancing.

Once the package of information reaches the library, it must be parsed out of the data blob into its relevant components – codewords, module signals, metadata, etc. For RS-FEC the data package must contain information on what kind of codeword is contained, as the various kinds used in ethernet ((544, 514), (272, 257+1), (528, 514)) all have different data sizes and thus different encoding/decoding algorithms. The library performs the calculations - RSFEC encoding or decoding as relevant – and compares against the DUT calculation, checking the full codewords and the calculated error locations. The result of this comparison is then repackaged and transmitted back across the TCP link. It is accompanied by other information – in the case of a failed comparison, all information can be sent for debug, or for a success, only information necessary for generating coverage data. Results can then be gathered in the observer module and dealt with as needed – for example using assertions and coverage bins, or UVM reporting if desired.

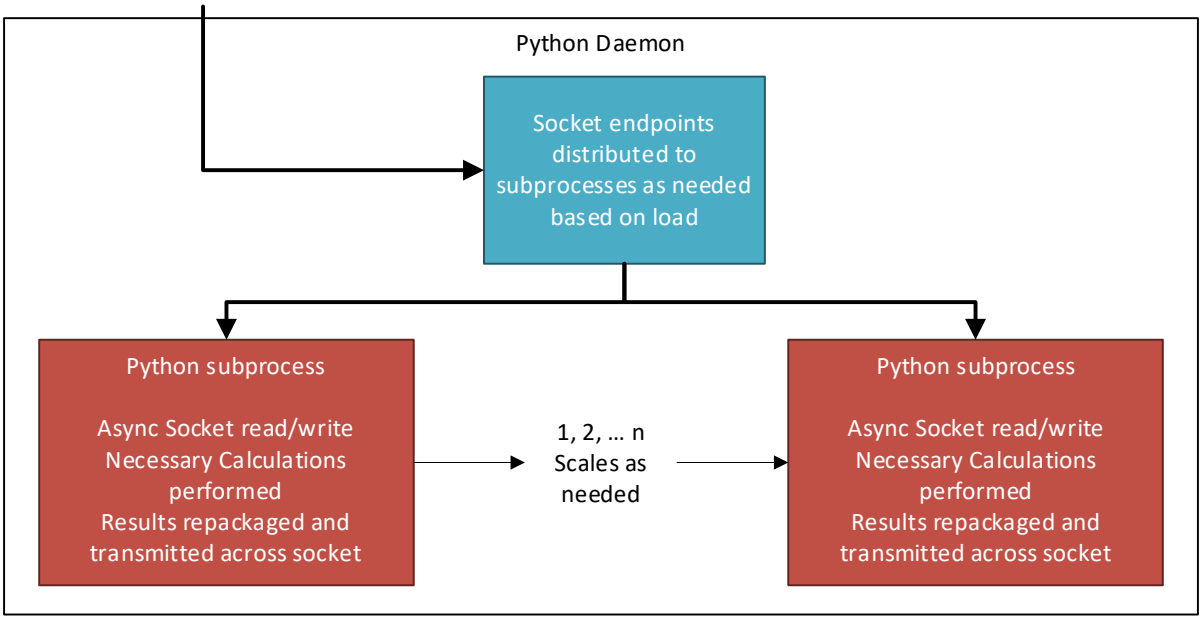


Figure 4. Verification application infrastructure (using Python as an example)

*D. Optimisations - RAM/CPU/Server slot usage*

The least complex implementation of this architecture utilises a single instance of the Python application for each simulation instance; The socket connection is one-to-one, and synchronising the start and end of each simulation is simple. However, each instance of the Python application detailed above uses approximately 300 MB of RAM; multiply this by 1000 simulation instances and a memory footprint of greater than 300 GB is required. Additionally, each Python instance takes up a new server slot. On the other hand, testing revealed that the simulation of the ethernet PCS design, for which this system was developed, produced codewords for verification on the order of hundreds of times slower than the Python application could process and return them. Reversed, this implies that a single Python instance can easily support hundreds of simulation instances, and that the CPU utilisation efficiency per slot increases dramatically. On this basis it made sense to connect multiple simulation instances to a single python instance rather than one-to-one connections, massively reducing RAM footprint and server slot usage.

Figure 5 shows the overall infrastructure of the verification system, with many simulation instances connected to a single external verifier. The external verifier (python in this instance) uses subprocesses (or threads in C/C++) to dynamically allocate resources for high loads. Note that the number of subprocess/threads need not be equal to the number of simulation instances, and is practically bound by the number of available CPU threads.

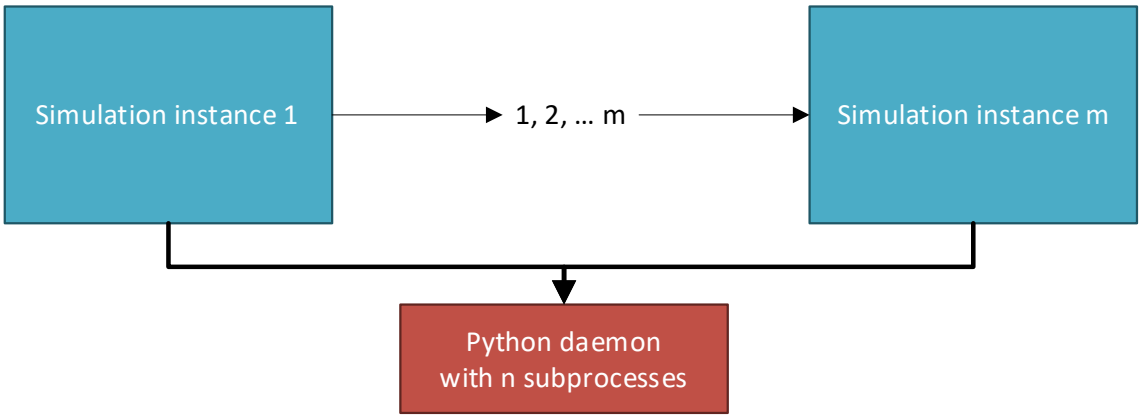


Figure 5. Full infrastructure with one python daemon connected to many simulation instances simultaneously.

### *E. Optimisations – Parallel and Asynchronous processing*

Offloading the RS-FEC verification calculations to Python running on a separate process is an easy source of parallelism, which is highly beneficial in a distributed server farm environment. Another source of optimisation is found in asynchronous sending and receiving of data packages to the verification app; more specifically, the simulation generates a data package, sends it to be verified, and then does not wait for the return message. The simulation continues without getting the results, and some time later, when the verification application is finished processing, the results arrive and can be dealt with. This is particularly important when using a setup with many simulations and one verification app, such as is described in this paper, as it removes a major potential source of bottlenecking. Figure 6 shows a visual representation of this. Note that wall clock time refers to "real time", where sim time is the artificial time represented by the simulations progress. It is worth noting here that for a direct DPI-C call to a C/C++ executable, using this asynchronous and parallel methodology is impractical. Whether this approach would execute faster than the socket calls is unexplored. However, if returning of the results in step with the simulation execution is required, the asynchronous approach described here is certainly less ideal.

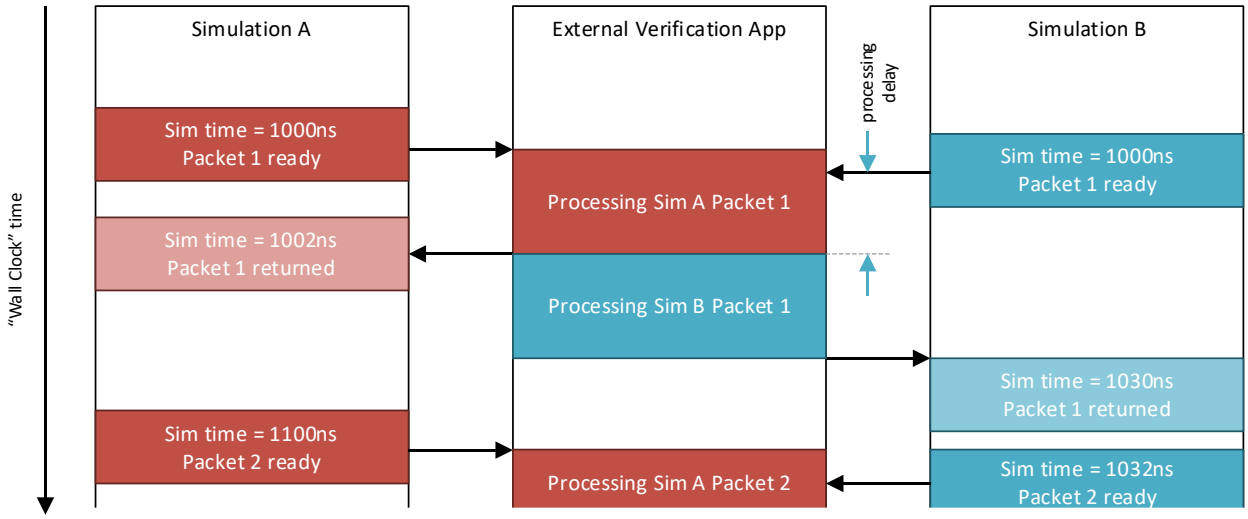


Figure 6. Asynchronous handling of results by the simulation. Processing times are not shown to scale, timestamps are illustrative only

## IV. RESULTS AND DISCUSSION

The proposed methodology was successfully implemented and validated for Reed-Solomon Forward Error Correction (RS-FEC) verification in an ultra-high-speed Ethernet controller design, with three key advantages:

- The addition of the Python based verification framework had no measurable impact on simulation execution time.
- Implementation of the framework did not require any of the in-depth mathematical knowledge need to implement the DUT module – only knowledge of how RS-FEC is used within the Ethernet standard.
- The methodology operates seamlessly as a "plug-in" to existing testbenches without requiring modifications to existing infrastructure.

The system demonstrated robust scalability, successfully supporting greater than 400 concurrent simulation instances connected to a single Python verification daemon running on a single thread without performance degradation. The upper bound of simulations-per-python-instance will be dependant on the nature of the design being simulated – in this case an ethernet PCS. Designs that produce more or less information to validate relative to the design size may have different concurrency limitations, especially after socket overhead is considered. Multi-threading (or multi-processing in Python) was successfully demonstrated to raise this limit as needed, supporting hundreds of simulations per CPU core. This required additional load balancing infrastructure in the application. There is a practical limitation to the number of Python subprocess that can be invoked of one per CPU core on the server in use. As each process can handle hundreds of concurrent connections, this is not expected to be a substantial limitation.

The shared daemon architecture provides significant resource optimisation compared to traditional one-to-one approaches. While a naive implementation would require 300 MB of RAM per simulation instance (totalling 300 GB for 1000 simulations[2]), the shared daemon approach reduces this to a single 300 MB Python instance plus minimal per-simulation overhead.

The addition of the Python based verification framework had no measurable impact on simulation execution time. A minimal overhead is added by the observer modules and TCP socket, but this is dwarfed by the simulation itself. This is particularly significant given that the DUT RS-FEC block itself consumes over 75% of the total simulation time, implying that if the traditional verification approach of implementing an independent model had been used, the total simulation time would have *increased* by ~75%. This could be further extrapolated to imply that the Python based verification represents an approximately 40% saving in execution time over an independent verification model. The asynchronous and parallel processing architecture ensures that the mathematical verification calculations execute without impeding the ongoing simulation, eliminating computational bottlenecks.

Utilisation of this methodology achieved a reduction in verification development effort and complexity. Implementation of the complete RS-FEC verification framework, including the development of the TCP infrastructure, SystemVerilog modules, and Python verification daemon, required substantially fewer man-hours than the original DUT module. This represents a significant improvement over implementation of a full independent model and did not require any of the in-depth mathematical knowledge need to implement the DUT module.

The observer-based architecture using SystemVerilog bind constructs enables seamless integration into existing verification environments. The methodology operates as a zero-impact plug-in to testbench frameworks, successfully interfacing with both UVM and pure SystemVerilog environments without requiring modifications to existing stimulus generation or validation infrastructure. The present validation focuses exclusively on RS-FEC functionality, representing a single-domain case study. While the architectural framework is designed for

[2] In reality this number will be lower due to shared virtual memory pool for libraries. The real number is difficult to calculate or predict, as it requires predetermined knowledge of the number of machines a regression will be split across for a given server farm.

extensibility to other mathematically complex functions, broader validation across different mathematical domains (cryptographic functions, signal processing algorithms, etc.) remains to be demonstrated.

## V. CONCLUSIONS

This paper presents a novel verification methodology that successfully addresses the challenge of verifying mathematically complex RTL blocks through external software library integration. The approach demonstrates three key contributions: (1) significant reduction in verification development effort through reuse of established mathematical libraries, (2) increased confidence in verification accuracy due to the use of third-party golden reference models, (3) excellent scalability characteristics suitable for large-scale regression testing, and (4) plug-in integration with existing verification frameworks.

The RS-FEC case study validates the practical applicability of the approach, achieving complete verification coverage with zero accuracy loss while maintaining simulation performance parity with traditional methods. The demonstrated scalability to over 400 concurrent simulations with optimised resource utilisation makes this methodology particularly suitable for verification environments where development efficiency and regression capacity are critical constraints. While the current validation is limited to a single mathematical example, the architectural framework provides a foundation for broader application to other mathematically complex verification challenges. Any mathematically complex block of RTL is a good candidate for validation using this system; digital/analogue signal processing, encryption, or neural network accelerators are all examples worth considering.

Finally, while this work focussed on the verification domain of simulation, other domains are also worth considering, such as FPGA and emulation validation. The major limitation in moving to these domains will be the use of the SystemVerilog bind construct, which is unlikely to be available when real hardware is in use. Binds were used here to expose an internal module port; exposing the port physically as a configurable part of the design should be used here instead. For full hardware emulation (including the testbench), the data will likely need to be post processed rather than processed live, as in this paper. This remains as future work to be explored.

## REFERENCES

[1] Bergeron J. “Writing testbenches: functional verification of HDL models”. Springer Science & Business Media; Dec 2012

[2] P. Goel, A. Sharma, H.V. Balisetty, ““C” you on the faster side: Accelerating SV DPI based co-simulation”, DVCon United Satates, 2014

[3] S. Aluri, J. Mehta, “Advanced functional verification methodology using UVM for complex DSP algorithms in mixed signal RF SoCs”, DVCon United States, 2014

[4] L. Jinghui, S. Haibo and G. Jiazhen, “Co-simulation platform of SystemC and System-Verilog for algorithm verification”, DVCon China, 2021

[5] A. Papagrigoriou, M.D. Grammatikakis and V. Piperaki, “A hybrid channel for co-simulation of behavioral SystemC IP with its full system prototype on FPGA”, DVCon Europe, 2018

[6] IEEE Standard for Ethernet, "IEEE Std 802.3-2022 - IEEE Standard for Ethernet," Section 91 (Reed-Solomon Forward Error Correction (RS-FEC) sublayer), IEEE, 2022

[7] Van Winkle L. “Hands-On Network Programming with C: Learn socket programming in C and write secure and optimized network code”. Packt Publishing Ltd, May 2019.

[8] Asyncio – Asyncronous I/O for Python, https://docs.python.org/3/library/asyncio.html

[9] Wagner M, Llort G, Mercadal E, Giménez J, Labarta J. “Performance analysis of parallel python applications”. Procedia Computer Science. Jan 2017