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

Comprehensive & Configurable Ethernet IP Verification Strategy

Tom O Connor, Cadence Design Systems, Cork, Ireland (*thomaso@cadence.com*)

Sameh El-Ashry, Cadence Design Systems, Cork, Ireland (*sameha@cadence.com*)

Atif Ansari, Cadence Design Systems, Noida, India (*aatif@cadence.com*)

Vinaykumar Kori, Cadence Design Systems, Noida, India (*vkori@cadence.com*)

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

Afroz Alam, Cadence Design Systems, Noida, India (*afroza@cadence.com*)

Paul Drum, Cadence Design Systems, Dublin, Ireland (*pdrum@cadence.com*)

***Abstract*— Verification signoff of Ethernet Design Intellectual Property (IP) with vast configuration related to Speed, type of Forward Error Correction (FEC) encoding, number of Serdes lanes has considerable challenges. Standalone Universal Verification Methodology (UVM) based verification is not sufficient to ensure comprehensive design validation. This paper presents a comprehensive verification strategy using a combination of UVM, Python-based modeling, and formal verification-based techniques. Leveraging advanced Electronic Design Automation (EDA) verification platforms including AI enhanced debug and formal tools, this work demonstrates how simulation, property checking, and custom reference modeling can be applied to validate complex IPs. The paper highlights how Machine learning (ML) based EDA tools accelerate root-cause analysis and coverage closure, while version-aware workflows ensure maintainable, reproducible regressions across design iterations. Through practical examples, this paper will show how each methodology complements the others, resulting in a scalable, coverage-driven, and bug-resilient verification flow for next generation Ethernet IPs.**

Keywords—UVM; IP Verification; Formal; Configuration; AI; Python

I. INTRODUCTION

Ethernet is a highly configurable IP, supporting a wide range of operating modes. Its design includes complex Finite State Machine (FSMs) and Forward Error Correction (FEC) blocks that require thorough verification across various configuration and corner cases. Due to this inherent complexity, traditional UVM-based verification alone is not sufficient to achieve complete coverage and sign-off.

In modern digital verification environments, no single methodology suffices to capture the full spectrum of functional and corner-case validation. As such, a combination of UVM, formal verification, and Python-based verification has emerged as a powerful and complementary suite of strategies.

UVM remains the industry-standard methodology for constrained-random simulation. It provides a reusable, modular, and scalable verification infrastructure, ideal for stimulus generation, coverage-driven validation, and regression automation. UVM is especially effective for system-level verification, protocol checking, and scoreboard-based functional validation of Ethernet data paths and control modules [1].

Formal verification, in contrast, offers exhaustive, mathematically proven coverage within bounded or unbounded conditions. It is particularly well-suited for verifying protocol properties, data integrity rules, and control logic in critical Ethernet sub-blocks such as Bit Interleaved Parity (BIP) calculation, alignment marker detection, and FEC syndrome generation. Formal verification can systematically uncover bugs in designs with extensive state spaces that would be difficult or impossible to reach through traditional simulation methods. Moreover, formal techniques can augment simulation-based approaches to provide comprehensive design verification [2].

Python-based validation, often used alongside simulation and post-processing flows, provides a high degree of flexibility for golden model development, bit-accurate reference checking, log analysis, and system-level modeling. Python enables rapid prototyping, especially when modeling complex behaviors such as RS-FEC

encoding/decoding, and FEC interleaving. Its integration with SystemVerilog makes it a powerful technique for verifying Ethernet block level verification.

To address these challenges, this work focuses on an automated verification flow that enables the generation of adaptable, automatically generated testbenches tailored to specific configurations. The methodology is complemented by formal techniques to verify the new Ultra Ethernet Consortium (UEC) standard and Ethernet 802.3 blocks considered non-trivial for traditional flows, Python-based modelling for FEC logic and the use of machine learning tools to enable efficient regression debugging and coverage closure. This comprehensive approach ensures robust and scalable verification of the Ethernet IP. Through case studies and practical examples, this paper demonstrates how these strategies work in concert to achieve a high-coverage verification flow for state-of-the-art Ethernet IPs.

II. Related Work

Vintila present a Portable Stimulus-driven verification methodology layered over a SystemVerilog/UVM environment to validate a high-capacity Ethernet communication bridge [3]. Their approach leverages Portable Stimulus Specification (PSS) to abstract and automate complex system-level scenarios, enabling efficient generation of directed and random test cases while minimizing scenario redundancy. While the paper effectively demonstrates the use of PSS for Ethernet-based system-level verification, it does not explore detailed strategies for block-level verification or the integration of formal methods. Moreover, UVM is not the central focus of their methodology but rather a supporting layer. This work is referenced here primarily due to its relevance to Ethernet verification, which aligns with the domain of the current study.

Reference [4] presents a UVM-based verification approach for a Gigabit Ethernet MAC, focusing on validating key functionalities such as frame transmission and reception via the Media Independent Interface (MII). Their methodology emphasizes coverage-driven verification using UVM features like factory and configuration mechanisms, self-checking, and coverage metrics. The authors develop a reusable testbench capable of executing diverse test scenarios within a unified environment, enabling efficient regression testing and improved coverage outcomes.

Chan introduces an open-source UVM Interactive Debug Library designed to significantly reduce debug turnaround time in SystemVerilog-based verification environments [5]. Implemented using the SystemVerilog Direct Programming Interface (SV-DPI), the library enables real-time interaction with the simulation, allowing users to read/write registers, control sequences, and invoke testbench functions without recompilation. While the work offers valuable enhancements to debugging efficiency, it does not address how debug time can be reduced by analyzing regression results or error signatures across regression runs.

There are several existing efforts using either Python or C calls from within a simulation [6, 7, 8, 9]. This work is distinct from each of these in several ways. Most use the Direct Programming Interface (DPI) and are thus limited to C/C++/SystemC, reducing the flexibility of the system. [9] uses Python as this work does, but with a different aim, focused on co-simulation of the Device Under Test (DUT) on an FPGA, where this work specifically seeks to offload computational work from the simulation. The Python validation approach in this work is distinct in its flexibility and ease of implementation.

III. Verification Methodology

Figure 1 depicts a decision tree that encapsulates the primary verification methodologies discussed in this paper. This systematic decision tree serves as a practical guidance framework to help engineers and project teams evaluate and select the most appropriate verification strategies on a per-project basis. Rather than applying a one-size-fits-all approach, this methodology selection process considers project characteristics such as design complexity, configurability requirements, state space constraints, and implementation targets. The decision tree relates to verifying RTL components. In short, the methodologies in section IV should be considered for highly configurable design components, formal verification as detailed in section V is most applicable to RTL blocks with large state spaces, and Python-based validation discussed in section VI is best suited to blocks with complex mathematics

which are time-consuming to implement in SystemVerilog. The Machine learning techniques discussed in section VII are generally applicable to all large test regressions and present a significant opportunity to enhance the efficiency of verification process by unlocking predictive coverage and intelligent analysis.

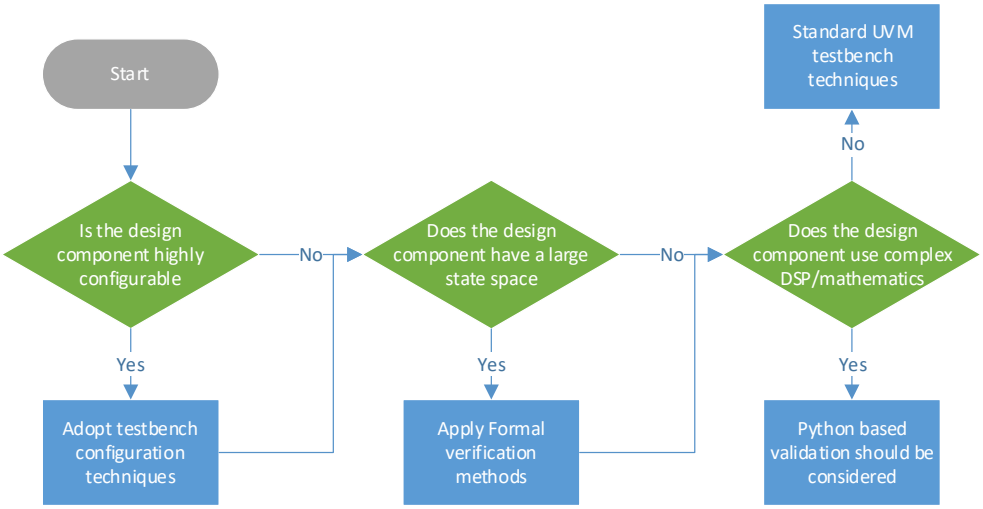

Figure 1: Methodology decision tree

IV. CONFIGURABLE UVM FLOW

Verifying highly configurable high-speed Ethernet controllers-supporting data rates up to 1.6T presents substantial challenges in testbench scalability, integration, and maintainability. The design configuration is divided into presale and post-sale parameters. Configuration features can be removed during synthesis based on the constant presale configuration values.

Traditionally, creating or updating a UVM testbench for each unique Register Transfer Level (RTL) configuration involves substantial manual effort, risking errors and impacting development velocity. To address this, a fully automated UVM-based verification methodology is presented, which bridges the gap between dynamically generated RTL and its corresponding verification environment. Our flow uses a Python parser in conjunction with Jinja2 templates to generate top-level UVM files, parametrized interfaces, and instance bindings that directly reflect the structure of the DUT.

The UVM environment architecture itself is designed to be highly modular and reusable. It includes parametrized agents, monitors, scoreboards, and sequencers that adapt to changing RTL configurations.

The methodology has been successfully applied to a wide range of real-world DUT configurations, covering various rates, lane widths, and protocol layer combinations. Functional coverage was defined at both protocol and configuration levels, ensuring that all possible mode combinations, traffic patterns, and error injection scenarios were exercised. Regression results demonstrate full reuse of UVM agents and scoreboards across configurations, with coverage convergence achieved significantly faster due to the consistency and accuracy of the generated testbench. The environment has been integrated into a Continuous Integration (CI) pipeline, enabling nightly builds with dynamic testbench regeneration for any updated RTL variant. This not only accelerates verification cycles but also ensures that no configuration drift occurs between design and testbench as shown in Figure 2.

The UVM environment and RTL are highly configurable. To illustrate the practical benefit of this flow, consider a verification task involving a 100G PCS-only with two MII ports and a PMA width of 128 bits. Using the automated flow, a single Makefile command generates a matching UVM top-level, including the correct MII agent connections, interface bindings, and VIP instantiations as shown in Figure 3. The same UVM environment that was previously used for full controller verification seamlessly handled this new configuration. All sequences, transmit, receive, and error injection were reused without modification through the virtual sequencer. This not only

eliminated any manual top-level edits, but also ensured first-simulation success and a consistent coverage-driven methodology. The entire setup and bring-up process was completed in under a minute, compared to the estimated two to three days required for a manual equivalent. A more detailed discussion of the complete flow, including template generation, RTL define parsing, and parameterized UVM components for the Ethernet controller, is presented in our companion paper [10].

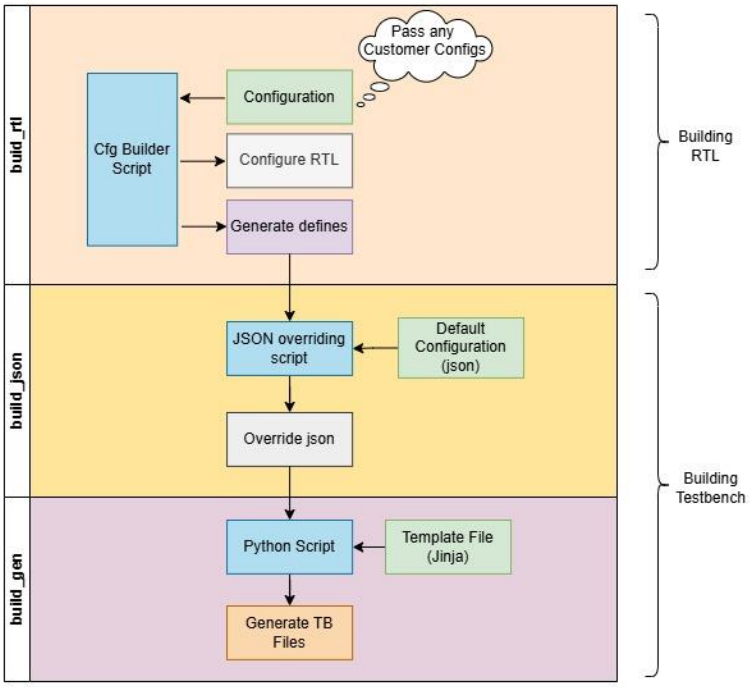

Figure 3: RTL and testbench configuration block diagram

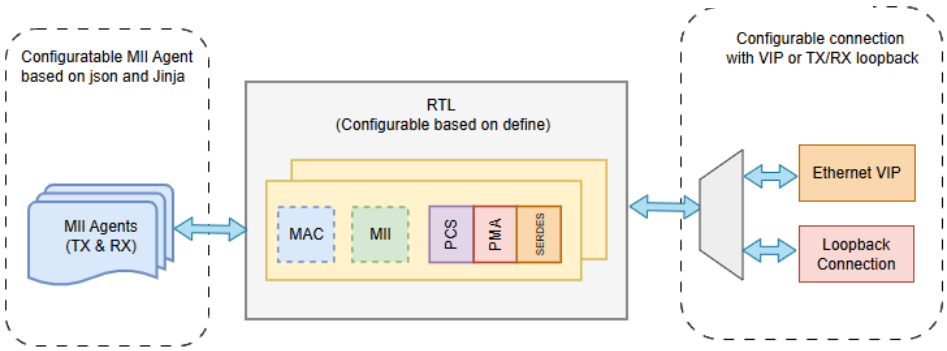

Figure 2: RTL and testbench configuration flow

V. Formal Verification Techniques

This section deals with formal verification of the IEEE 802.3 Ethernet design spanning 25GBASE-R through 1.6T BASE-R and the new Ultra Ethernet Consortium protocol. Using Cadence Jasper in compliment with novel formal techniques which demonstrate an ability to find bugs for a complex full stack design. Using these techniques RTL blocks can be verified and signed off that are typically not considered formal friendly. It is worth noting the context of a configurable design a given formal block level applies the same methodology as the UVM based flow [11][12].

A. Counter Abstraction on RX aligners with FSM State Correlation

An RX aligner is responsible for detecting a Codeword Marker (CWM) at a given point in time and ignores a CWM outside this period which is described in IEEE 802.3 Clause 82 [13]. Traditional formal verification struggles

with Ethernet PCS timing requirements due to large counter periods creating infeasible state spaces. Counter abstraction approach overcomes this by abstracting design counters while maintaining FSM state correlation using a TCL command: **`abstract -counter -env {i_aligner.clk_cnt}`**.

Unlike conventional counter abstraction, this approach maintains a parallel verification FSM that independently tracks expected state transitions based constrained input stimulus, enabling verification of critical timing-dependent functionality while focusing on CWM corruption for Common Marker (CM) and Unique Marker (UM) and predicting if CWM lock should occur. This technique ensures all scenarios with respect to FSM state space were hit without state explosion.

In Figure 4 example, **`clk_count`** will represent real `clk` cycles that would be required to reach an FSM state. The counter value required before an evaluation an FSM state will be known as a critical point. The formal tool automatically detects these. The formal tool can abstract the counter to any value on a given clock cycle. This will be known as abstracted count. In the below example it would take 2000 **`clk_count`** cycles to go from CMP to 2nd CMP. The engine in this instance jumps 2000 **`clk_cnt`** cycles in 3 formal clock ticks or 3 **`abstract_count`** increments. This greatly reduces the clock cycles required to verify the full state space and provides the tool the freedom to add stimulus at any stage between critical points.

B. Advanced CDC Clock Abstraction with Optimized Ratios

Verifying an asynchronous CDC FIFO required clock abstraction techniques to formally prove correct data transfer, pointer synchronization, and empty/full flag generation across independent clock domains. The formal

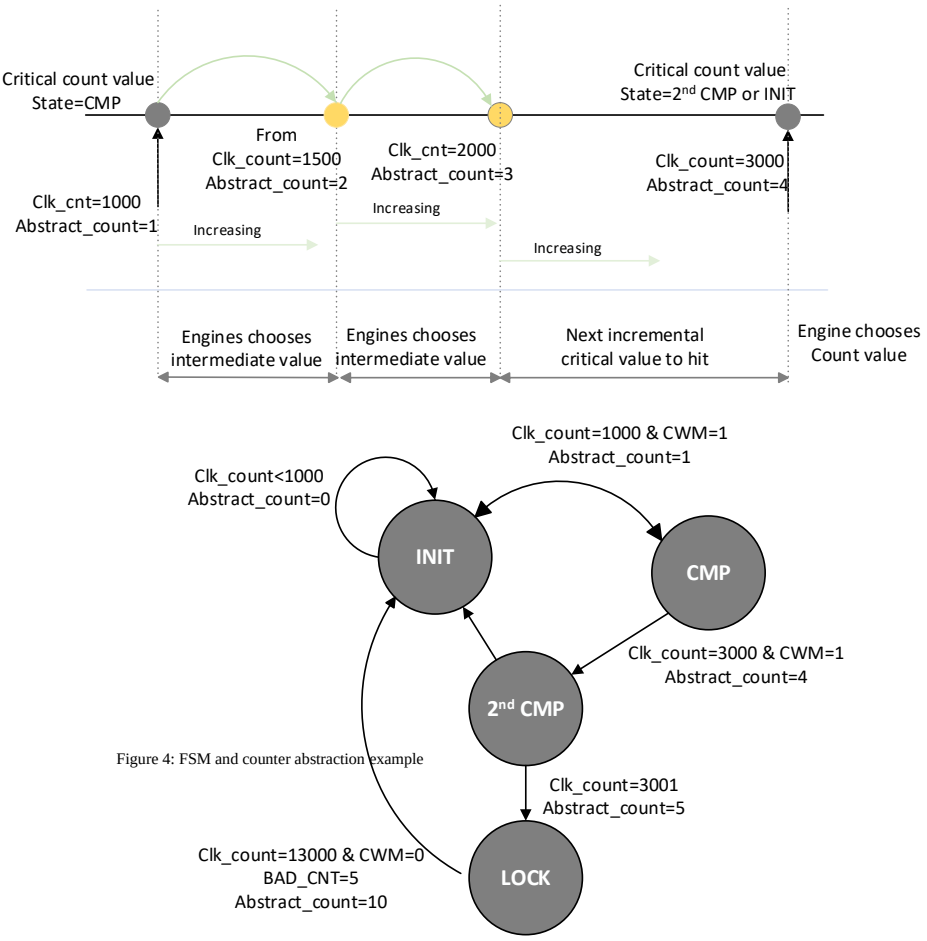

Figure 4: FSM and counter abstraction example

model proceeds in discrete steps under a single global clock. When using asynchronous clocking logic, one will have two real-valued integer clock periods T_1 and T_2 which are based off the global clock. For a given clock period T_1=3 and T_2=2, T_1 runs for 3 global clock ticks while T_2 runs for 2 global clock ticks to mimic asynchronous functionality. Consider an asynchronous FIFO with two clocks: T_1=1.861824 ns and T_2=1.176 ns. This provides a ratio T_1/T_2 = 1.58318367347, which when converted to the exact integer fraction yields T_1/T_2=9697/6125 = 1.58318367347. However, this exact ratio would be infeasible for formal verification due to the large integer values, instead the use practical approximations with acceptable error bounds is chosen:

Table I. Clock accuracy abstraction

Type Approximation	Decimal	Value	Error (%)
Simplest	3/2	1.500000000000	5.25%
Medium	19/12	1.583333333333	0.009%
Most Precise	9697/6125	1.583183673469	0.000%

For formal verification, the medium approximation is selected (19/12) which provides excellent accuracy (<0.01% error) while ensuring a more pessimistic clock relationship approximation. In addition, the pessimistic clock will stress the CDC FIFO further than most precise and therefore negate any risk. This ensures tractable verification runtime without sacrificing precision. The formal tool is now capable to guaranteeing the FIFO will never overflow across any scenario.

C. Verification time reduction using formal

The use formal methods, one can verify an FSM cascaded 8 times with state space of 1,679,616 efficiently and in a matter of minutes. There were four of these in the design.

- TX stated and stateless FSM's
- RX stated and stateless FSM's

Using formal exploration of the full state space has been verified, which is nearly impossible with simulation, formal achieved 100% FSM state coverage with proof of correctness.

D. Hunt & Swarm Protocol Exploration and Proof cache

Jasper's advanced bug hunting modes were used for RX aligners protocol verification. Using Cycle & Bound Swarm Mode one can target critical protocol corner cases while moving forward in formal clock cycles replicating longer time periods seen in simulation. This semi formal approach ensures that one can cover cases that counter abstraction techniques could otherwise miss. To reduce turnaround time (TAT), proof cache is used which records previous formal data and allows complete cover points or examples to be retrieved or continue from where it previously finished. This greatly reduced the time to prove correctness and allowed for faster block sign off.

VI. PYTHON BASED VALIDATION OF THE MATHEMATICALLY COMPLEX RS-FEC

Verification of mathematically complex designs is an increasingly prevalent challenge in modern hardware design, as more applications incorporate signal processing, cryptography, or error correction, such as the Reed-Solomon Forward Error Correction (RS-FEC) for high-speed Ethernet. The RS-FEC decoder of a high-speed Ethernet module typically comprises of greater than 50% of the Physical Coding Sublayer's logic gates; validating this presents a challenge, as the traditional approach would require developing an independent analogous model against which to compare the encoding and decoding. This is a time consuming and difficult task, requiring an engineer proficient in the complex mathematics and algorithms involved in the design in question. However, there are several software languages (Python and C/C++ in particular) that implement libraries that can accurately reproduce these functionalities. Python's ReedSolo was the library chosen for this project. Utilizing these libraries allows for three main benefits:

- Reduction in implementation time of the verification code.
- Increased trust in the reliability of the verification, due to the use of the golden reference models.
- Software models run much faster than a full System Verilog model.

As SystemVerilog has no ability to directly interface with Python (as it does with C/C++), the SystemVerilog simulator must be brought up in parallel with a standalone Python application, and then connected with a standard Transmission Control Protocol (TCP) socket. On the SystemVerilog side, the TCP socket must be implemented using a call to the DPI-C, as there is no native socket function either. This architecture allows multiple simulation instances to simultaneously connect to a single Python process, saving system resources (see Figure 5). Additionally, because of the portability of the socket, it expands the connectivity of the simulation to almost any software language.

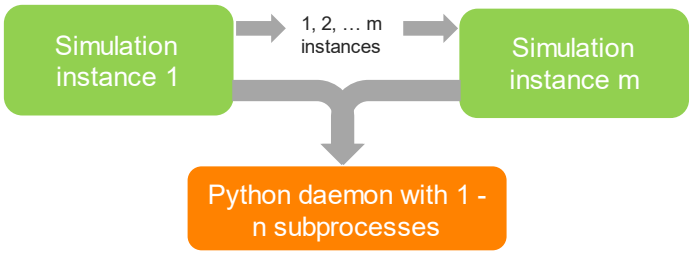

Figure 5 - Full infrastructure with one Python daemon connected to many simulation instances simultaneously

Within the simulation verification code, "observer" modules are added which bind to the DUT modules of interest – the RS-FEC encoders/decoders in this instance. The observer modules compile the data across multiple clock ticks, package it and send it to the socket for verification. They also check the socket for results fed back from the Python application, applying assertions and coverage as relevant. This is performed asynchronously – when the simulation sends a package of data for verification, it continues the simulation without waiting for the results, avoiding introducing any delay due to the RS-FEC verification. Additionally, as the Python application is a standalone application, the verification is fully parallel, running on a different server process. These two factors in combination allow the verification of the RS-FEC to be done with near zero impact on simulation times.

In practice, 400 simulations have been observed connected to a single Python instance simultaneously with zero processing backlog[1]. The Python based verification system was able to validate millions of FEC codewords per regression, contributing coverage, and catching bugs. The development time of this system, while not without challenges, was dramatically shorter than an implementation of a full RS-FEC encoder/decoder and allowed the simulations to run orders of magnitude faster than the FEC modules themselves. It also offers a flexible and extensible platform for future enhancements, especially in cases where native SystemVerilog implementation proves challenging due to language constraints or inherent mathematical complexity. This work presents a verification methodology that incorporates techniques from [14] alongside additional validation approaches. Reference [14] contains the complete technical development of the individual verification methods integrated here.

VII. Machine Learning Based Regression Debug And Coverage Analysis

A. *Efficient Load balancing on server farm queues*

Regression runs & analysis of highly a configurable IP such as Ethernet involves many configuration modes to be regressed and analyzed. This has significant compute resource requirement. Load balancing on server farm queues is managed efficiently using Machine Learning (ML) based regression tool, analyzing the tests run duration and intelligently scheduling them to provide the fastest TAT for regression throughput. Figure 6 depicts an example TAT of 40 units without ML and a TAT of 24 units with ML based load balancing on farm queues. Adopting ML approach in our flow reduced regression TAT by 12%.

B. *ML based Regression failures Triage and Debug*

Considerable verification team effort goes into ensuring regression status is intact while making progress on new features with multiple parallel verification and RTL code check-ins. In a regression there can be:

[1] It is worth nothing that while a Python implementation was chosen here, a C/C++ implementation via a DPI-C call, with no separate daemon, would likely have been substantially more efficient in terms of CPU and memory usage. Python was chosen here due to ease of implementation. Much of the CPU utilisation inefficiency is however counteracted by the use of a standalone daemon.

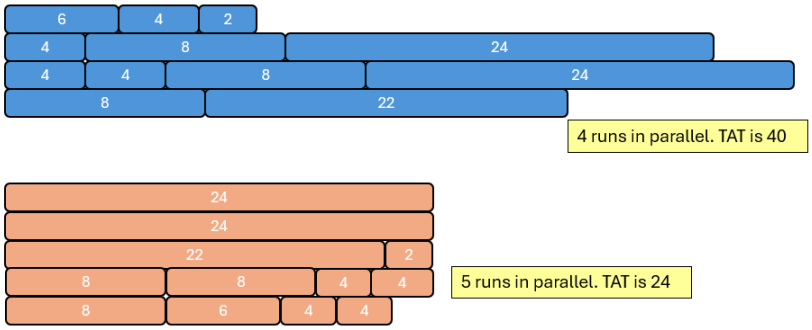

Figure 6 – Farm queue load balancing for fastest turnaround time

- Re-occurring/repeated failures.
- Similarity between failures.
- DUT/Verification IP dependent failures.

Regression Triage is required in the project cycle to focus on the number of new failures or clusters of failures which need attention. This is followed by replicating signatures and using waveform and log files to trace the functional issue responsible for failure.

ML based tools were used for automated clustering of failures and narrowing down the debug scope by finding the functional difference between failing and passing snapshots. These tools are used in the verification cycle for quick and efficient debug. Figure 7 demonstrates the ML-based triage and debug flow.

Figure 7: ML based triage and snapshot comparison flow

C. Fast-tracking Code coverage closure through regression compression.

Achieving comprehensive code coverage during functional verification typically requires running extensive regression suites. Due to the size and configurability of IPs like ethernet, this process involves executing a large number of simulation runs.

To accelerate code coverage closure, ML-based tool is used to perform regression compression which intelligently prioritize a minimal yet representative subset of testcases (with their own set of tuned constraint settings) while retaining the coverage. This approach reduces the regression run time and compute resource requirement. As shown in Figure 8 and Table II, the adoption of the ML-based tool resulted in a 43% reduction in regression run time is observed while maintaining similar code coverage numbers indicating better results as compared to work described in [5].

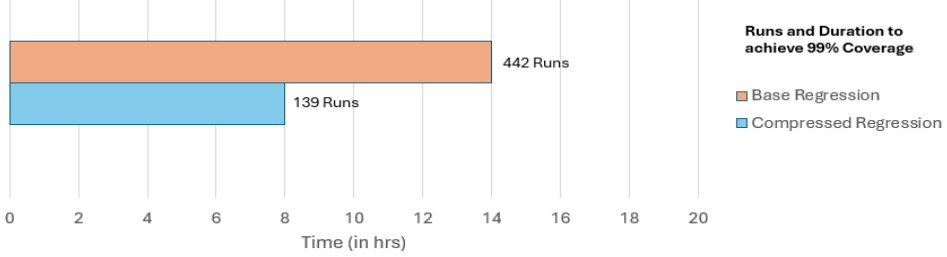

Figure 8: ML based regression compression and coverage optimization

Table II. Compressed vs base regression results

Regression Type	Seed Count	Duration (hrs.)	Code Coverage
Base Regression	442	14	99.73%
Compressed Regression	139	8	99.77%

VIII. CONCLUSION

The proposed flow for the ethernet IP verification leverages a JSON-driven auto-generated testbench framework enabling scalable stimulus generation. Formal verification techniques are employed to exhaustively prove critical FSM paths with a large state space, codeword markers for an aligner and async FIFO's. Additionally, Python-based FEC modelling facilitates accurate cross verification of DUT's FEC functionality against algorithmic golden models. The ultra-high-speed RS-FEC decoding and encoding operations were verified through comparison with a third-party software library implementation. Notably, while the RS-FEC decoder block consumes over 50% of the simulation time, the validation software has no discernable impact on sim time. Integrated use of ML based advanced debug and analysis tools further enhances traceability, root-cause analysis and overall verification efficiency. This approach has been successfully employed in Ethernet IP project, which is highly configurable and involves a complex design framework. The flow is not limited to one specific design or standard. The multifaceted approach outlined can be adapted across any large-scale configurable design. The approach reduces time to revenue by providing faster turnaround for debug with machine learning tools; exhaustively verifies large state space using formal methods and can adapt to a design with 100's of permutations while using the same UVM environment. The deployed methodologies can significantly reduce verification efforts and enable rapid standard compliance verification for evolving standards while ensuring a high verification standard to meet sign off criteria.

REFERENCES

[1] Nagesh, K.A., Shilpa, D.R. (2021). Verification of SerDes Design Using UVM Methodology. In: Nath, V., Mandal, J.K. (eds) Proceeding of Fifth International Conference on Microelectronics, Computing and Communication Systems. Lecture Notes in Electrical Engineering, vol 748. Springer, Singapore.

[2] Asi, A., Jain, A., & Gupta, A. (2024). Formal Verification Approach to Verifying Stream Decoders: Methodology & Findings. Presented at the Design and Verification Conference and Exhibition (DVCon U.S. 2024), San Jose, California.

[3] A. Vintila and I. Tolea, "Portable Stimulus Driven SystemVerilog/UVM Verification Environment for the Verification of a High-Capacity Ethernet Communication Bridge," in Proc. DVCon U.S., 2019.

[4] S. Chitti, P. Chandrasekhar and M. Asha Rani, "Gigabit Ethernet verification using efficient verification methodology," 2015 International Conference on Industrial Instrumentation and Control (ICIC), Pune, India, 2015.

[5] H. Chan, "UVM Interactive Debug Library: Shortening the Debug Turnaround Time," in Proc. DVCon U.S., 2017.

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

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

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

[9] 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.

[10] El-Ashry, Sameh, "A Novel Configurable UVM Architecture To Unlock 1.6T Ethernet Verification", DVCon Europe, 2025.

[11] A. Sullerey, "Design Guidelines for Formal Verification", DVCon United States, 2015

[12] M. Eslinger, N. Tusinschi, "How to Avoid the Pitfalls of Mixing Formal and Simulation Coverage", DVCon United States, 2022

[13] "IEEE Standard for Ethernet," in IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018) , vol., no., pp.1-7025, 29 July 2022, doi: 10.1109/IEEESTD.2022.9844436.

[14] Coulter, Simon, "Offloading Complex Mathematical Computations in SystemVerilog Testbenches", DVCon Europe, 2025.