2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
OCTOBER 14-15, 2025

Comprehensive & Configurable Ethernet IP Verification Strategy

Tom O Connor, Atif Ansari, Sameh El-Ashry, Vinaykumar Kori, Simon Coulter, Afroz Alam, Paul Drum

cadence®

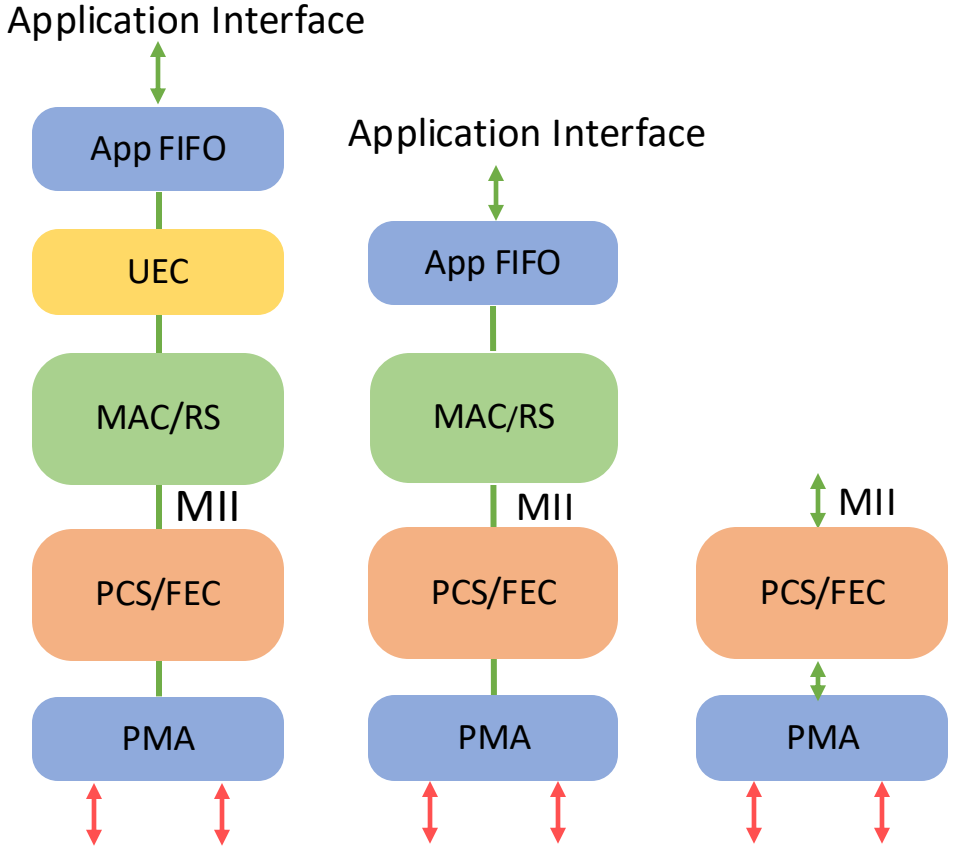
accellera SYSTEMS INITIATIVE

Agenda

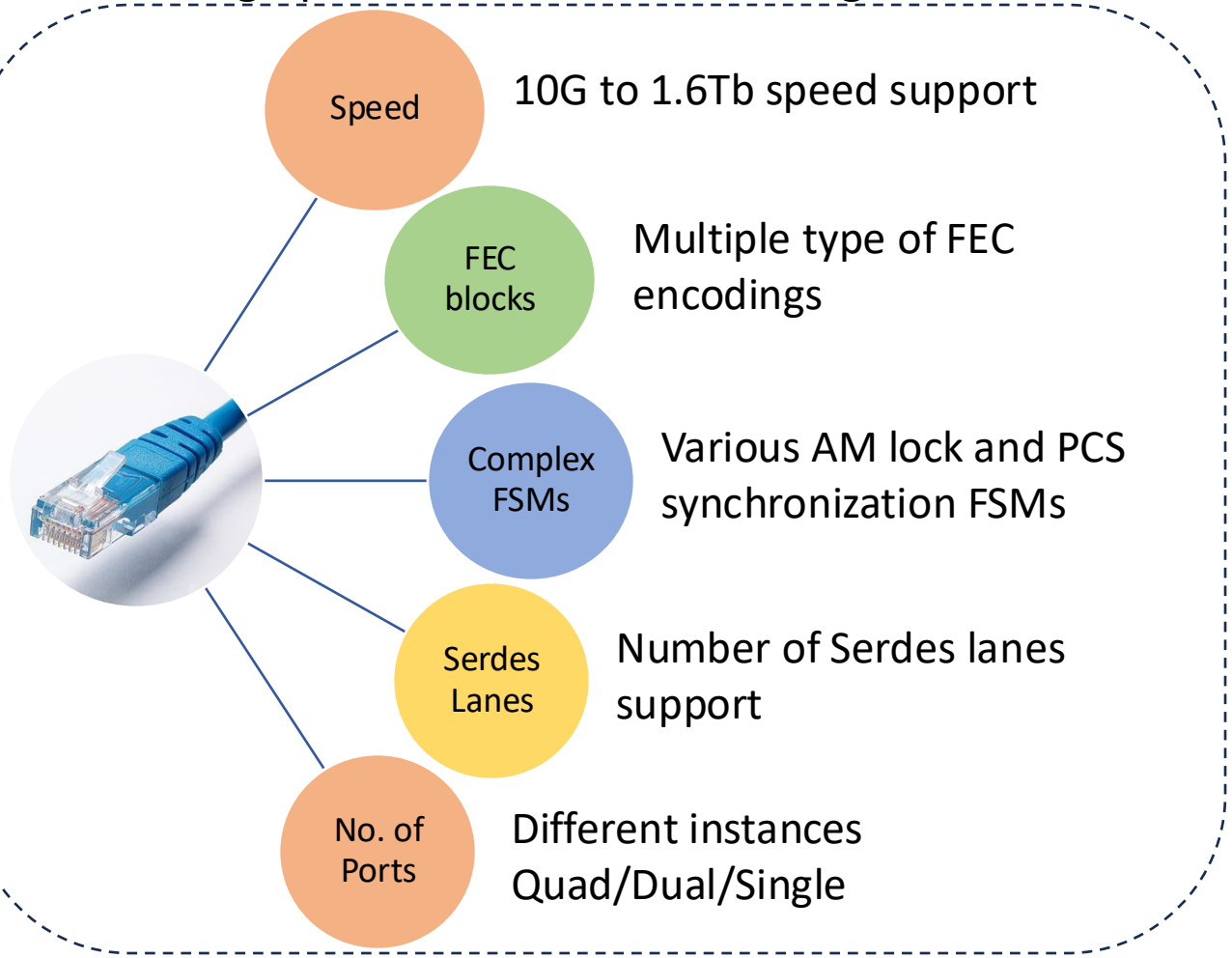
- Introduction
- Verification Methodology
- Formal Verification Techniques
- Python based Validation
  Simon Coulter: Offloading Complex Mathematical Computations in System Verilog Testbenches
- Configurable UVM Testbench Flow
  Sameh El-Ashry: A Novel Configurable UVM Architecture To Unlock 1.6T Ethernet Verification
- Machine Learning Based Regression Debug And Coverage Analysis
- Conclusion

Introduction

Design Configurability

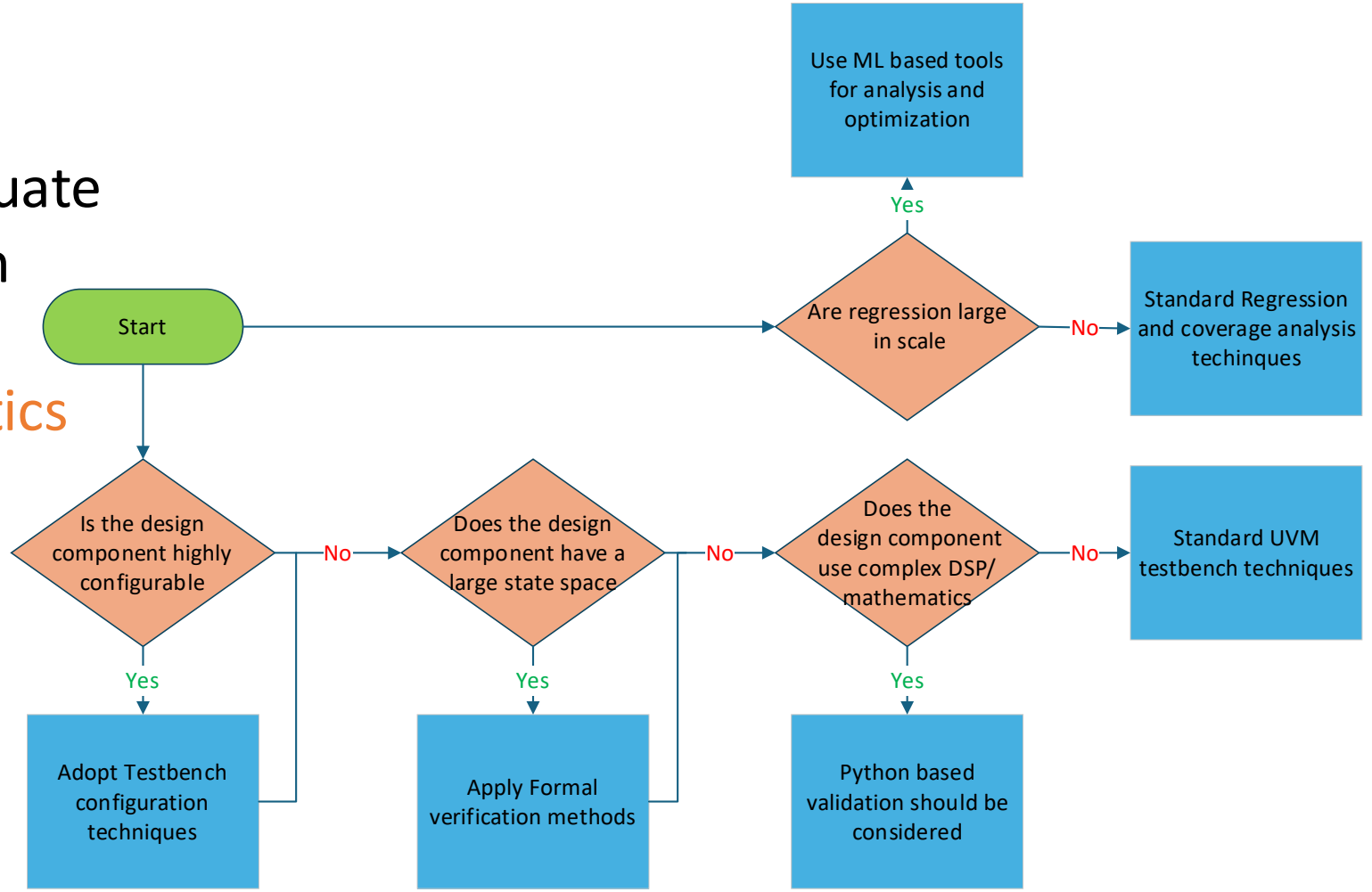


Design permutations within configuration



Verification Methodology

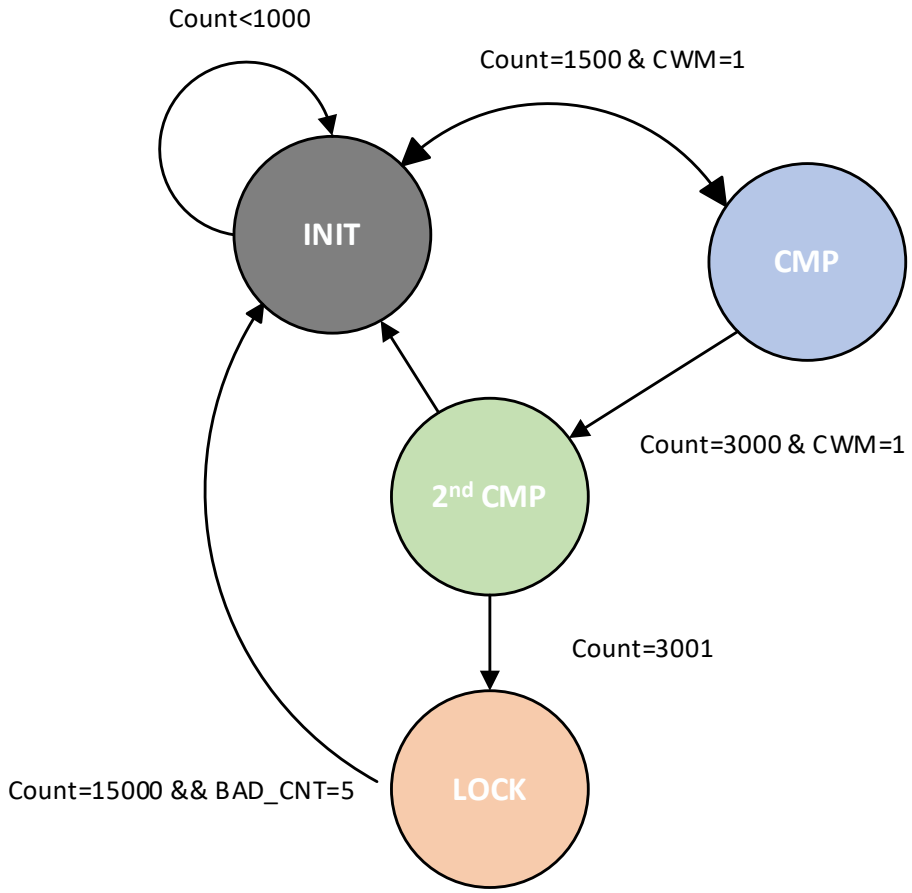
- Guidance framework to evaluate most appropriate verification strategy
- Consider project characteristics such as:
  - Design complexity
  - Configurability
  - State space constraints
  - Implementation targets

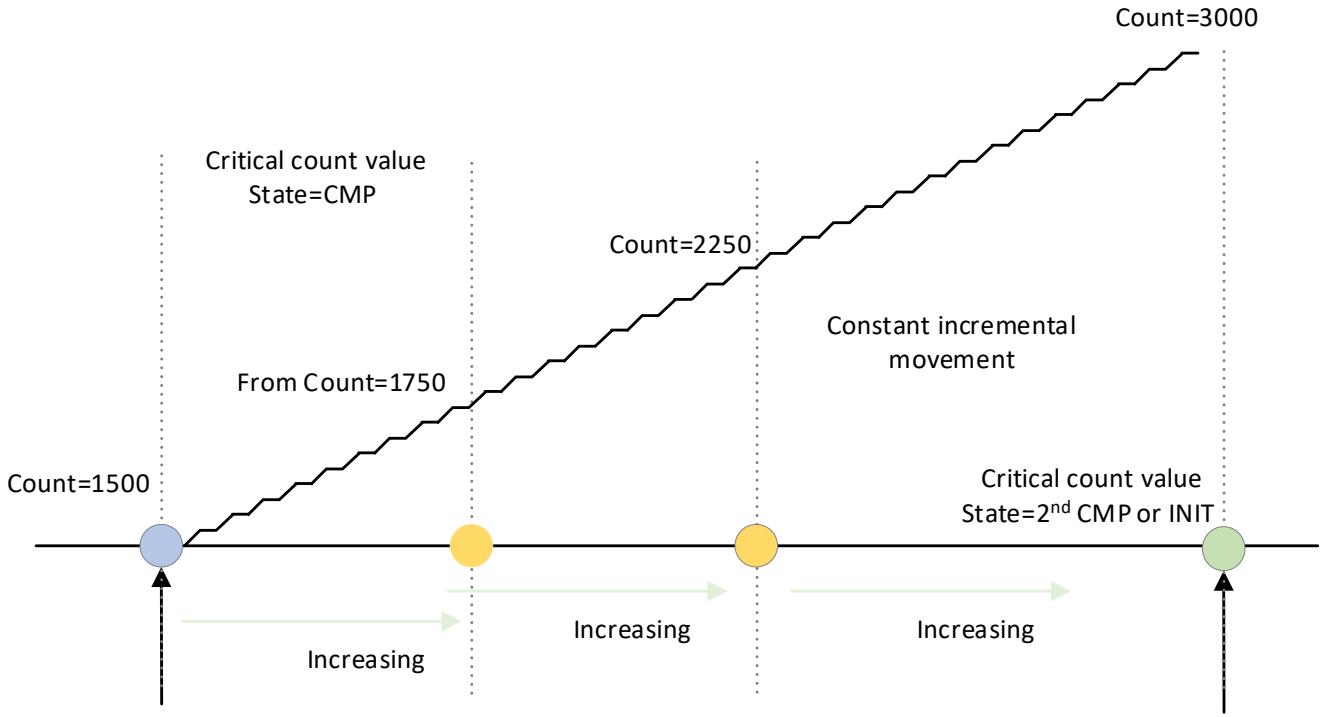


Formal Verification Techniques

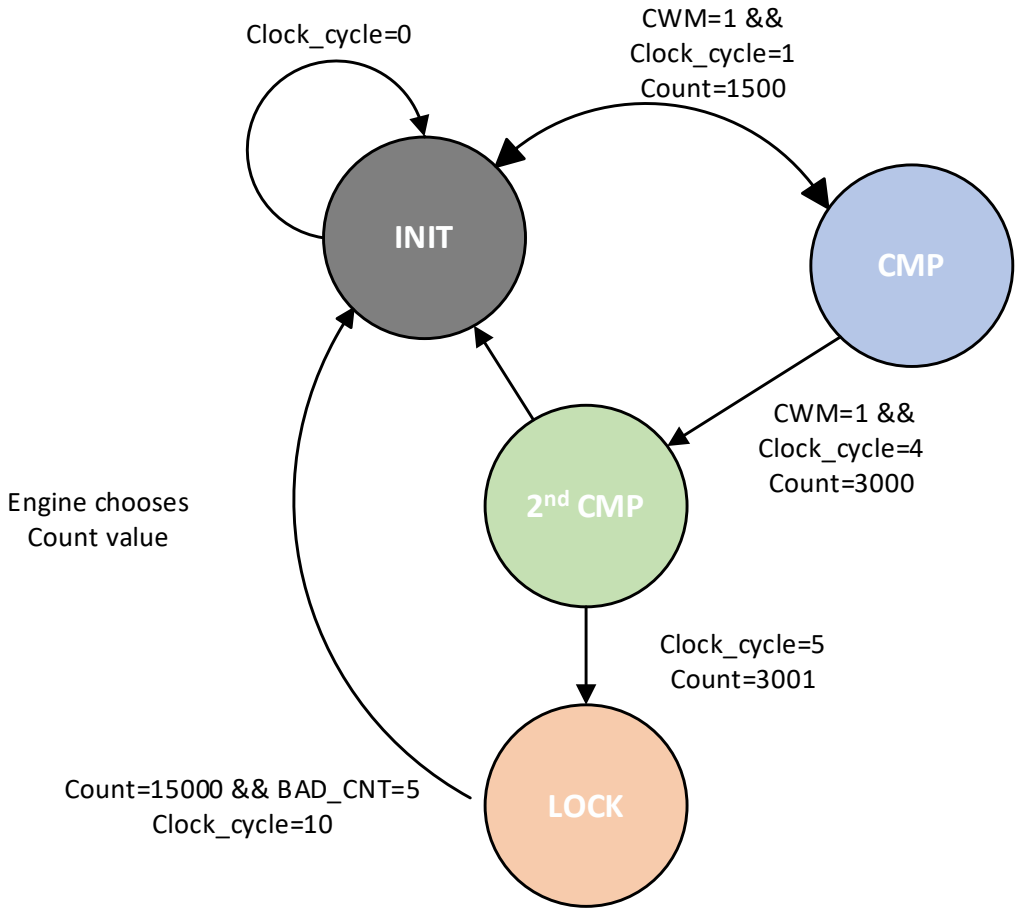
- Extensive state spaces difficult or impossible to reach through simulation
- Formal verification uses proofs to verify the full state space
  - Counter abstraction overcomes state explosion while maintaining FSM state correlation using counters
  - CDC FIFO verification using optimized clock ratios ensures tractable runtime
  - Cycle & bound Swarm modes with proof cache to dramatically reduce turnaround time and adapt semi-formal techniques
  - Best practice formal Verification, 8 cascaded FSM's verified.
    - Achieving 100% state coverage with full proof in minutes

FSM With Counter



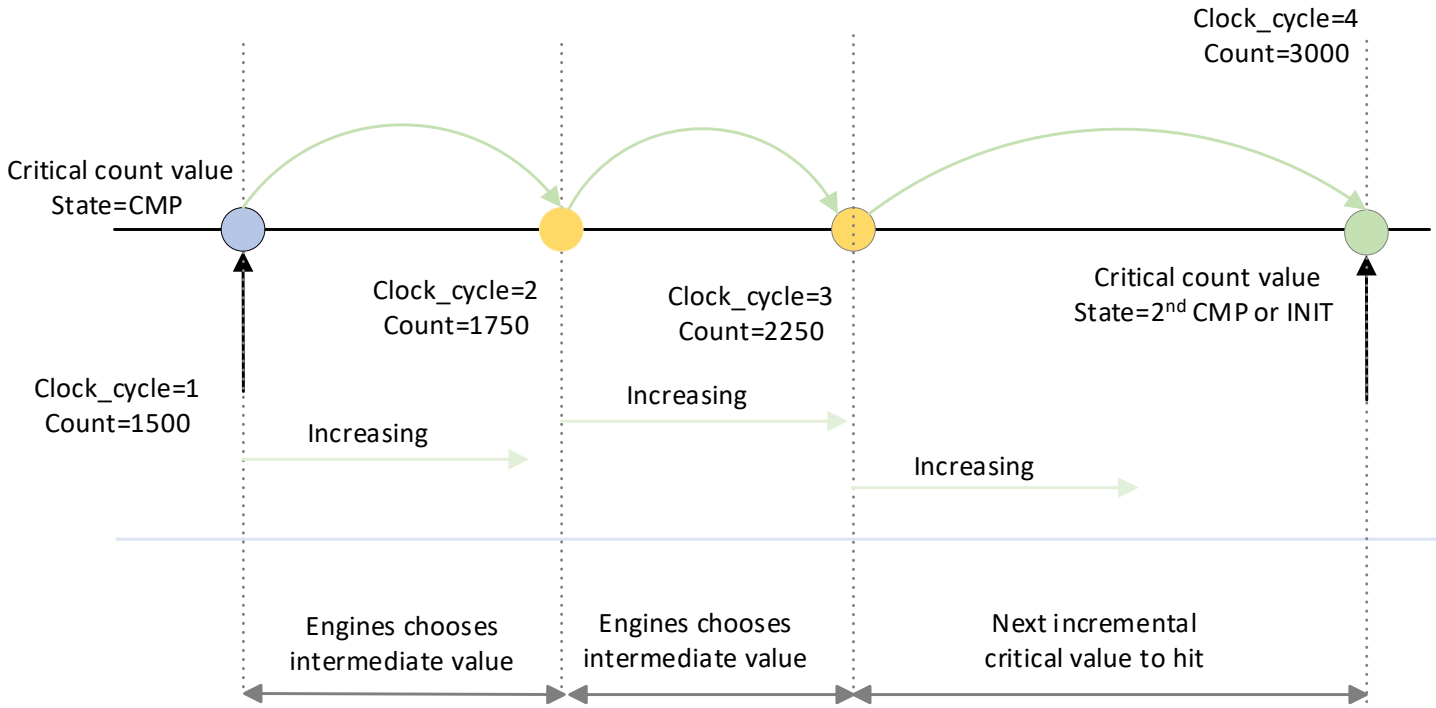


Counter Abstraction



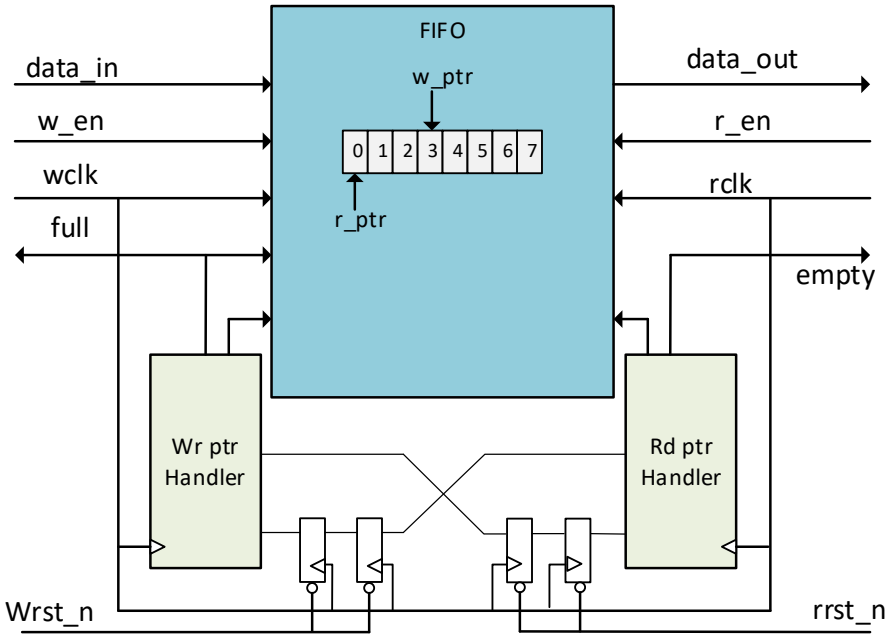
Code example :

```
%abstract -counter -task <embedded> counter_path.cntr
```



Advanced CDC Clock Abstraction

- Asynchronous FIFO
  - Two independent clocks: T_1=1.861824 and T_2=1.176
- Exact integer ratio (9697/6125)
  - Infeasible for formal verification due to large values
- Medium approximation
  - Ensuring tractable verification runtime
  - Pessimistic ratio stresses CDC FIFO beyond real conditions, negating any risk



Type Approximation	Decimal	Value	Error (%)
Simplest	3/2	1.50	5.25%
Medium	19/12	1.583333333333	0.009%
Most Precise	9697/6125	1.583183673469	0.000%

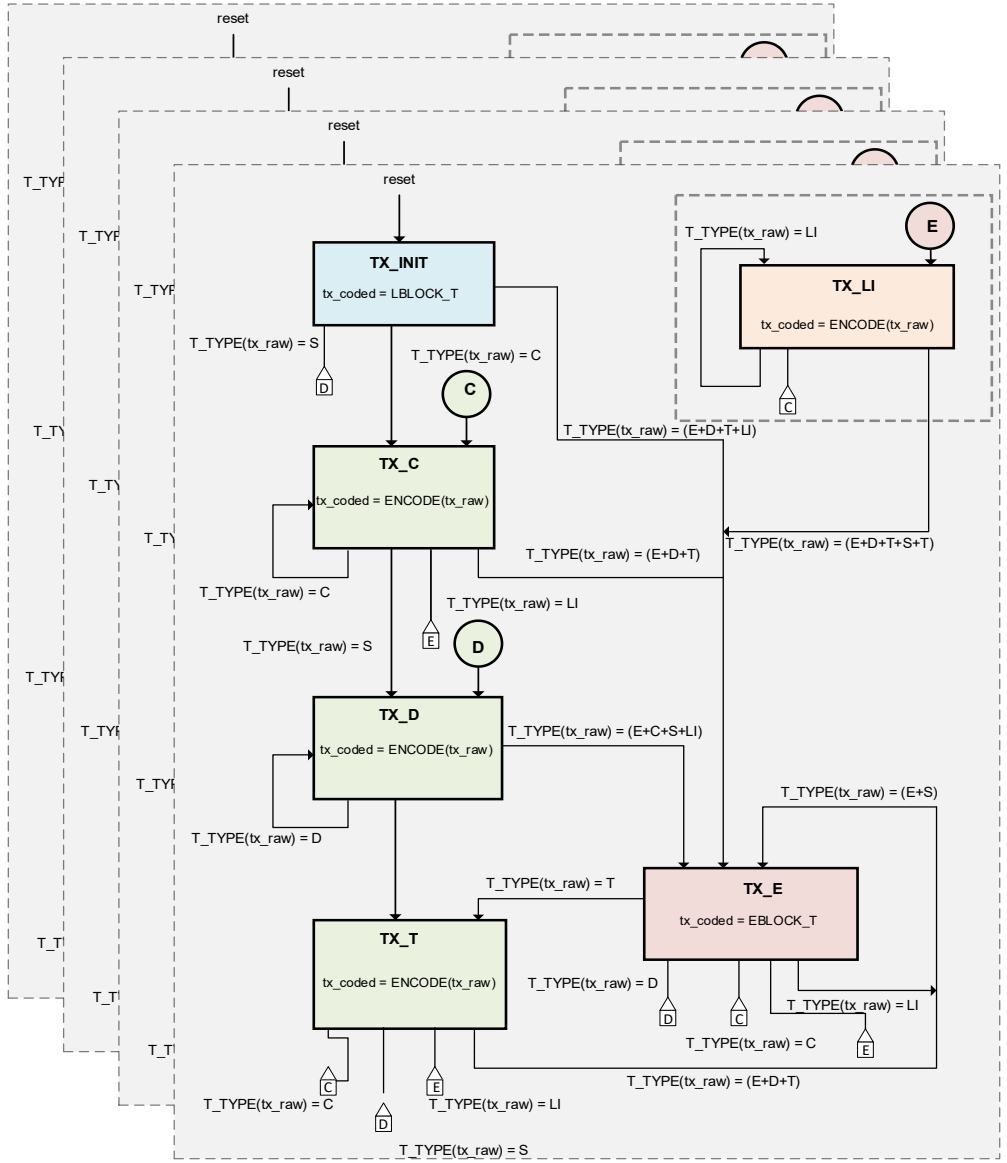
Cycle, Bound Swarm and Proof Cache

- Cycle & Bound Swarm
  - Semi-formal verification - Bridges gap between pure formal and simulation
  - Formal clock advancement - Replicates extended simulation time periods
  - Deep protocol exploration - Uncovers complex multi-cycle protocol interactions

- Proof master : Proof Cache
  - Historical data recording - Stores previous formal verification results
  - Smart continuation - Resumes verification from last checkpoint
  - Cover point retrieval - Quickly accesses cached results
  - TAT reduction - Decreases turnaround time for iterative verification

Formal State Space

- Example formal block :Transmit state
  - IEEE Std 802.3-2022
  - 8 Cascaded FSMs with state space of 1,679,616
  - Formal verification run completes in minutes
- Formal work to date
  - 4 FSM’s: stated and stateless transmit and receive
  - 2 Receive aligners with FSM’s
  - BIP checkers
  - Ultra Ethernet CBFC and LLR design verification
  - Bugs found from Formal (20%)

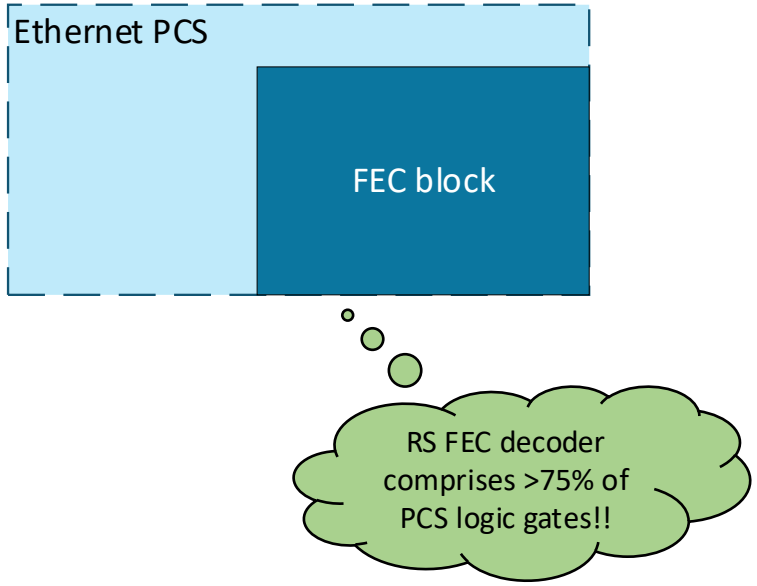


Agenda

- Introduction
- Verification Methodology
- Formal Verification Techniques
- Python based Validation
  - Simon Coulter: Offloading Complex Mathematical Computations in System Verilog Testbenches
- Configurable UVM Testbench Flow
  - Sameh El-Ashry: A Novel Configurable UVM Architecture To Unlock 1.6T Ethernet Verification
- Machine Learning Based Regression Debug And Coverage Analysis
- Conclusion

Python Based Validation

- Traditional RS-FEC verification:
  - Develop an independent analogous model against which to compare the encoding and decoding
  - Requires an engineer proficient in the complex mathematics and algorithms
  - Time consuming and difficult task
- Suggested approach uses software languages such as Python and C/C++
  - Makes use of third-party libraries that accurately reproduce these functionalities

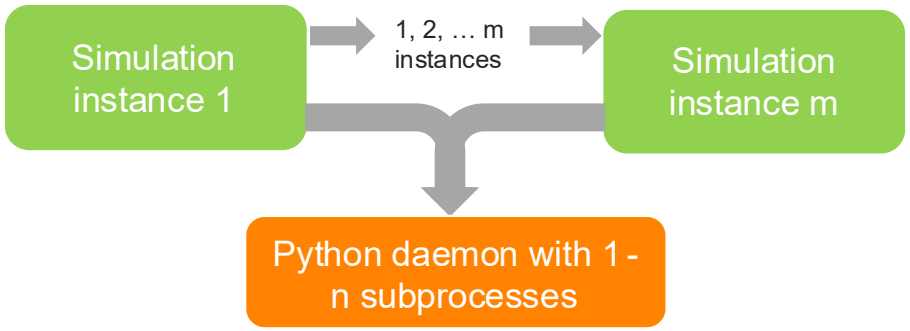


Python Based Validation

- Using Python's reedsolo library allows
  - Reduction in implementation time
  - Reliability, use of the golden reference models
  - Software models run much faster than System Verilog model
- Able to validate millions of FEC codewords per regression, contributing coverage, and catching bugs
- Offers a flexible and extensible platform for future enhancements

Python Implementation/Challenges

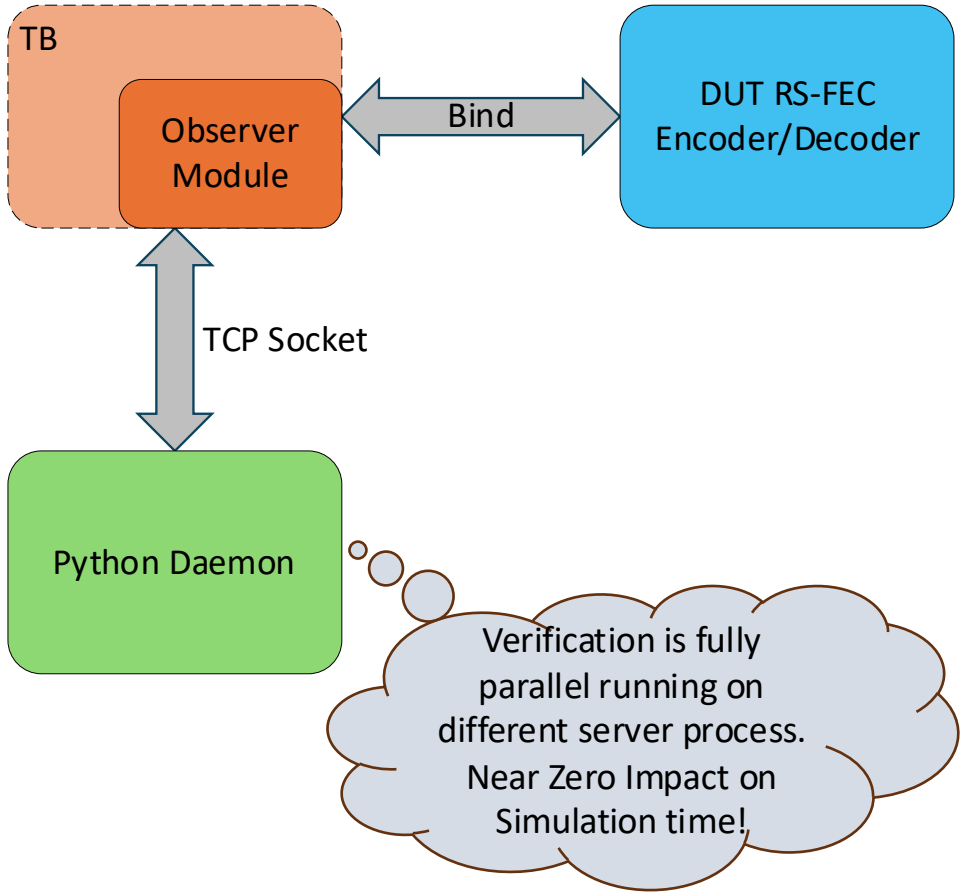
- Simulator is brought up in parallel with a standalone Python application
- Python is connected with a standard Transmission Control Protocol (TCP) socket
- SystemVerilog has no ability to directly interface with Python (as it does with C/C++)
- SystemVerilog TCP socket is implemented using a call to the DPI-C
- Portability of the socket, allows connectivity of the simulation to almost any software language



This architecture allows multiple simulation instances to simultaneously connect to a single Python process, saving system resources

Python Observer modules

- Bind Observer to DUT RS-FEC blocks
- Compile data across multiple clock ticks, package and send it to the socket for verification.
- Check socket for results fed back from the Python application
  - Apply assertions and coverage as relevant.
- Methodology is non-blocking
  - Simulation sends a package of data for verification
  - Simulation continues without waiting for the results
  - Avoids any delay due to the RS-FEC verification.



Agenda

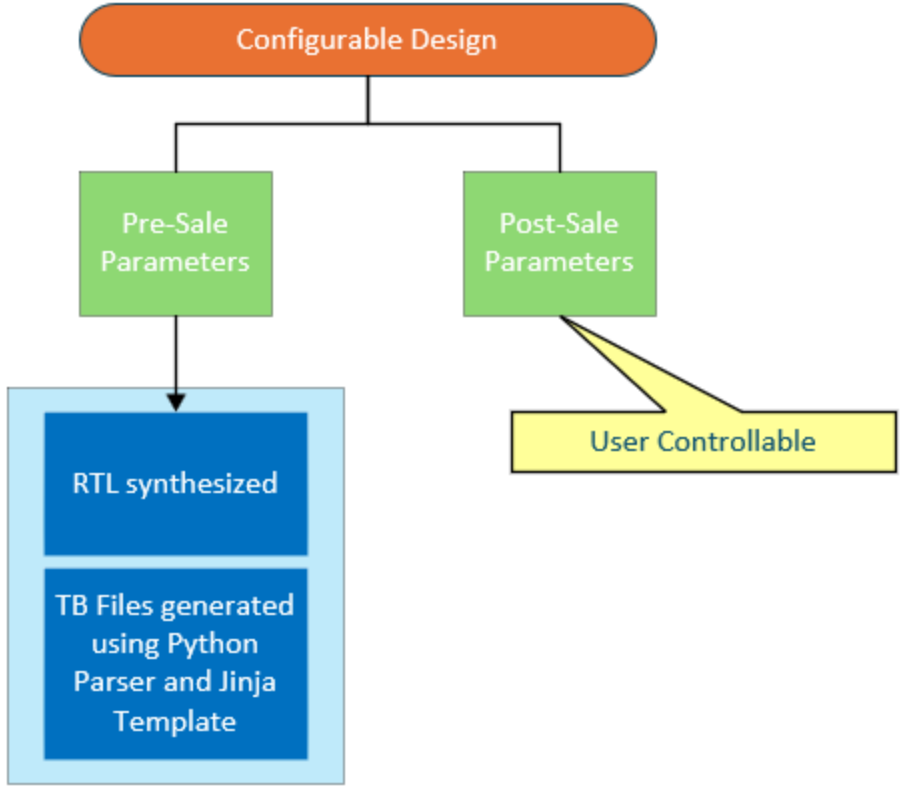
- Introduction
- Verification Methodology
- Formal Verification Techniques
- Python based Validation
  - Simon Coulter: Offloading Complex Mathematical Computations in System Verilog Testbenches
- Configurable UVM Testbench Flow
  - Sameh El-Ashry: A Novel Configurable UVM Architecture To Unlock 1.6T Ethernet Verification
- Machine Learning Based Regression Debug And Coverage Analysis
- Conclusion

Configurable UVM Testbench Flow

- Fully automated UVM-based verification methodology
  - bridges the gap between dynamically generated RTL and its corresponding verification environment
  - accelerates verification cycles while ensuring no configuration drift occurs between design and testbench
  - highly modular and reusable
  - eliminates any manual top-level edits
  - coverage convergence achieved significantly faster due to the consistency and accuracy of the generated testbench

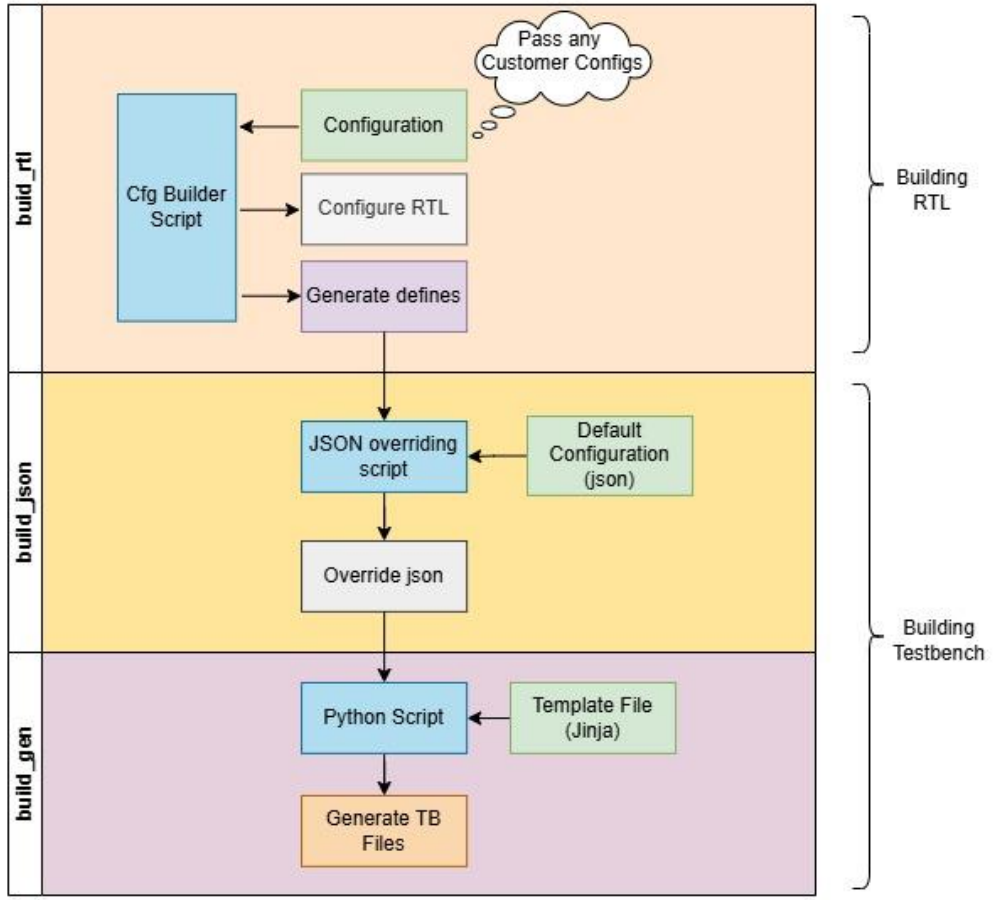
Configurable UVM Testbench Flow

- Design configuration is divided into
  - Pre-sale and
  - Post-sale (User tunable) parameters
- Pre-Sale Parameters
  - Configuration features are removed during synthesis based on these parameters
  - A Python parser is used in conjunction with Jinja2 templates that generates top-level UVM files, parametrized interfaces, and instance bindings that directly reflect the structure of the DUT



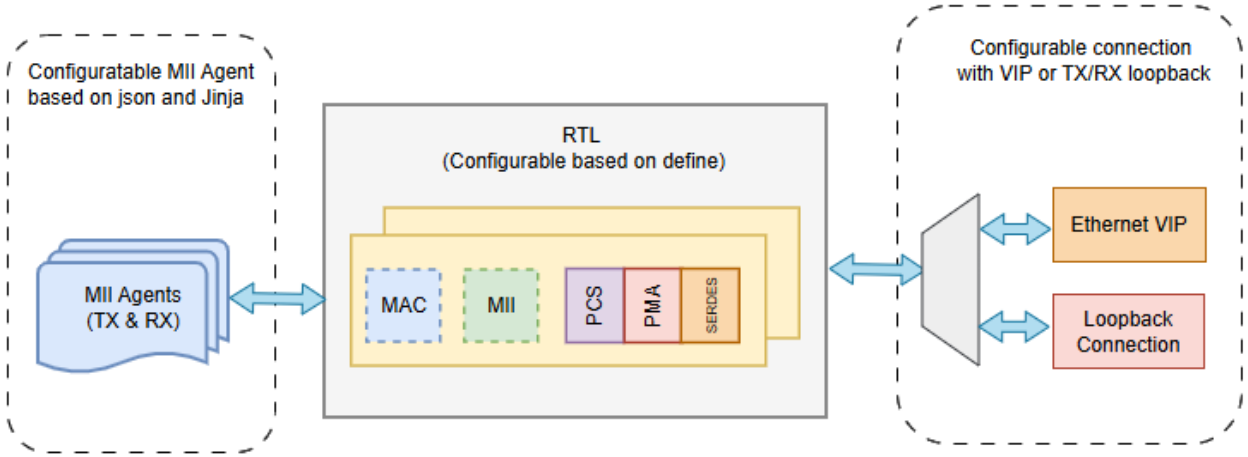
Configurable UVM Testbench Flow

- Build RTL phase:
  - RTL generated according to the config passed
  - Register model created
- Build JSON phase:
  - JSON file updated in consistent to RTL defines
- Build Gen phase:
  - A python script takes JSON file and template file as input and generates Testbench files (Interface, TB Top, DUT configs)



Configurable UVM Testbench Flow

- Configurable agent architecture
  - Application interface agents, MII Agents or Verification IP instances
- Scalable Multi port design
  - Number of port agents are fully configurable
- Flexible connectivity
  - DUT can share interface with multiple heterogeneous agents
  - Supports loopback operation or external VIP driven configuration.



One Environment, Multiple Modes

Agenda

- Introduction
- Verification Methodology
- Formal Verification Techniques
- Python based Validation
  - Simon Coulter: Offloading Complex Mathematical Computations in System Verilog Testbenches
- Configurable UVM Testbench Flow
  - Sameh El-Ashry: A Novel Configurable UVM Architecture To Unlock 1.6T Ethernet Verification
- Machine Learning Based Regression Debug And Coverage Analysis
- Conclusion

Machine Learning Based Regression Debug and Coverage Analysis

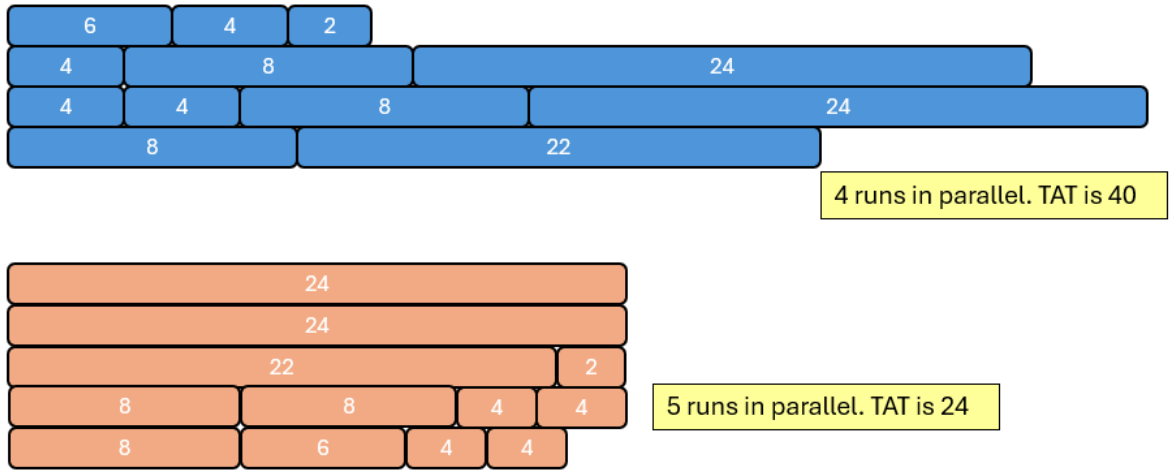
Verification completeness is governed by Regression & Coverage Metrics.

Maintaining regression stability while integrating new features demands significant effort

Extensive coverage demands large regressions – Ethernet IP size and config increases complexity

Machine Learning Based Regression Debug and Coverage Analysis

- Efficient Load balancing on server farm queues using ML-based tool
  - Test run duration is analyzed
  - Intelligently schedule jobs
  - Provide fastest Turn Around Time for regression throughput

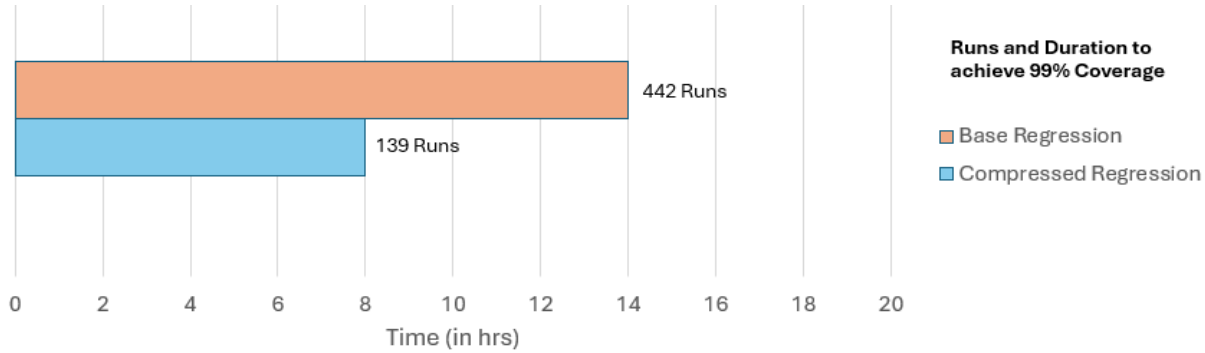


Machine Learning Based Regression Debug and Coverage Analysis

- ML based Regression failures Triage and Debug
  - Automate clustering of failures (Re-occurring, similar, known bugs, etc.)
  - Narrow down the debug scope by finding difference between failing and passing snapshots
  - Helps in quick and efficient debug.

Machine Learning Based Regression Debug and Coverage Analysis

- Code coverage closure through regression compression
  - Intelligently prioritize minimal yet representative subset of testcases
  - Retains coverage
  - Reduces regression runtime and compute resource requirements



43% reduction in regression runtime

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

Conclusion

- A comprehensive strategy for verifying Ethernet IP
  - Flexible framework for verification of other large-scale configurable IPs
  - Improves efficiency and reliability of verification process
  - Ensures higher quality and robustness in diverse system designs
  - Early bug detection and testbench development - Using both Formal verification and Python modelling
  - Reduces time to market for complex designs.

Questions