Deploying AI in DV for Smarter and Faster IP Verification

Mike Bartley, CEO, Alpinum Consulting | Arjumand Yaqoob, Staff Engineer, Qualcomm

Abstract

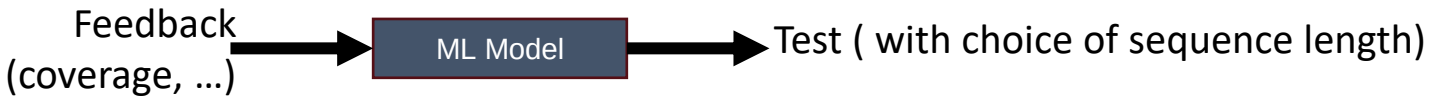
- AI is set to play a key role in optimizing the traditional design verification flows and challenges. Providing a faster and smarter platform to deploy and use in design verification while verifying designs of different complexities. We will be presenting our proposed AI model and strategy. And will apply that to generate a verification environment and Test Bench for an IP design to prove rapid prototyping and efficient verification of designs.

Outline

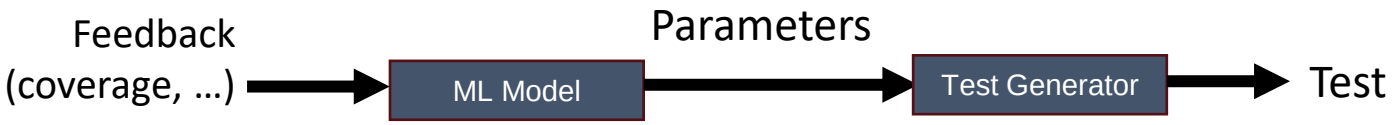
- AI Strategy
- UVM Test Bench Generation
- Results and Analysis
- AI Strategy – Model & Techniques
- AI Strategy – Features Implemented
- Conclusion
- Reference
- Questions

AI Strategy – Inputs

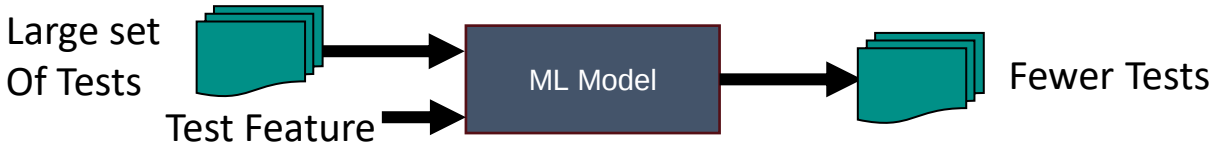
- Test generation, model is trained through various ML techniques



- Test direction the model is trained to direct something else to generate the tests, parameterizing constraint random test generation



- Test selection is based on choosing tests from pre-generated tests on which model is tuned to optimize the selection based on optimization, filtering and prioritizing

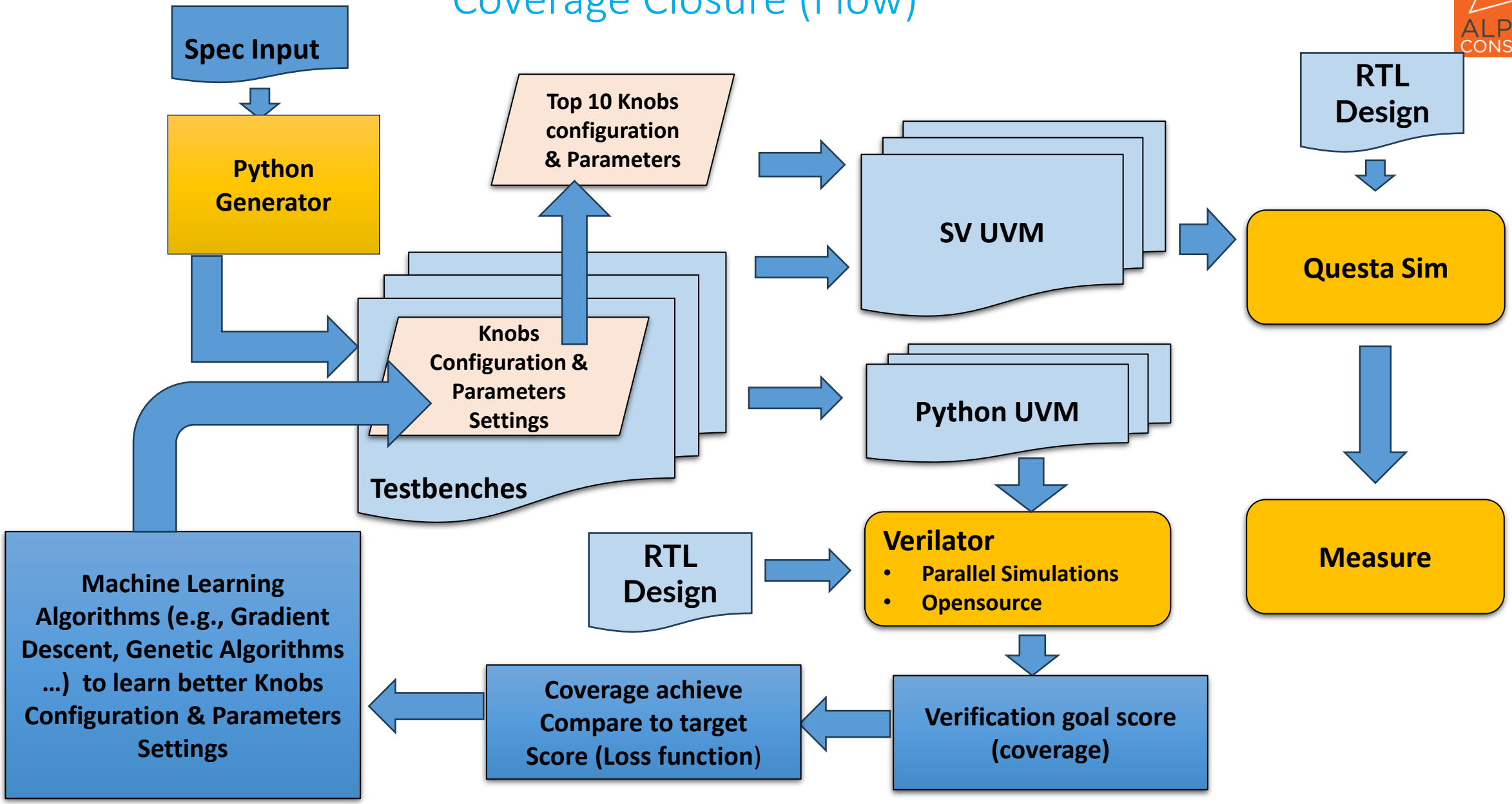


AI – Strategy Coverage Directed Feedback

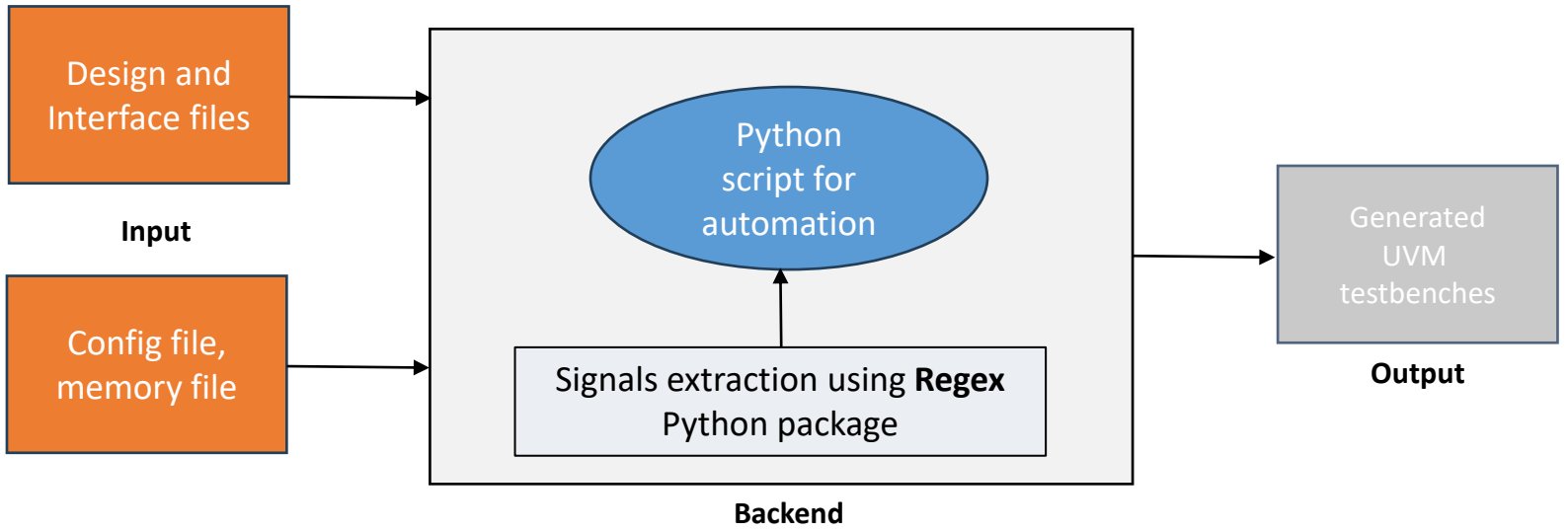
ALPINUM CONSULTING

- Model receive feedback based on coverage score
- Approach based on supervised learning for coverage directed tests selection with novelty driven learn and identify stimulus different from previous
- Biases tests with higher probability of coverage and prioritize those
- Improving coverage and failure prediction
- AI assisted method for coverage feedback selection
- Training set , constraints extractions to assign weights on test scenarios
- Training data optimized the tests selection with higher coverage score

Coverage Closure (Flow)

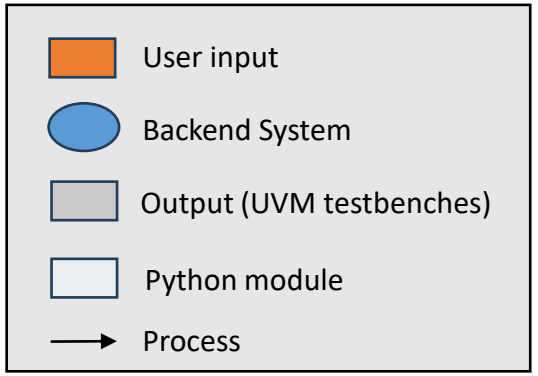


Tool 5: UVM TB Automation with Python scripts



Examples

- **FIFO**
- **Single Port RAM**
- **Dual Port RAM**
- **AXI….**



Input

- User should provide **Design**, **Interface, Config (optional), Memory (optional)** files.

Backend System

- The UVM standard code will be saved in **python script**.
- Python script will extract signals from design and interface files.
- By utilizing the extracted signals, the test bench components gets created.

Output

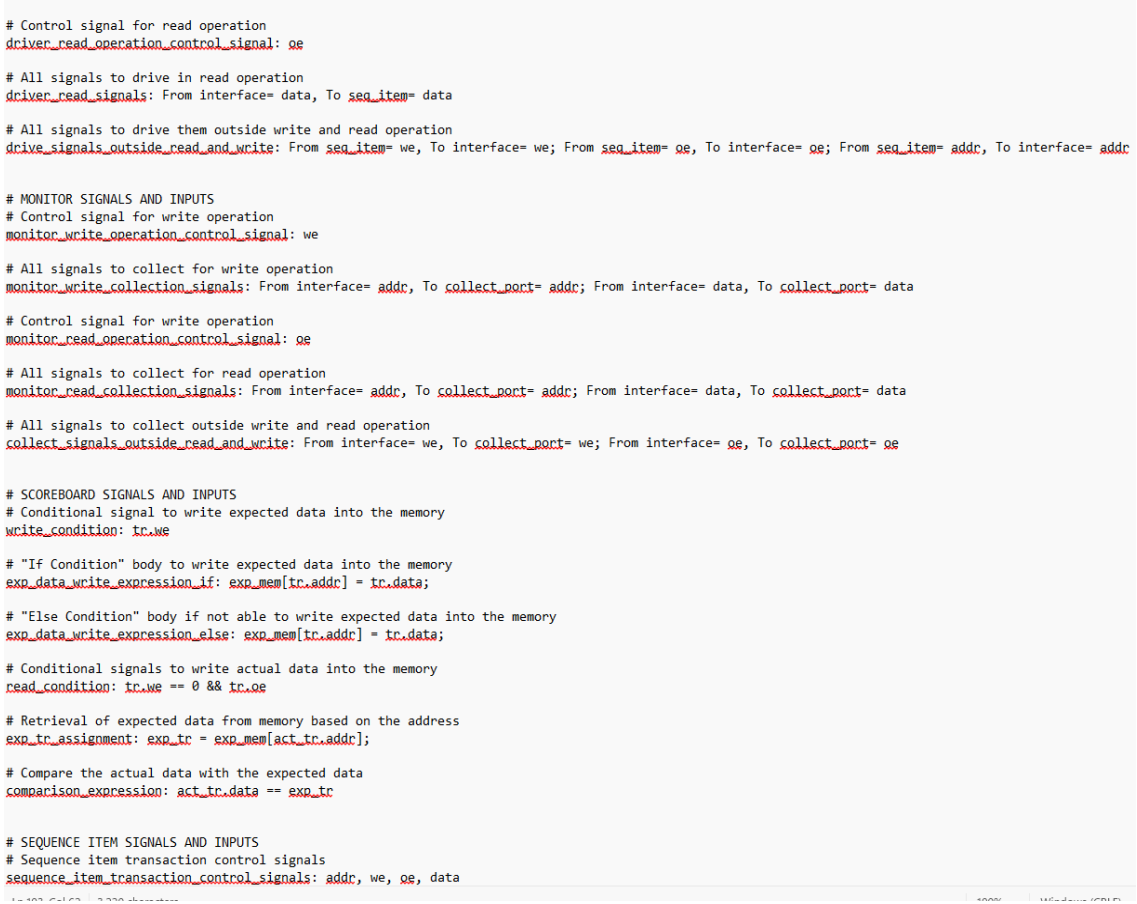
- The UVM testbench components in **System Verilog** format will get saved.

AI for Config files, ML for parameter optimization

UVM Test Bench Generation – Inputs Detail

- **Design I/O, Interface:** Design I/O Top level design I/O and Interface read from a .SV file
- **Reading Configuration Files**: Configuration files specify additional details and information which guides the python script to customize the generated components. Configs are specified in a text file

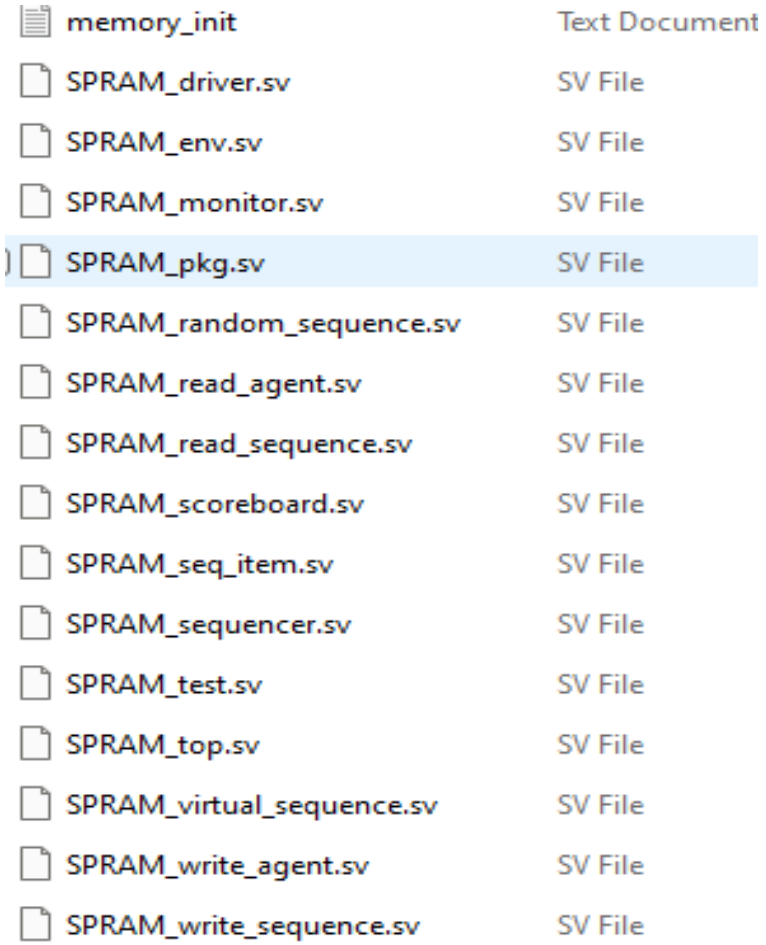
  - Prio , used to provide read and write operation priorities
  - Memory initialization
  - Reset
  - Sequencer info
  - Total number of write/read transactions
  - Chip select
  - Test extension
  - Total number of test extension
  - Interface randomization
  - Resource pool
  - Parameterization
  - Control signals for driver, monitor and scoreboards



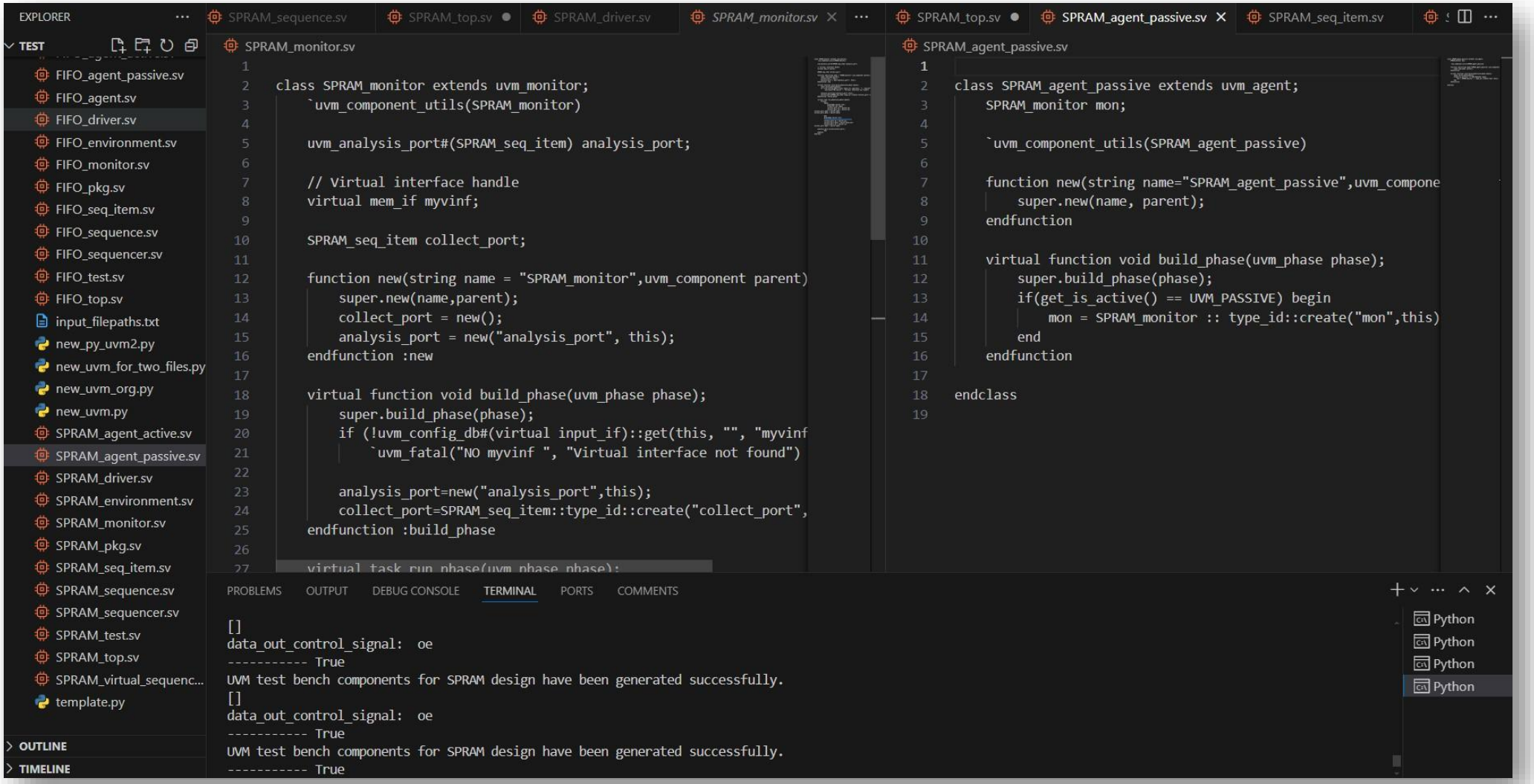
```
# Control signal for read operation
driver_read_operation_control_signal: oe

# All signals to drive in read operation
driver_read_signals: From interface= data, To seq_item= data

# All signals to drive them outside write and read operation
drive_signals_outside_read_and_write: From seq_item= we, To interface= we; From seq_item= oe, To interface= oe; From seq_item= addr, To interface= addr


# MONITOR SIGNALS AND INPUTS
# Control signal for write operation
monitor_write_operation_control_signal: we

# All signals to collect for write operation
monitor_write_collection_signals: From interface= addr, To collect_port= addr; From interface= data, To collect_port= data

# Control signal for write operation
monitor_read_operation_control_signal: oe

# All signals to collect for read operation
monitor_read_collection_signals: From interface= addr, To collect_port= addr; From interface= data, To collect_port= data

# All signals to collect outside write and read operation
collect_signals_outside_read_and_write: From interface= we, To collect_port= we; From interface= oe, To collect_port= oe


# SCOREBOARD SIGNALS AND INPUTS
# Conditional signal to write expected data into the memory
write_condition: tr.we

# "If Condition" body to write expected data into the memory
exp_data_write_expression_if: exp_mem[tr.addr] = tr.data;

# "Else Condition" body if not able to write expected data into the memory
exp_data_write_expression_else: exp_mem[tr.addr] = tr.data;

# Conditional signals to write actual data into the memory
read_condition: tr.we == 0 && tr.oe

# Retrieval of expected data from memory based on the address
exp_tr_assignment: exp_tr = exp_mem[act_tr.addr];

# Compare the actual data with the expected data
comparison_expression: act_tr.data == exp_tr


# SEQUENCE ITEM SIGNALS AND INPUTS
# Sequence item transaction control signals
sequence_item_transaction_control_signals: addr, we, oe, data
```

UVM Test Bench Generation - Output

- UVM TB is generated using the Python Script model
- Complete UVM Verification environment is generated which including sequences and tests
- UVM TB overview for an SPRAM design



UVM Test Bench Generation



Tool 5: : UVM TB Automation - Benefits

- ~90% time reduction in the efforts needed for UVM TB generation
- Higher accuracy
- Consistent process
- Less chance of human errors

Tool 5: UVM TB Automation - Challenges

- New designs may bring unseen challenges

Tool 5: UVM TB Automation - Roadmap

- AI to generate a config file
- Generated TB is entirely controlled by parameters
- Use of AI to optimise parameters
- Switch between different output formats, including Python VUM/CoCoTB, VHDL OSVVM

AI Strategy – Model & Techniques

- Objectives
  - Increase verification efficiency
  - Improve test coverage, bug detection, and debug time
  - Enable intelligent automation in verification using AI/ML
- ML Techniques and Models
  - Supervised Learning:
  - Learn from input-output pairs (e.g., failure patterns)
  - Unsupervised Learning
  - Discover patterns and anomalies in test data
  - Reinforcement Learning
  - Optimize test sequences via reward feedback

AI Strategy – Model & Techniques

- AI – Enhanced Verification Pipeline
  - Input: Test & Random Data → ML Model
  - Predict Failures & Coverage Bins
  - Guide Test Generation, Direction, and Selection
  - Run Simulations
  - Feedback Loop: Update Knowledge Base / Generate New Tests
- Training Methodologies
  - Offline Training:
    - Use historical regression data for initial training
  - Online Training:
    - Incrementally update model after each simulation run
  - Hybrid Training:
    - Bootstrap with offline data, continuously improve online

AI Strategy – Model & Techniques

- Advanced Techniques
  - NLP: Automatic spec extraction & assertion generation
  - Smart Regression: Nearest neighbor algorithm for test reuse
  - GNN: Predict connectivity weights in complex designs
  - AI-driven bug & coverage exposure using adaptive test strategies
- Inputs and Model Training Data
  - Input Layer: Test & Random Stimuli
  - Output Layer: Coverage Bins, Failure Signatures
  - Training Data: Regression logs, connectivity graphs
  - Use for predictive modeling and verification decision-making

AI Strategy – Model & Techniques

- Debug Automation
  - Bug isolation using AI-driven pattern recognition
  - Failure triage with historical signature matching
  - Clustering and root cause prediction using ML models

AI Strategy – Model & Techniques

- Key Features: AI models are set to power up and being used to
  - Power Test optimization
  - Bug predictions & Root cause analysis
  - Post Silicon validation (Detecting Hw anomalies for faster TTM)
  - AI assisted formal verification techniques to develop properties for formal engines
- Challenges: Several challenges are also associated with this
  - Model explainability
  - Scalability
  - Integration with legacy system
  - Verification of AI HW

AI Model – Features Implemented

- UVM TB Automation – Input Configuration and Parameter Optimisation
  - AI and ML techniques to infer and generate the required configurations from a design
- This enhanced AI strategy based on an ML model helped create the complete ECO system for the Verification Environment
- Enhancement to the current AI strategy and model can be added based on more advanced ML techniques that can add value in several ways
  - Bug isolation
  - Test plan creation
  - Bug prediction and root cause analysis
  - Creation of a high-level Reference model from design I/O and design files to be used in scoreboards
  - Constraint optimisation to generate different constraints dynamically
  - Debugging and Triaging

Conclusion

- We are Set to reduce manual time and efforts in TB implementations and building verification environment
- The proposed model and infrastructure can be augmented with more AI assisted tools to generate TB features i.e. Assertions
- AI engine is used for feature extraction and coverage tunning
- Automation is used for generate TB, UVCs , sequences and tests
- This model can be further extended to fine tune using Machine Learning based on the analysis of following data
  - Simulation results
  - Debug data
  - Regressions data

References

[1] Bartley, M., Soni, M., C Tessolve (2024) . AI strategy for DV Flow & TB AI Tool. https://www.tessolve.com