TEXAS A&M UNIVERSITY

Accelerating Functional Verification with Machine Learning

Survey and Applications

Outline

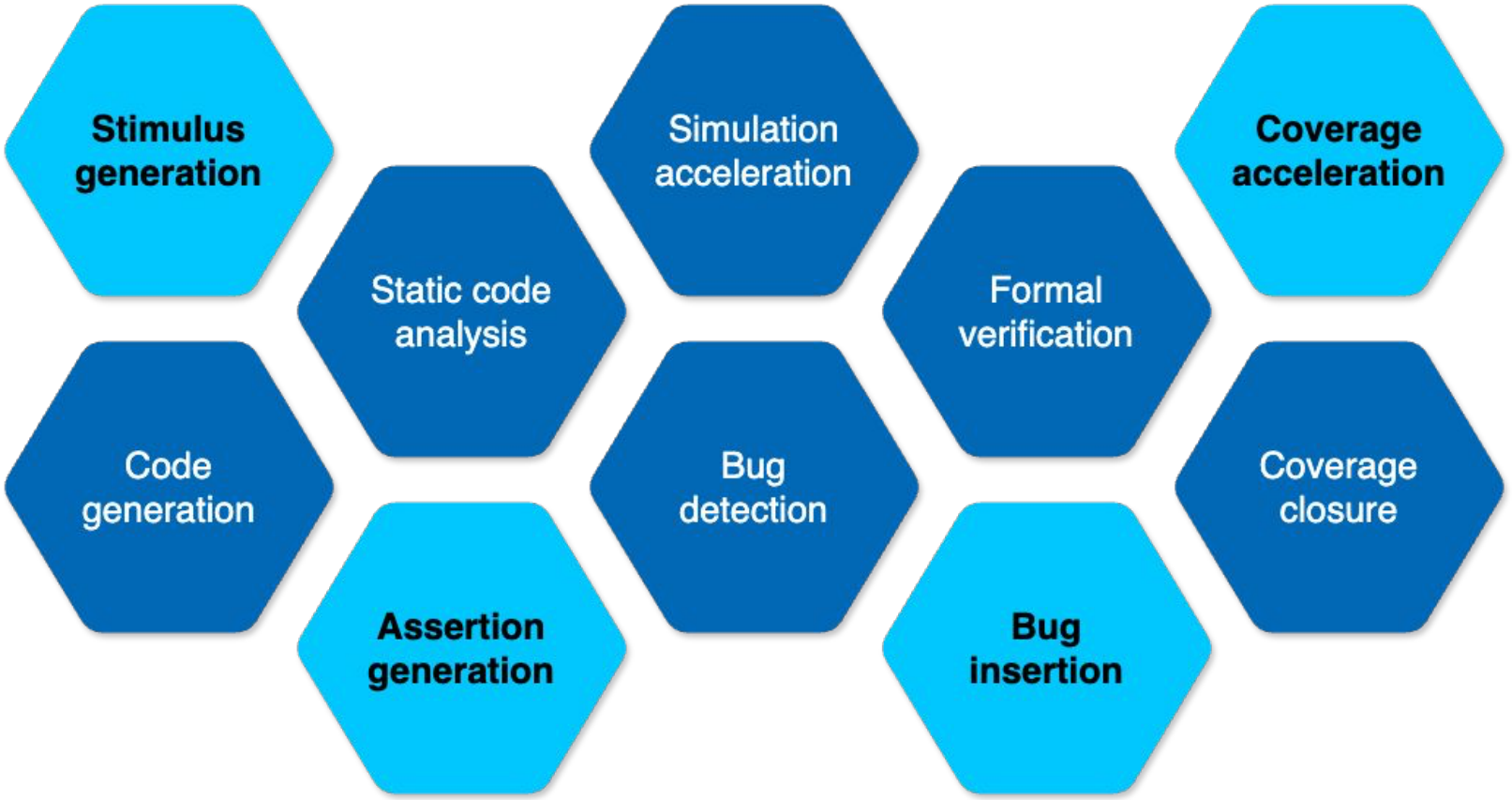
- State-of-the-art survey of ML applications in verification
- Stimulus Generation and Optimization using Machine Learning Predictors
- Automatic Bug Classification
- Automatic Bug Insertion using LLMs
- Coverage Acceleration
- Assertion Generation using LLMs
- Test Bench Generation using LLMs

TEXAS A&M UNIVERSITY

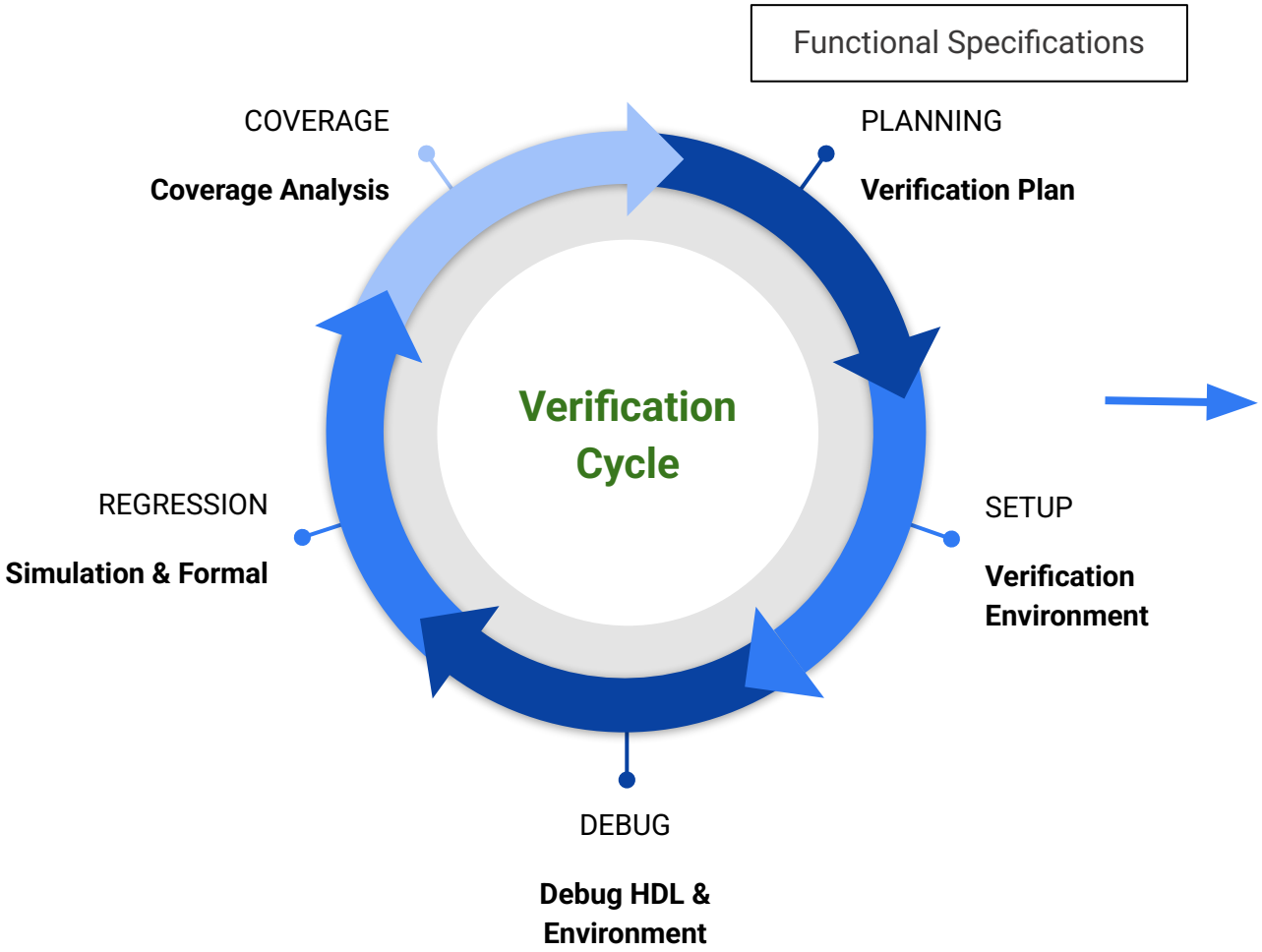
State-of-the-art Survey of ML applications in verification

Review

Applications of ML to Hardware Verification



Rich Data Set Available in Functional Verification



Step	Data	Format
Simulation	Waveforms	Table
Simulation	Test vectors	Text, Table
Simulation	Test results	Text, Table
Simulation	Logs	Text
Simulation	Coverage	Table
Formal	Design	Code
Formal	Proof results	Table
Formal	Traces	Text, Table
Formal	Constraints	Code
Formal	Assertions	Code
Debug	Trace logs	Text, Table
Debug	State dumps	Table
Debug	Breakpoint	Table
Debug	Testbench	Code

Functional verification generates rich datasets ideal for machine learning applications.

Motivation and Research Questions?

- Enhancing Functional Verification Efficiency
  - How to reduce simulation time using ML predictions?
  - Can ML optimize test generation resources?
- Leveraging Functional Verification Data and Managing Complexity
  - How to leverage ML to extract value out of functional verification data?
  - How to handle varying design sizes and complexities?
- Tailoring ML Approaches for Verification Tasks
  - Which ML architectures best suit different verification tasks?
  - Can ML assess verification completeness?
  - How to validate ML-based verification results?

State-of-the-art survey of ML applications in verification

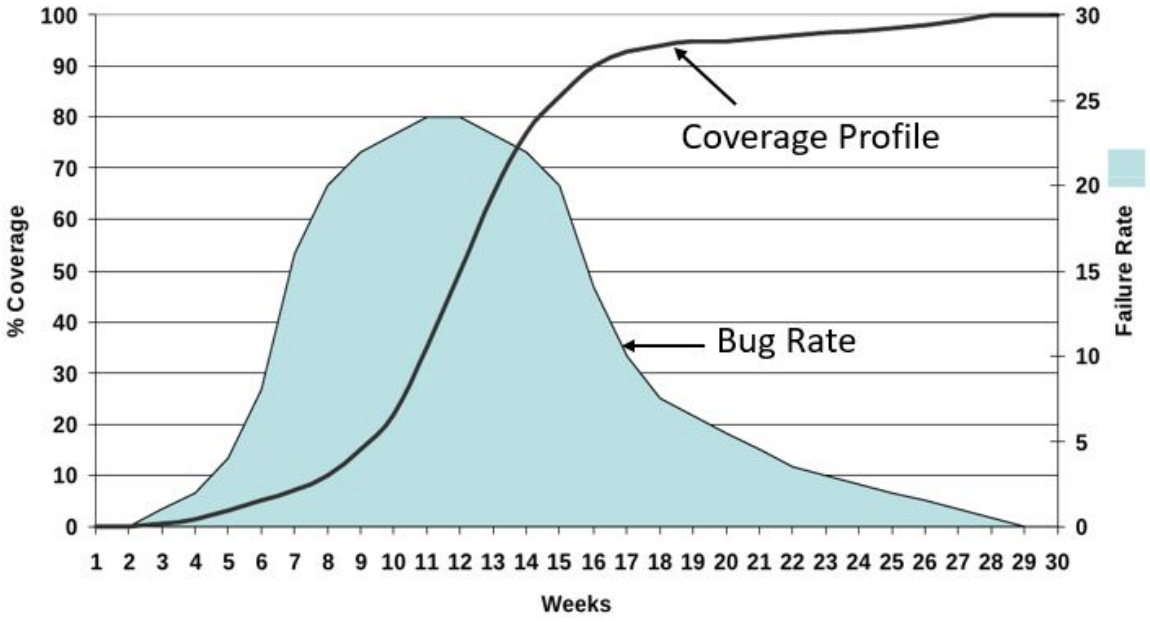
Categories	Applications
Requirement engineering	Code generation from requirement specs
	LLM-aided testbench generation
	Automatic assertion generation from specifications
Verification Acceleration	Stimulus generation and optimization
	Coverage-directed test generation (CDTG)
	ML-guided random test generation
Coverage Closure	Last-mile coverage optimization
Bug detection and localization	ML-guided bug triage
	Multimodal bug analysis (waveforms, design architecture, reports)
Formal verification	Assertion checking optimization
	Property verification assistance

Stimulus Generation and Optimization using Machine Learning Predictors

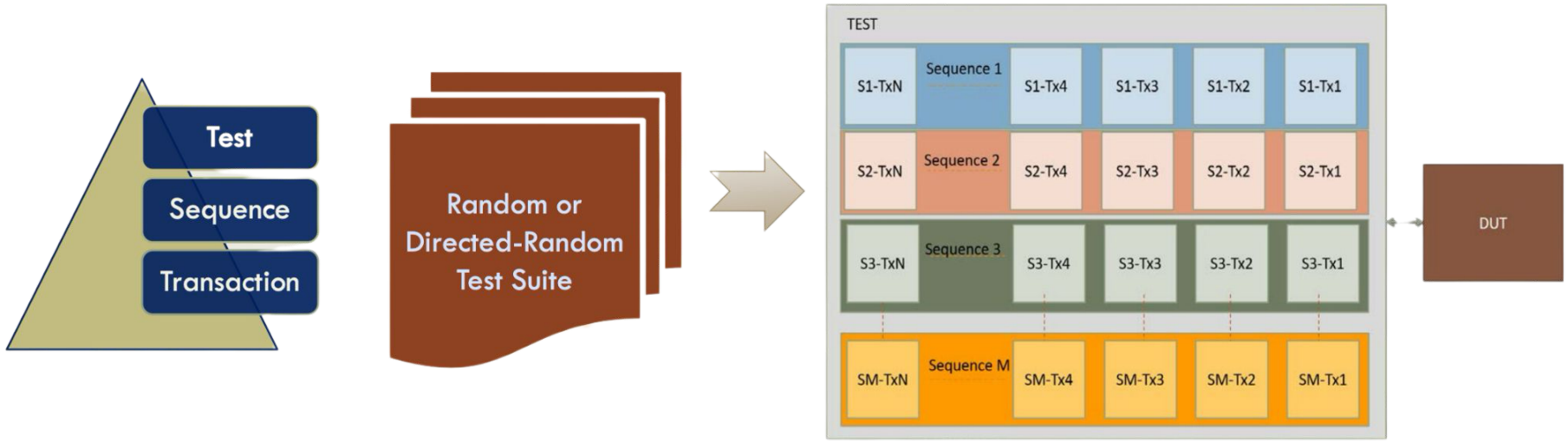
Methodology and experimental results

Motivation

- Functional verification consumes >55% of chip design resources
- Simulation-based verification faces exponentially growing input space
- Random tests often redundant, directed tests need extensive manual effort
- Coverage remains a major challenge
- **ML-guided stimulus generation promises faster coverage with fewer resources**



Methodology: Multi-Level Stimulus Optimization

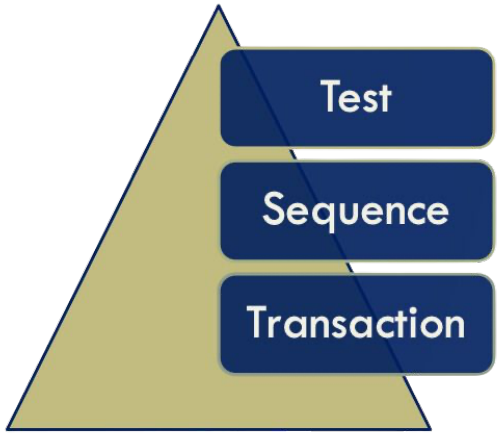


1. **Test Level Stimulus Optimization**
2. **Transaction Level Stimulus Optimization**

Methodology: Multi-Level Stimulus Optimization

Test Level Stimulus Optimization

Control knobs provide optimization that influence stimulus generation. The ML model predicts functional coverage based on **knob settings**. Redundant tests are pruned while continuously refining the predictions through retraining.



Transaction Level Stimulus Optimization

- **Online Transaction Pruning** evaluates transactions during simulation. When the ML model identifies a transaction as unlikely to improve coverage, it is pruned.
- **Offline Directed Sequence Generation** uses ML to construct optimal transaction sequences **before simulation**. It predicts combinations of transactions that will improve coverage.

Methodology: Multi-Level Stimulus Optimization

Test Level Stimulus Optimization

- Control through test knobs
- ML prediction of functional coverage
- Pruning of redundant tests
- Continuous model retraining

Transaction Level Stimulus Optimization

Online Transaction Pruning

- Real-time processing during simulation
- Immediate pruning of non-beneficial transactions

Offline Directed Sequence Generation

- ML-guided sequence creation before simulation
- Optimized for coverage improvement

Experimental Setup

DUT: Quad-core MESI-based Cache design

- L1 cache (private to each core)
- Shared L2 cache
- MESI cache coherency protocols
- 4-way cache associativity

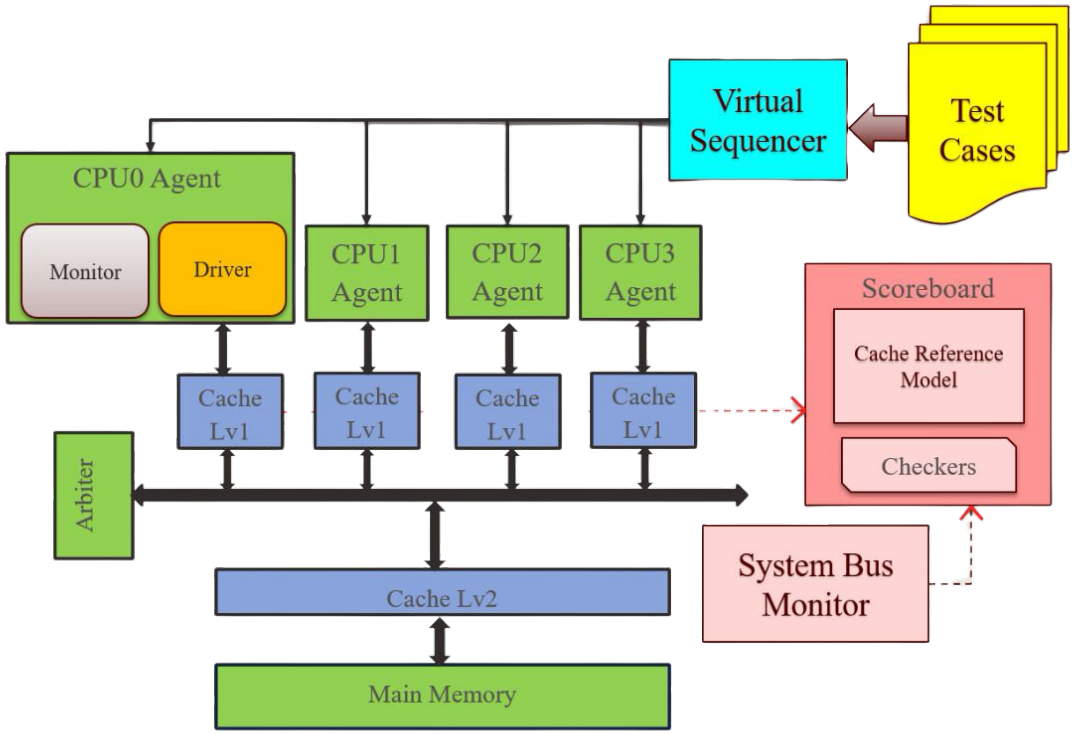
Testbench: UVM compliant testbench

Libraries

- Keras
- Scikit-Learn

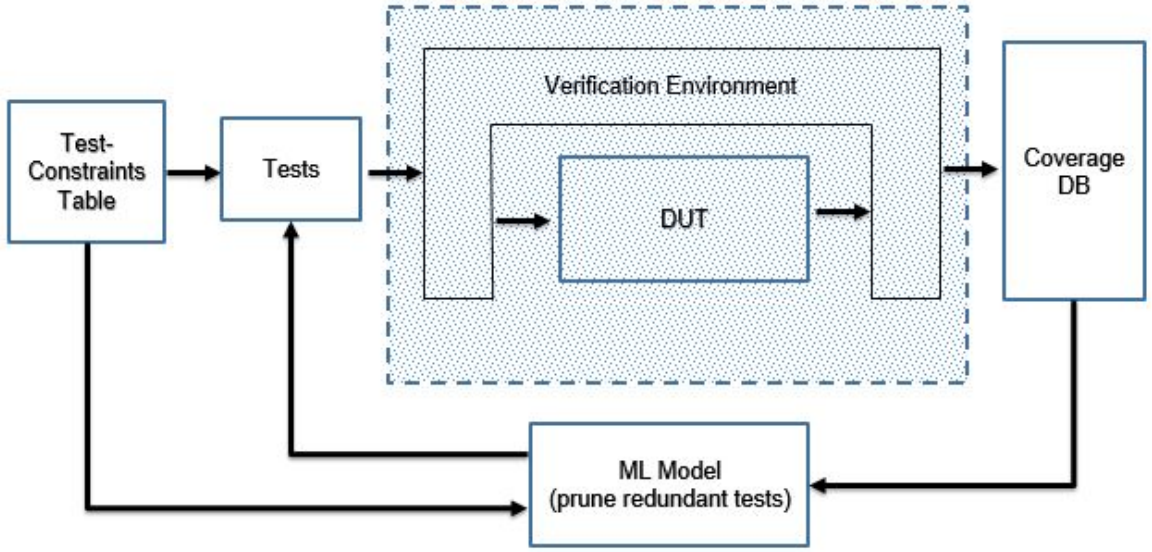
ML Models

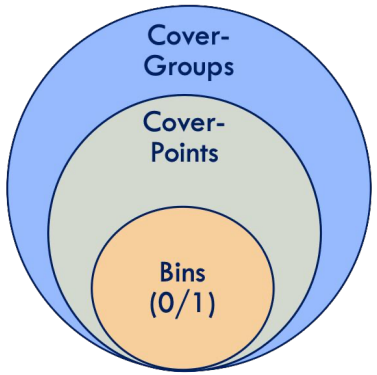
- Deep Neural Network (DNN)
- Random Forest (RF)
- Support Vector Machine (SVM)
- Long Short Term Memory (LSTM)



Test Level Stimulus Optimization – Flow

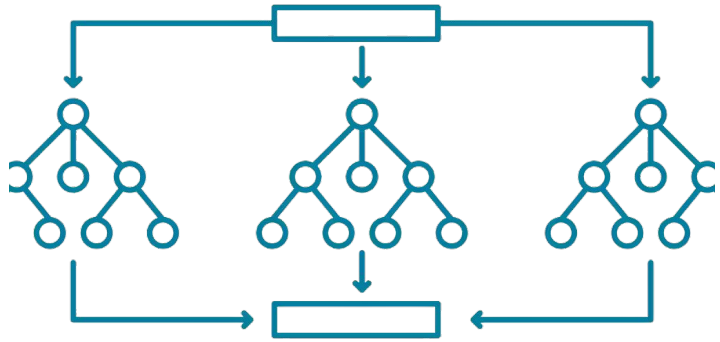
1. Run random tests with randomized test knobs
2. Collect coverage data and logs as training data
3. Extract features from test knobs for ML model
4. Train model to predict coverage from knob settings
5. Use model to predict coverage for new test knobs
6. **Prune tests predicted to hit only covered bins**
7. Retrain model with new simulation data
8. Repeat until full coverage achieved





Test Level Stimulus Optimization – Test Coverage Prediction

Will a test improve verification coverage?

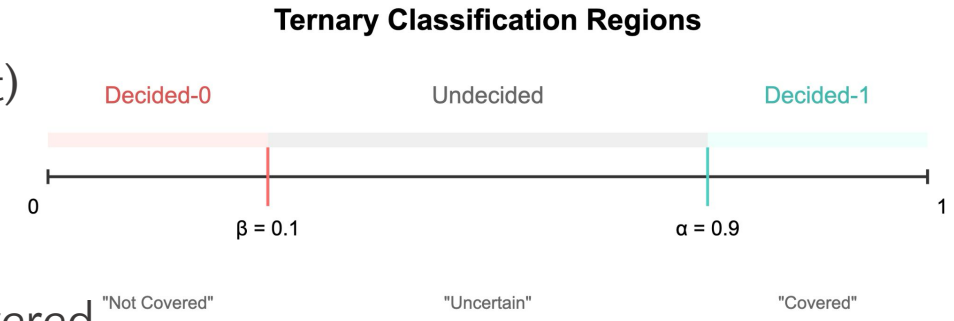


We use a **probability-based classification** with two thresholds **(α, β)**.

1. The ML model (Random Forest) outputs a probability **p** of each **coverage bin** hit by a test.

2. Use 02 threshold values **0 < β < α < 1**, classify each bin into **03 categories**:

- **Decided-1 (Covered)**: if p ≥ α (high confidence bin will be hit)
- **Decided-0 (Not Covered)**: if p ≤ β (high confidence bin won't be hit)
- **Undecided**: if β < p < α (model uncertain about coverage)



3. Make pruning decisions based on classifications:

- **Prune test** if high % of decided bins and decided-1 bins already covered
- **Run test** if it has decided-1 predictions for uncovered bins
- **Run test** if high % of undecided bins (suggests novel coverage potential)

Test Level Stimulus Optimization – Results

Group A (827 bins, 6 metrics)

Strong correlation with test knobs

Initial training achieved <10% error with just 50 random tests

Baseline: Required 587 tests for full coverage

With ML optimization:

- **Random Forest: 129 tests (78% reduction)**
- **DNN: 137 tests (77% reduction)**
- **SVM: 185 tests (68.5% reduction)**

Group B (911 bins, 2 metrics)

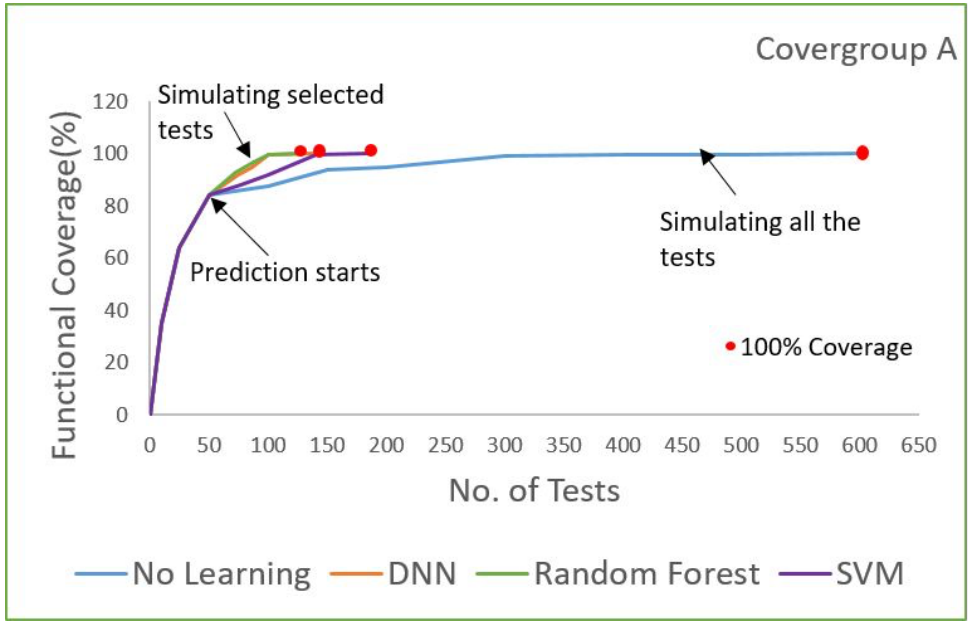
Poor correlation with test knobs

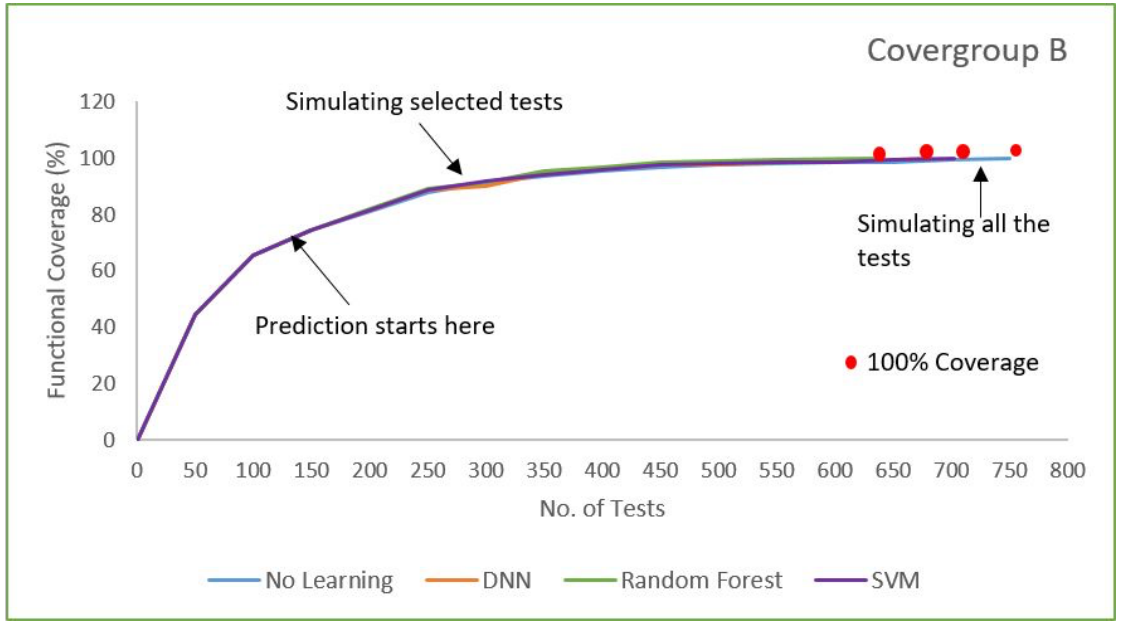
High classification error even after intensive training

ML optimization showed minimal improvement

Led to development of Transaction Level approach

Test Level Stimulus Optimization – Group A vs. Group B





Functional coverage closure for 02 groups with test level optimization.

1. Plot on the left shows the coverage closure compressed towards left for covergroup A
2. Plot on the right shows no significant left compression for covergroup B.

Observation:

The results reveal limitation of test level optimization

It confirms the need for transaction level optimization for coverage metrics in covergroup B.

Transaction Level Stimulus Optimization – Flow

Phase 1 – Initial Training

1. Run random simulations and collect transaction data
2. Extract features (transaction attributes + context/history)
3. Train ML model to predict coverage impact

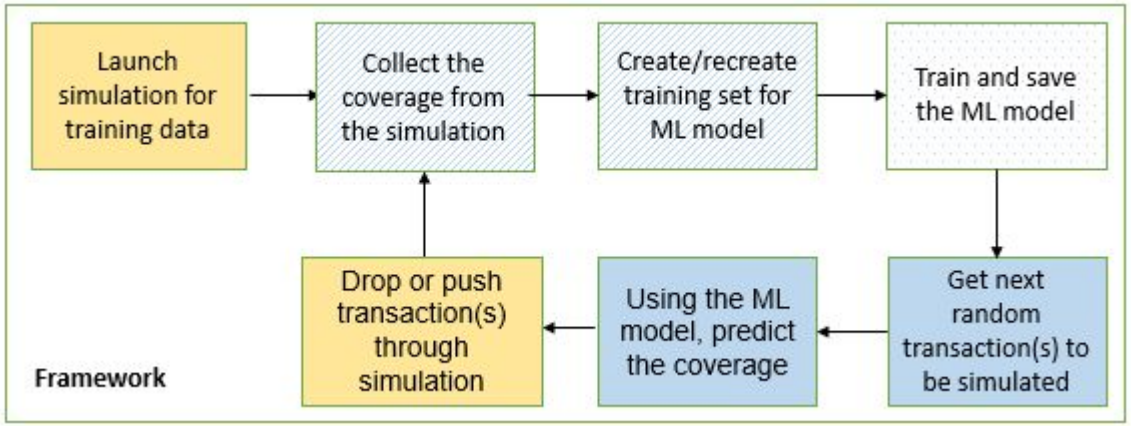
Phase 2 – Coverage Optimization

Online Transaction Pruning

1. Generate random transactions
2. Predict coverage impact before simulation
3. Prune transactions unlikely to improve coverage
4. Simulate remaining transactions

Offline Directed Generation

1. Use ML to predict high-impact transaction sequences
2. Generate complete transaction sequence
3. Simulate entire sequence at once



Transaction Level Stimulus Optimization – Flow

FSM Transition Coverage

Input Features

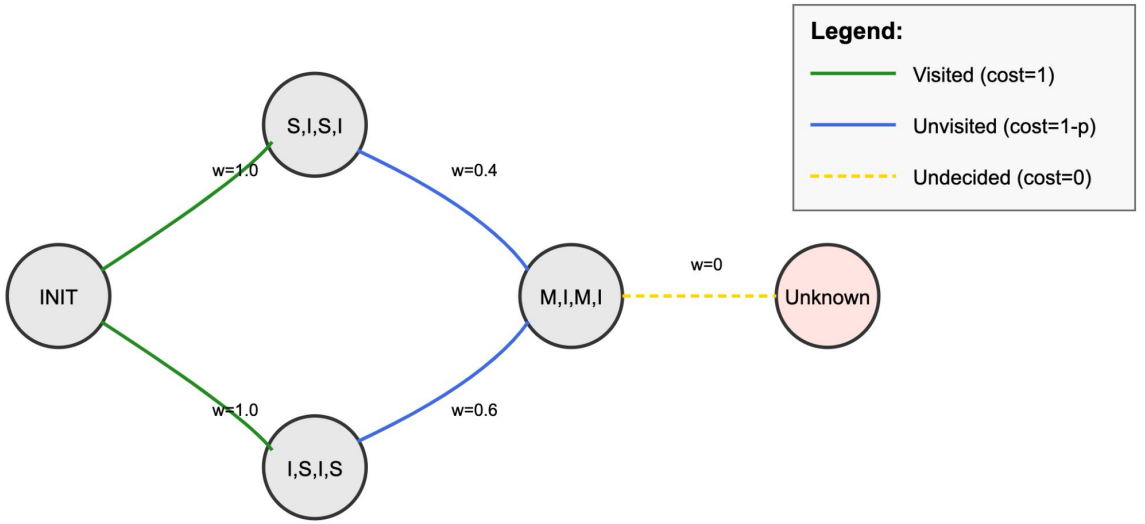
- Transaction attributes (core, request type, address)
- Current state information

ML Model

- Predicts next state based on **current state + transaction**
- Used to forecast state transitions

Coverage Strategy

- Online Pruning: Filter transactions unlikely to create new transitions
- Offline Generation: Use graph-based approach with Transaction Attribute (TA) transition graph.



Transaction Attribute Transition Graph

Path Analysis

- Path 1 Total Cost: 2.4 (Recommended)
- Path 2 Total Cost: 2.6

Transaction Level Stimulus Optimization – Flow

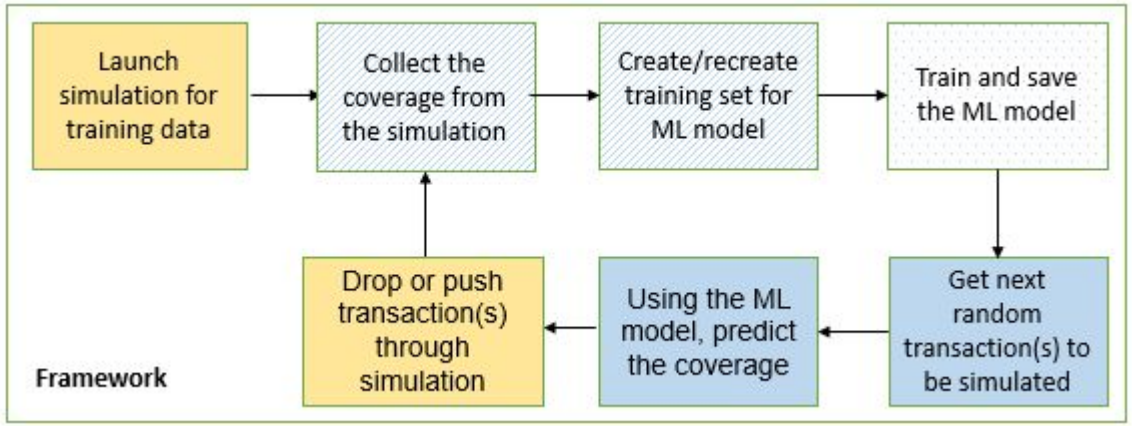
Non-FSM Event Coverage (e.g., Cache Hits)

Input Features

- Sequence of w consecutive transactions (window size)
- Each transaction's attributes

ML Model

- LSTM (Long Short-Term Memory) handles sequence dependencies
- Predicts if events of interest will occur



Coverage Strategy

- Online Pruning: Examine w-transaction windows for event prediction
- Offline Generation: Generate complete sequences targeting uncovered events

Transaction Level Stimulus Optimization – Results

FSM Transition Coverage Results

Baseline Random Testing:

- 50K cycles to reach 80% coverage
- Additional 65K cycles needed for remaining 20%
- Shows high redundancy in conventional approach

Optimization Results

FSM (MESI State Transitions)

Online Transaction Pruning: • 48% reduction in simulation cycles • 57% reduction in total CPU time

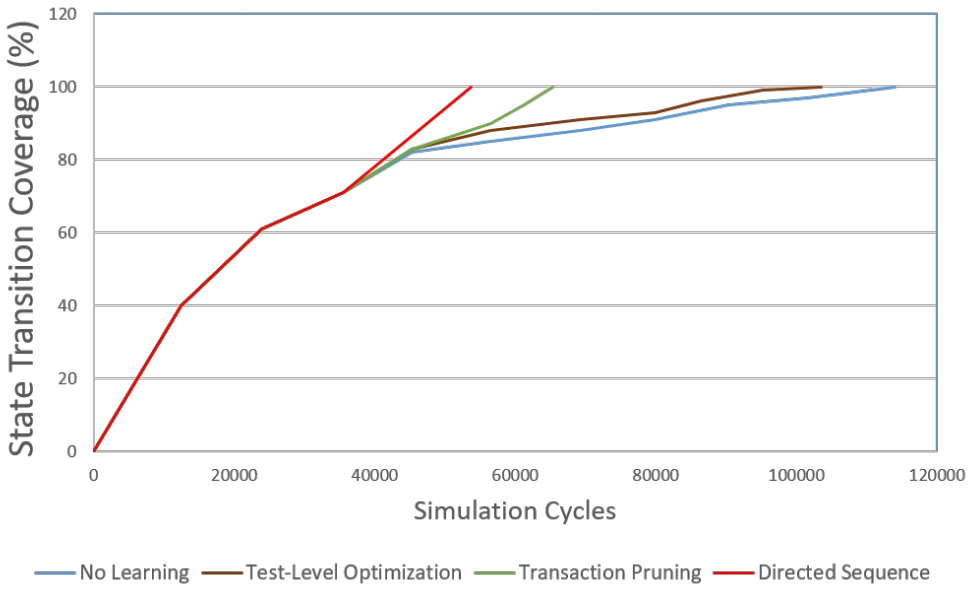
Offline Sequence Generation: • 55% reduction in simulation cycles • 69% reduction in total CPU time

Non-FSM (Cache Hit Events)

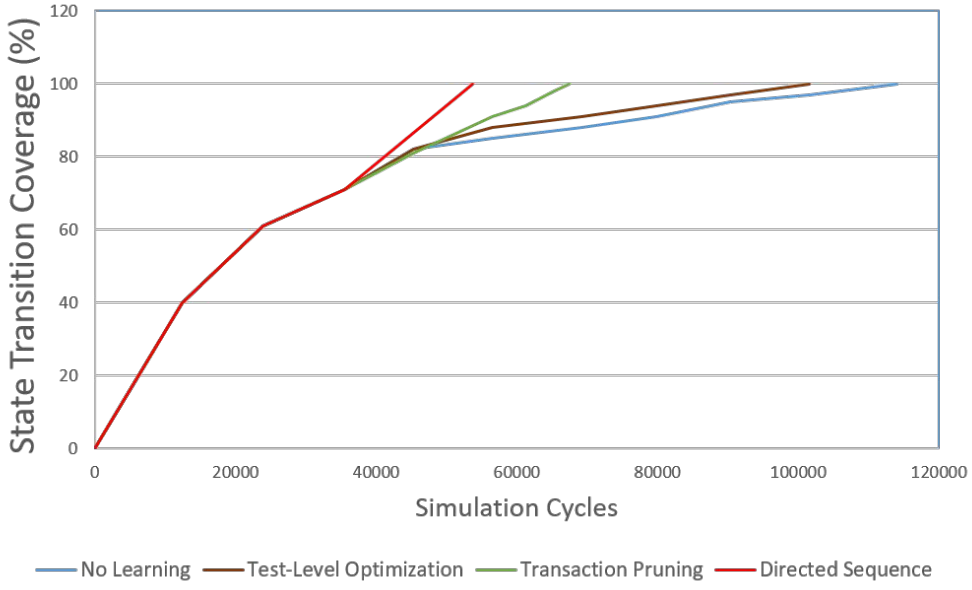
Online Transaction Pruning: • 55% reduction in simulation cycles • 65% reduction in total CPU time

Offline Sequence Generation: • 61% reduction in simulation cycles • 72% reduction in total CPU time

Transaction Level Stimulus Optimization – Results



FSM transition coverage through DNN classifier

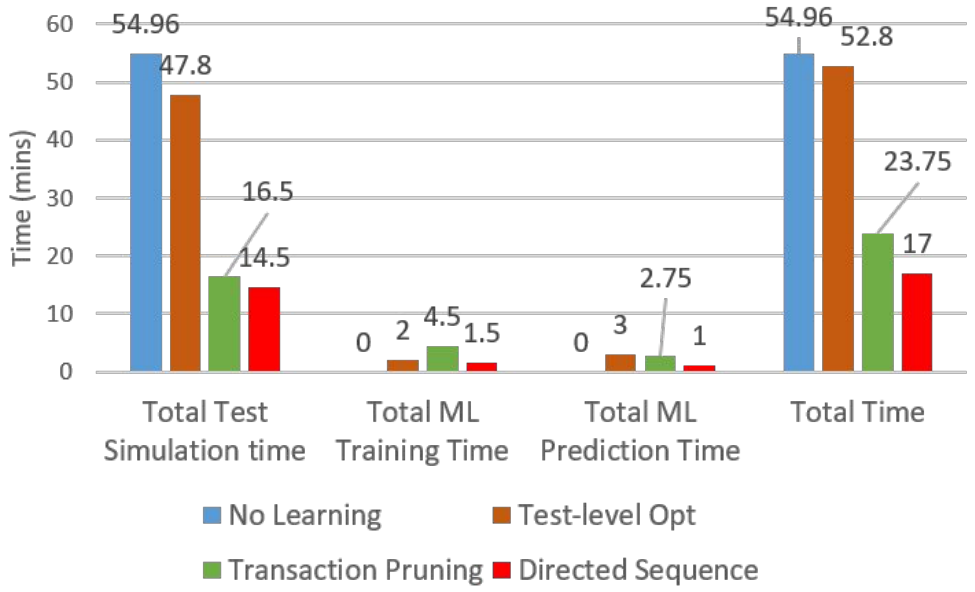


FSM transition coverage through random forest classifier

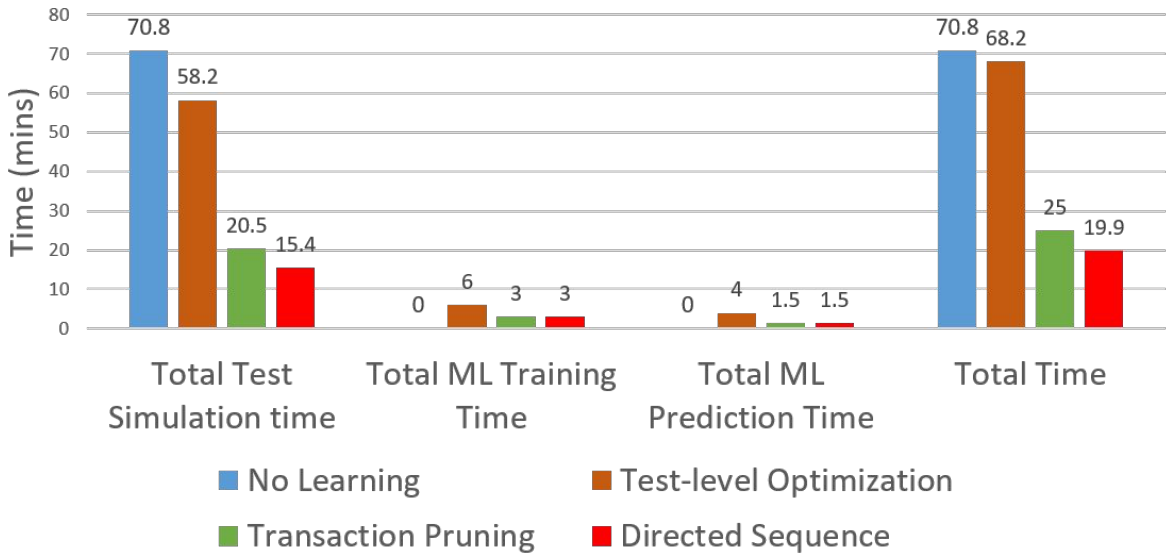
Observations:

- DNN and Random Forest achieve nearly identical coverage acceleration, reducing simulation cycles by approximately 48–55%.
- Random Forest showed better training efficiency vs DNN.

Transaction Level Stimulus Optimization – Results



CPU runtime for random forest-based technique (FSM transition overage)



CPU runtime for LSTM-based technique (non-FSM event coverage)

Observations:

Coverage closure with **transaction pruning** and **directed sequence generation** noticeably reduce CPU runtime.

Conclusion

- Transaction-level ML optimization achieves 70% reduction in verification time
- Both online pruning and offline generation prove effective for coverage improvement
- Random Forest and LSTM models outperform traditional approaches
- Method integrates easily with existing verification environments
- Results demonstrate clear path forward for ML in verification

Automatic Bug Classification

Overview

Motivation

When failures encountered, DV engineers must figure out the reason(s)

- Multi-step process
- Bug triage: Evaluating, bucketing, and prioritizing bugs
- Root-Cause Analysis: Find the root-cause of a bug

Artificial Intelligence

- In verification, AI has enabled: Faster coverage closure, More efficient stimulus generation
- AI for bug triage and root-cause analysis is still in its infancy
- Analyzes data from current, concurrent, and previous projects

Reusability and Extensibility

- Reusability: Allowing solutions to be replicated across various scenarios, saving time and resources
- Extensibility: Enabling easy modifications and improvements, helping solutions evolve to meet future requirements
- Prioritizing reusability and extensibility can significantly enhance the applicability of verification research to real-world projects

Previous Works

A failure triage engine based on error trace signature extraction – Z. Poulos et al.

- Bug triage
- FAE: Autoencoder-based engine for failure binning – C.-H. Shen et al.

Clustering and classification of UVM test failures using machine learning techniques – A. Truong et al.

- Bug triage

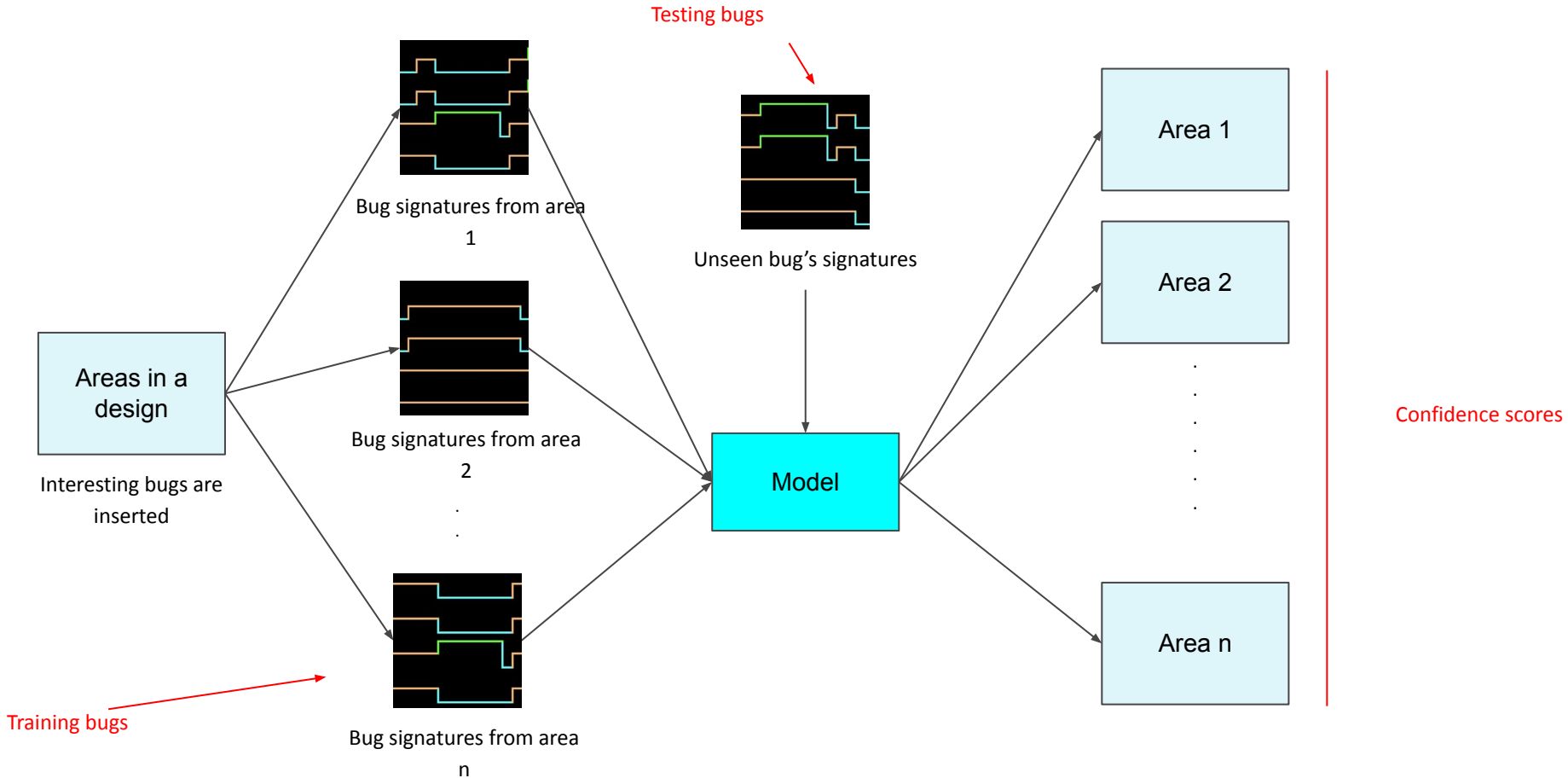
BugMD: ML-powered mismatch detection via RTL and ISS – B. Mammo et al.

- Bug triage/RCA

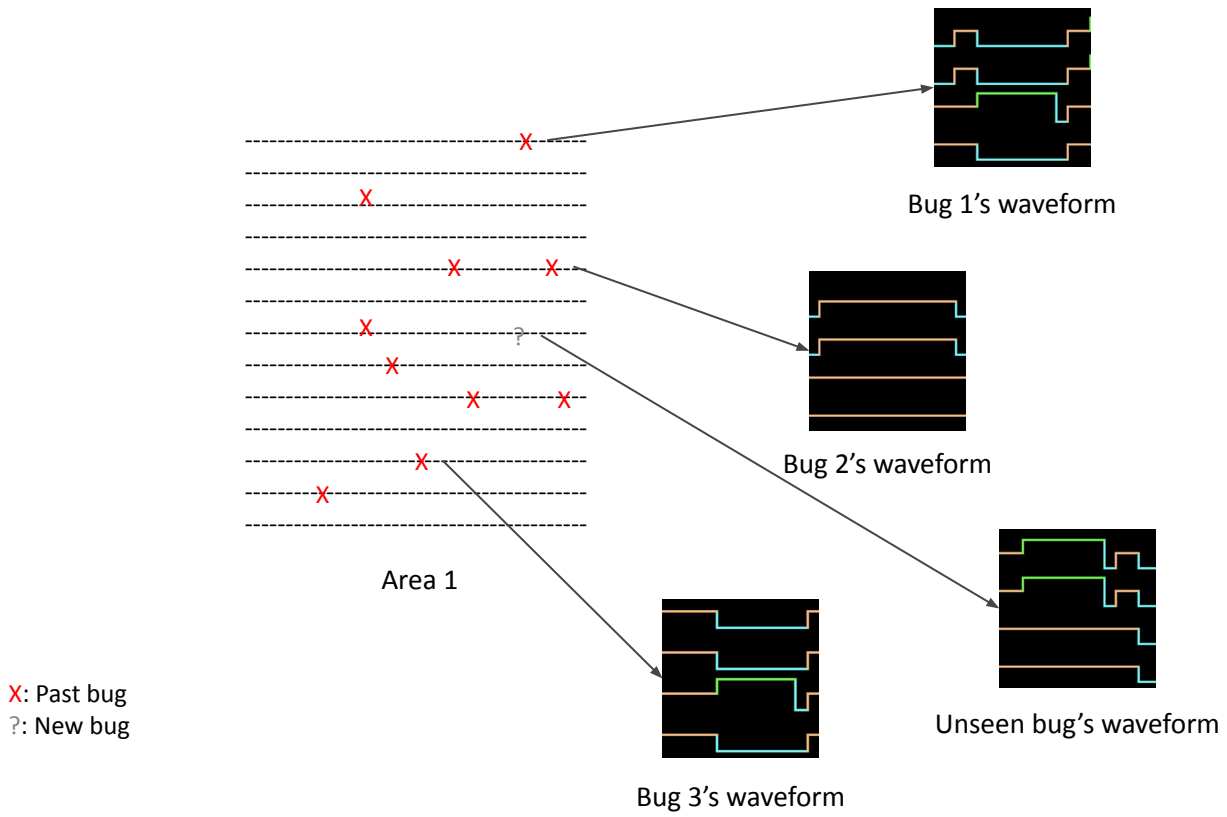
BugMD: ML-powered mismatch detection via RTL and ISS – B. Mammo et al.

- RCA

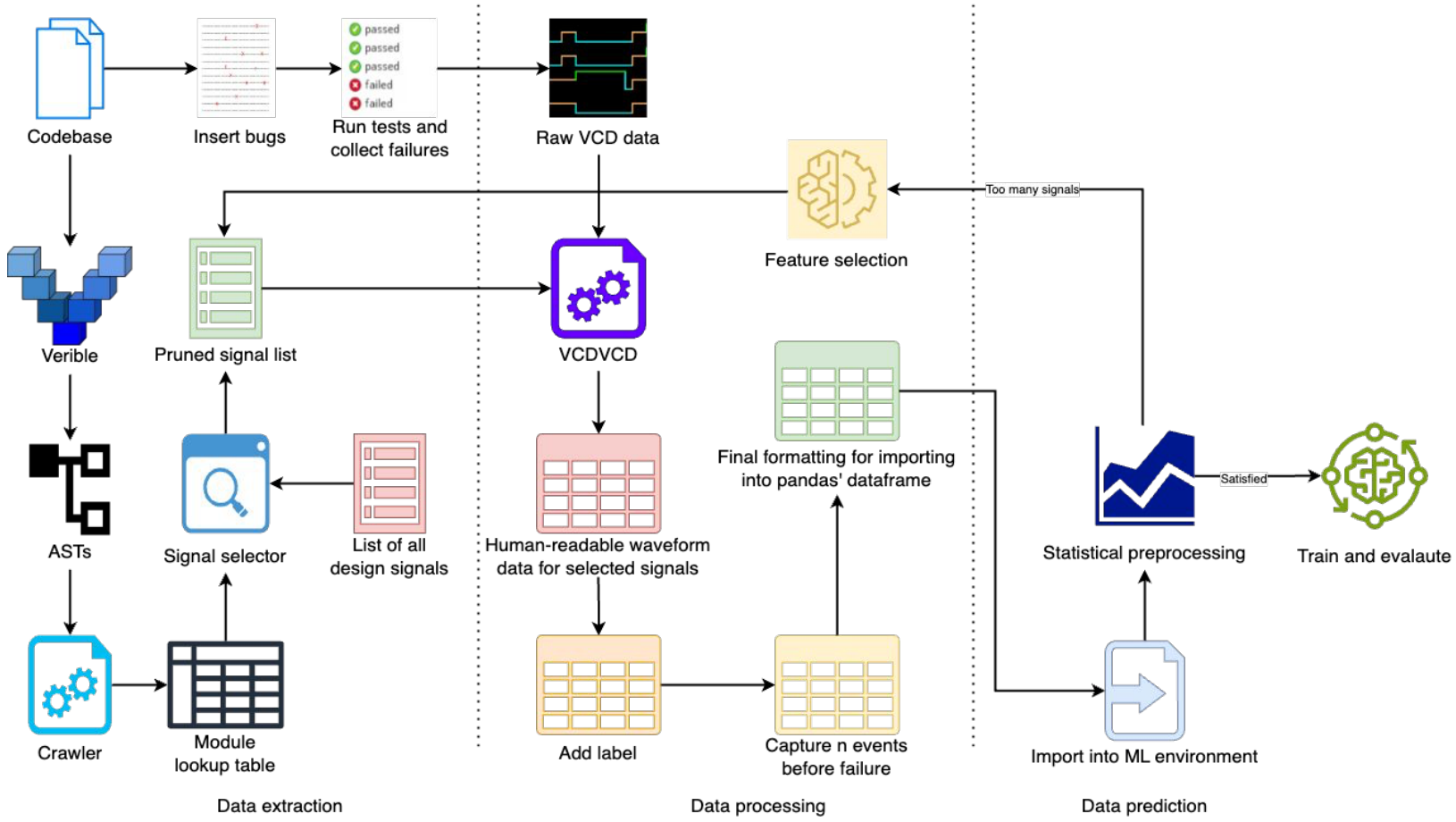
Overall Concept



Overall Concept



Overall Concept

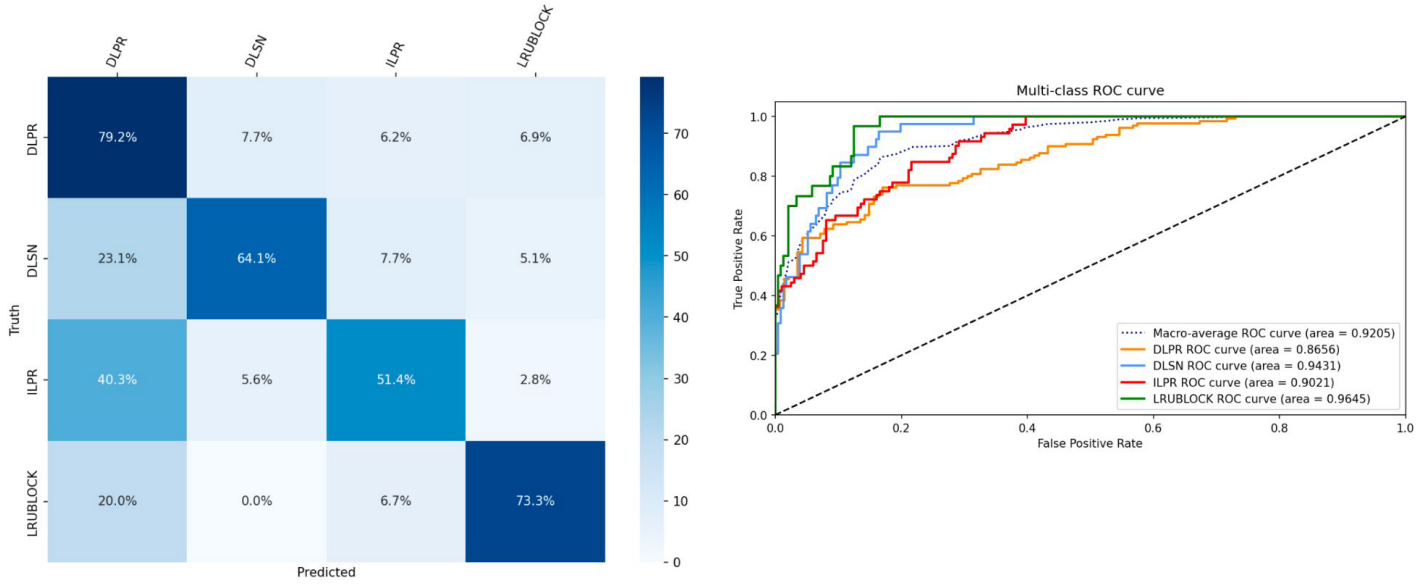


Preliminary Results

Metrics	KNN	SVM	RF	XGB	GBM
Multicore MESI Based Cache - 4 areas, 340 signals, 1GB					
Avg. accuracy	0.4834	0.5351	0.6531	0.7011	0.7232
Top 3 avg. accuracy	0.8413	0.9225	0.9779	0.9741	0.9742
Macro avg. precision	0.4463	0.5128	0.6230	0.6782	0.7063
Macro avg. recall	0.4741	0.5258	0.6650	0.6646	0.7032
Macro avg. F1-score	0.4527	0.4943	0.6328	0.6689	0.6955
Macro avg. AUC ROC	0.7112	0.7976	0.8966	0.9126	0.9199
Total training size	886	886	886	886	886
Total testing size	271	271	271	271	271
OpenTitan AES - 7 areas, 1031 signals, 23GB					
Avg. accuracy	0.3977	0.4291	0.6306	0.6589	0.6500
Top 3 avg. accuracy	0.7915	0.8593	0.9423	0.9383	0.9514
Macro avg. precision	0.4155	0.3153	0.7068	0.7344	0.7273
Macro avg. recall	0.3405	0.3088	0.6411	0.7010	0.6976
Macro avg. F1-score	0.3570	0.2400	0.6658	0.7078	0.7031
Macro avg. AUC ROC	0.7348	0.7976	0.9032	0.8959	0.9017
Total training size	3372	3372	3372	3372	3372
Total testing size	988	988	988	988	988
FabScalar - 12 areas, 4860 signals, 43GB					
Avg. accuracy	0.4621	0.2096	0.5176	0.5303	0.5379
Top 3 avg. accuracy	0.6061	0.3257	0.7247	0.7576	0.7601
Macro avg. precision	0.4583	0.4897	0.5148	0.5098	0.5261
Macro avg. recall	0.4497	0.1842	0.5149	0.5204	0.5148
Macro avg. F1-score	0.4436	0.1807	0.4970	0.5019	0.5073
Macro avg. AUC ROC	0.7003	0.6033	0.8609	0.8742	0.8732
Total training size	2670	2670	2670	2670	2670
Total testing size	396	396	396	396	396

Baseline Classification Performance

Preliminary Results

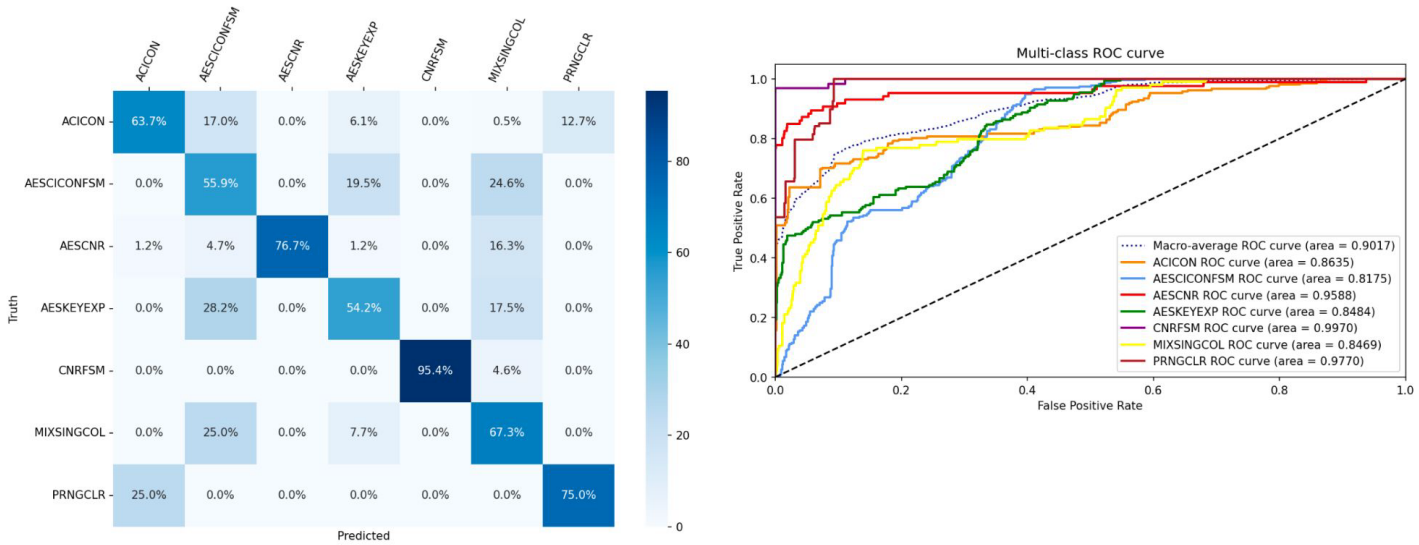


(a) Confusion Matrix

(b) ROC Curve

Baseline Graphs for MESI with LightGBM model

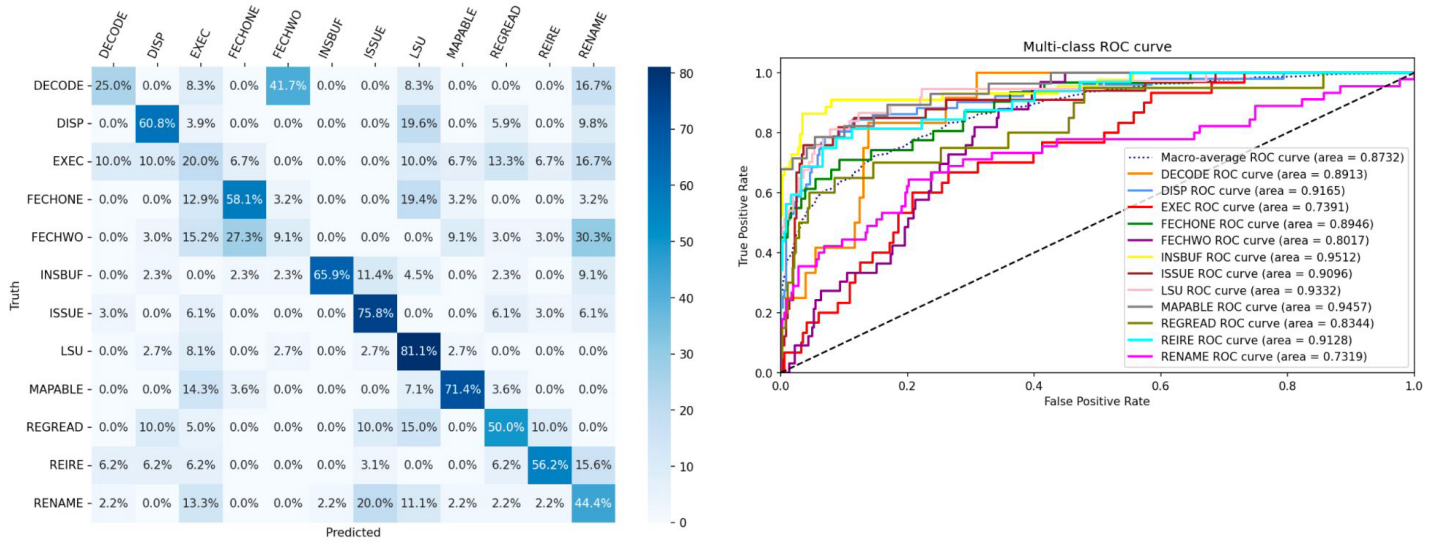
Preliminary Results



(a) Confusion Matrix (b) ROC Curve

Baseline Graphs for AES with LightGBM model

Preliminary Results



(a) Confusion Matrix

(b) ROC Curve

Baseline Graphs for FabScalar with LightGBM model

Evaluation: Performance – Compare to BugMD

Metrics	BugMD	VCDiag
Avg. accuracy	0.7319	0.5597
Top 3 avg. accuracy	0.9000	0.7765
Total number of bugs	7080	402
Training data size	35352	3048
Testing data size	3756	461

Numbers from BugMD paper vs. actual VCDiag performance both using Random Forest

Automatic Bug Insertion using LLMs

Overview

Motivation

Verification teams face significant overhead in manual bug insertion and testing

- Engineers manually analyze designs, identify potential vulnerabilities, and insert artificial bugs
- Requires deep expertise in design and verification
- Manual efforts scale very poorly across large designs or multiple testbenches

Manual efforts are time-intensive and error-prone

- Each bug is carefully crafted to test specific mechanisms, creating long development cycles
- Human bias leads to non-representative bug distributions: over-tested and under-tested classes
- Significant effort put into debugging and refining inserted bugs

How LLMs can help significantly

- Can generate **meaningful** and **varied** bugs automatically
- More bug scenarios lead to greater coverage across different failure modes
- Greater consistency, easier error recovery, & limited bias

Key challenges & Solutions with LLM Processing

HDL and SystemVerilog files frequently exceed LLM context windows

- We propose a modular LLM verilog splitter that intelligently chunks large verilog files into a complete partition of workable regions

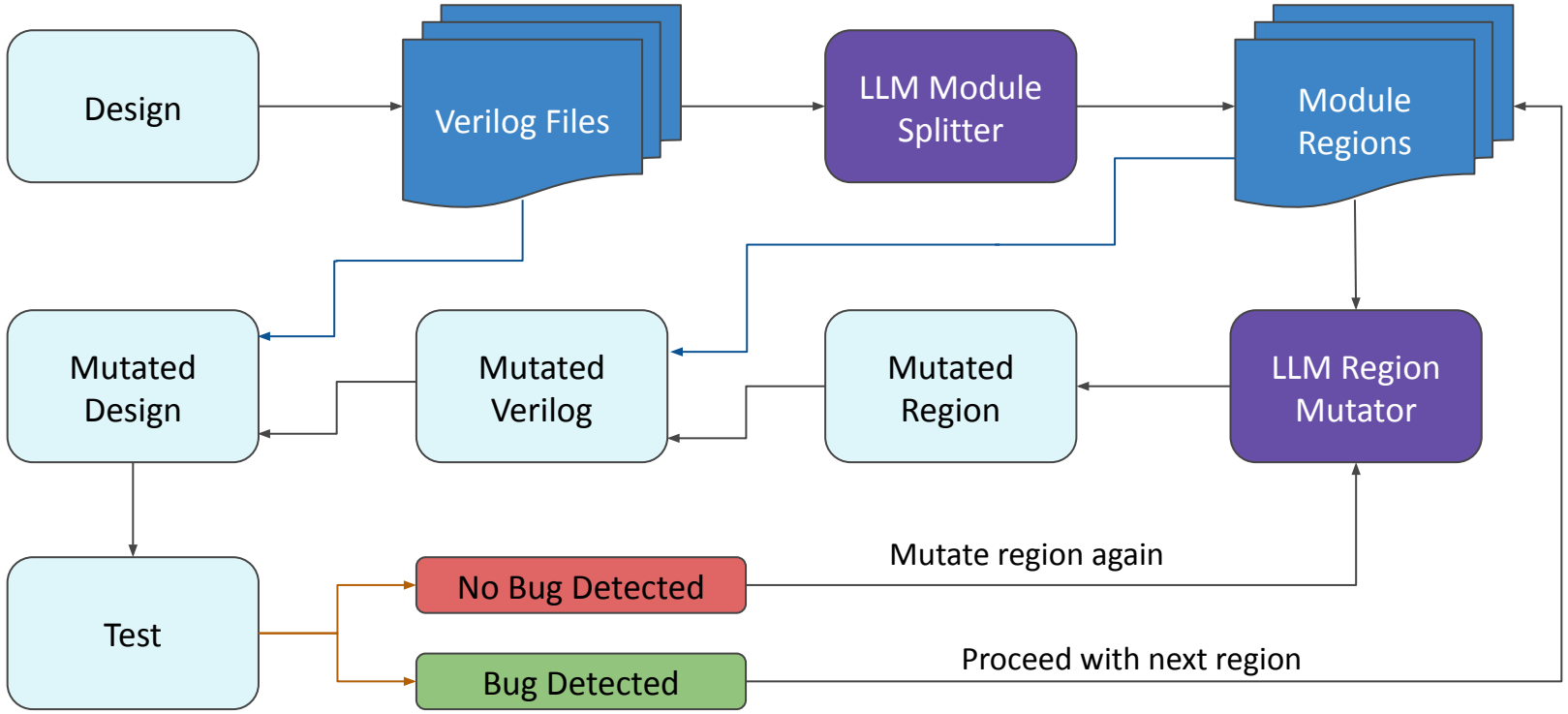
Variability in bug generation and overall stability

- Follow a class-based mutation paradigm with detailed instructions for each mutation class
- Verify generated bugs for syntactical & functional correctness & redo if necessary

Model comprehension degrades with large code sections

- Divide the task into a series of 3 sequential steps
  - Select the region based on a list of LLM-generated region descriptions
  - Select what bug to inject into the region and in which line(s) to insert it
  - Inject the bug given the identified line(s) and detailed instructions

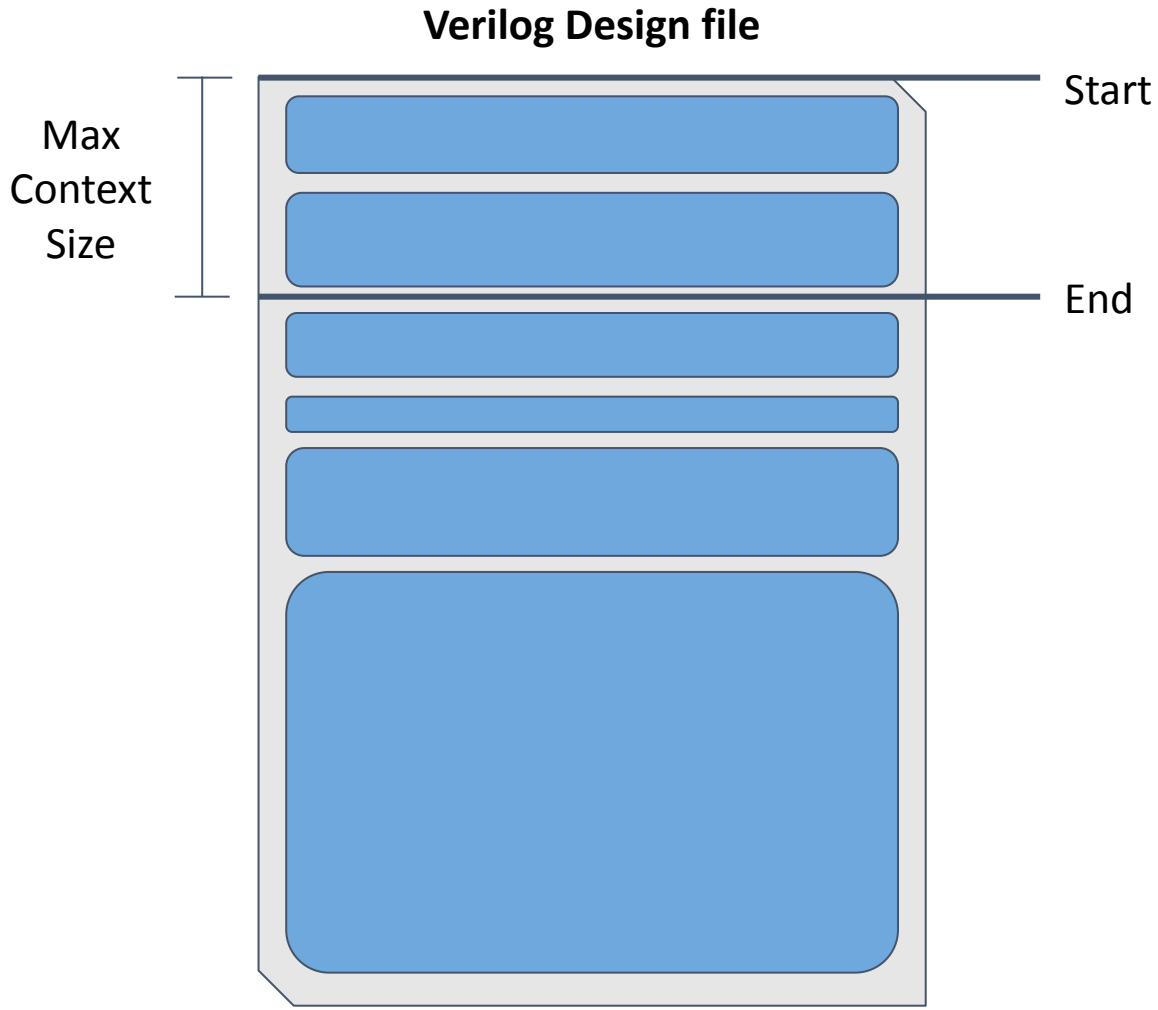
Overall LLM Workflow



LLM Module Splitter

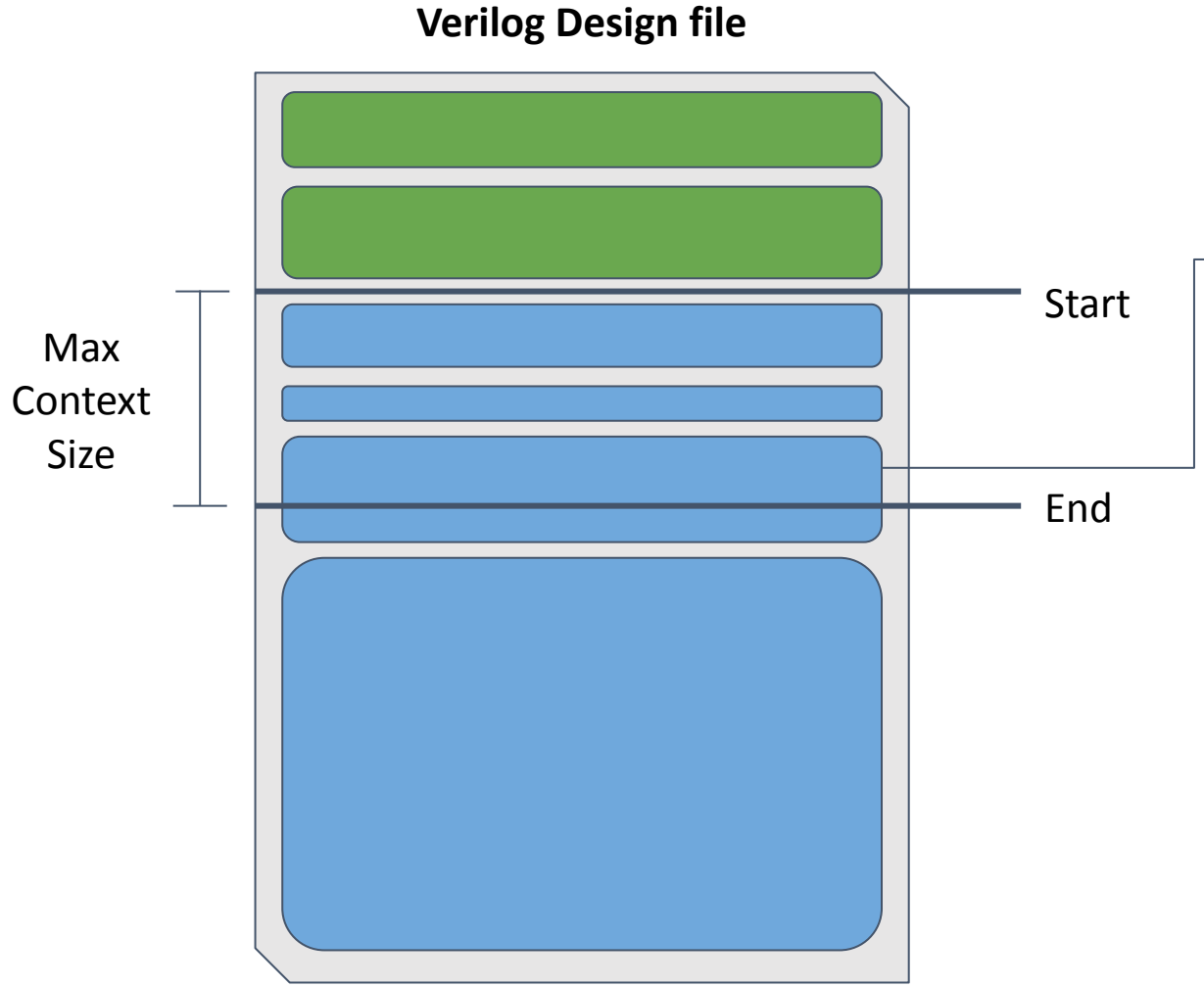
- Allow the LLM to define regions however it sees fit
  - Given a brief high-level hint as to what a region intuitively means
- Add line numbers preceding each line in the verilog content
- Ask LLM to produce a list of regions where each region is defined by:
  - Starting line number
  - Ending line number
  - Brief description of region

LLM Module Splitter – Chunking



If intuitively defined regions fit within context, there is no issue.

LLM Module Splitter – Chunking



How can we recognize the region is incomplete?

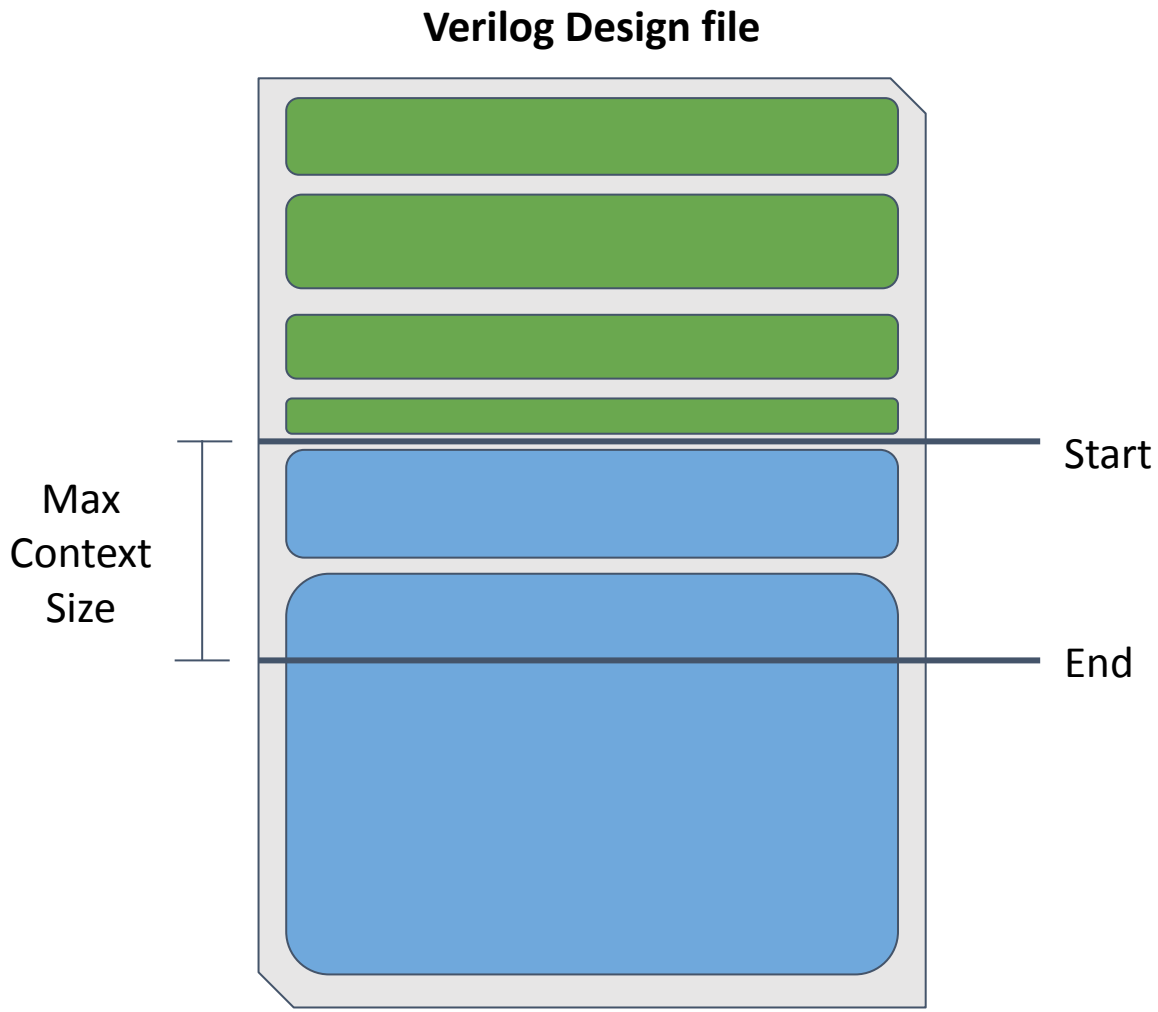
```
/* Following assigns decoded packets to the output ports.
*/
assign decodedPacket0_o    = decodedPacket_f[0];
assign decodedPacket1_o    = decodedPacket_f[1];
assign decodedPacket2_o    = decodedPacket_f[2];
assign decodedPacket3_o    = decodedPacket_f[3];
assign decodedPacket4_o    = decodedPacket_f[4];
assign decodedPacket5_o    = decodedPacket_f[5];
assign decodedPacket6_o    = decodedPacket_f[6];
assign decodedPacket7_o    = decodedPacket_f[7];
assign decodedVector_o    = decodedVector;

assign decodeReady_o    = fs2Ready_i;
```

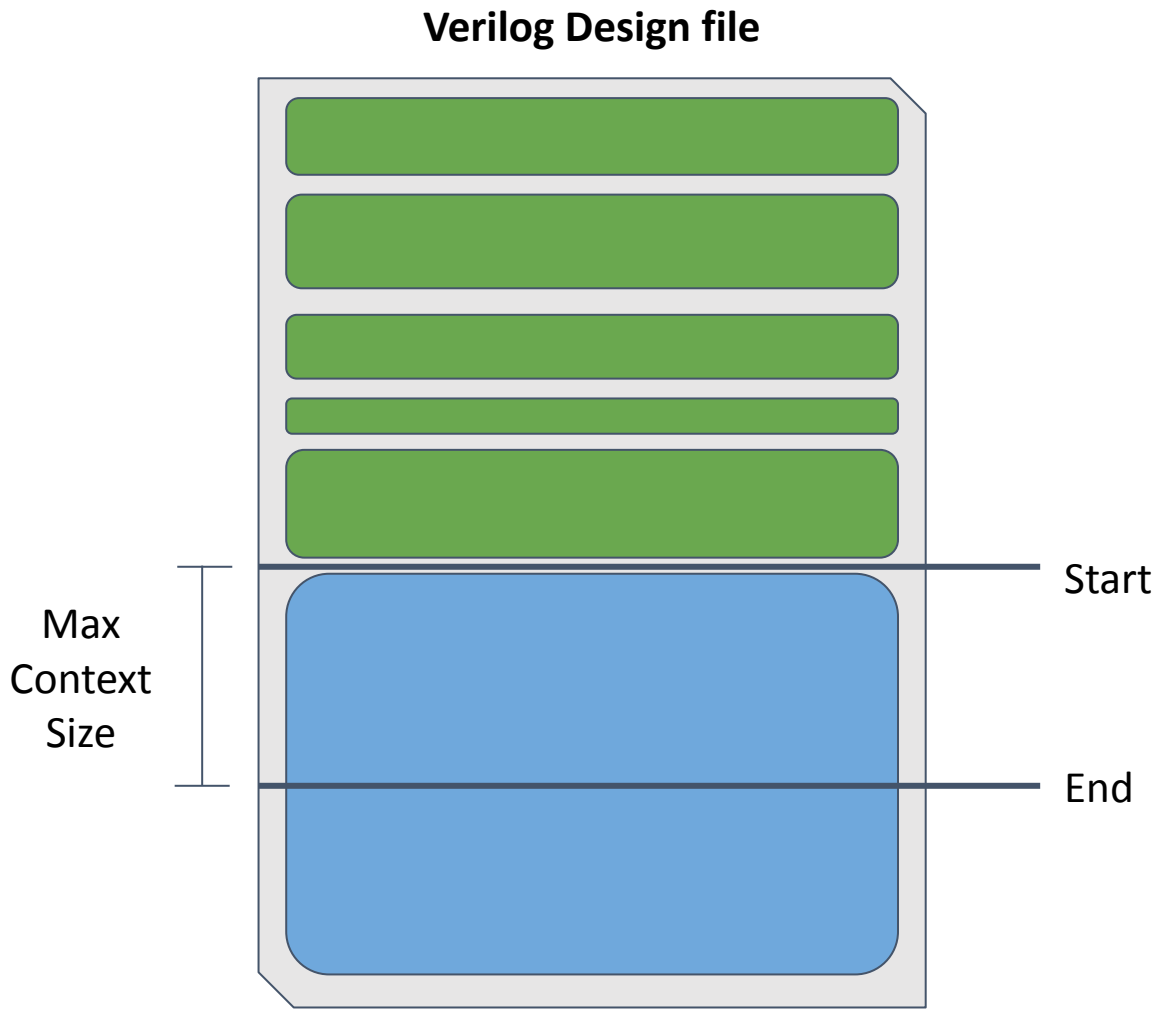
Solution: Provide auxiliary lines from next chunk

Instruct LLM to skip region if it spills over into the next chunk

LLM Module Splitter – Chunking

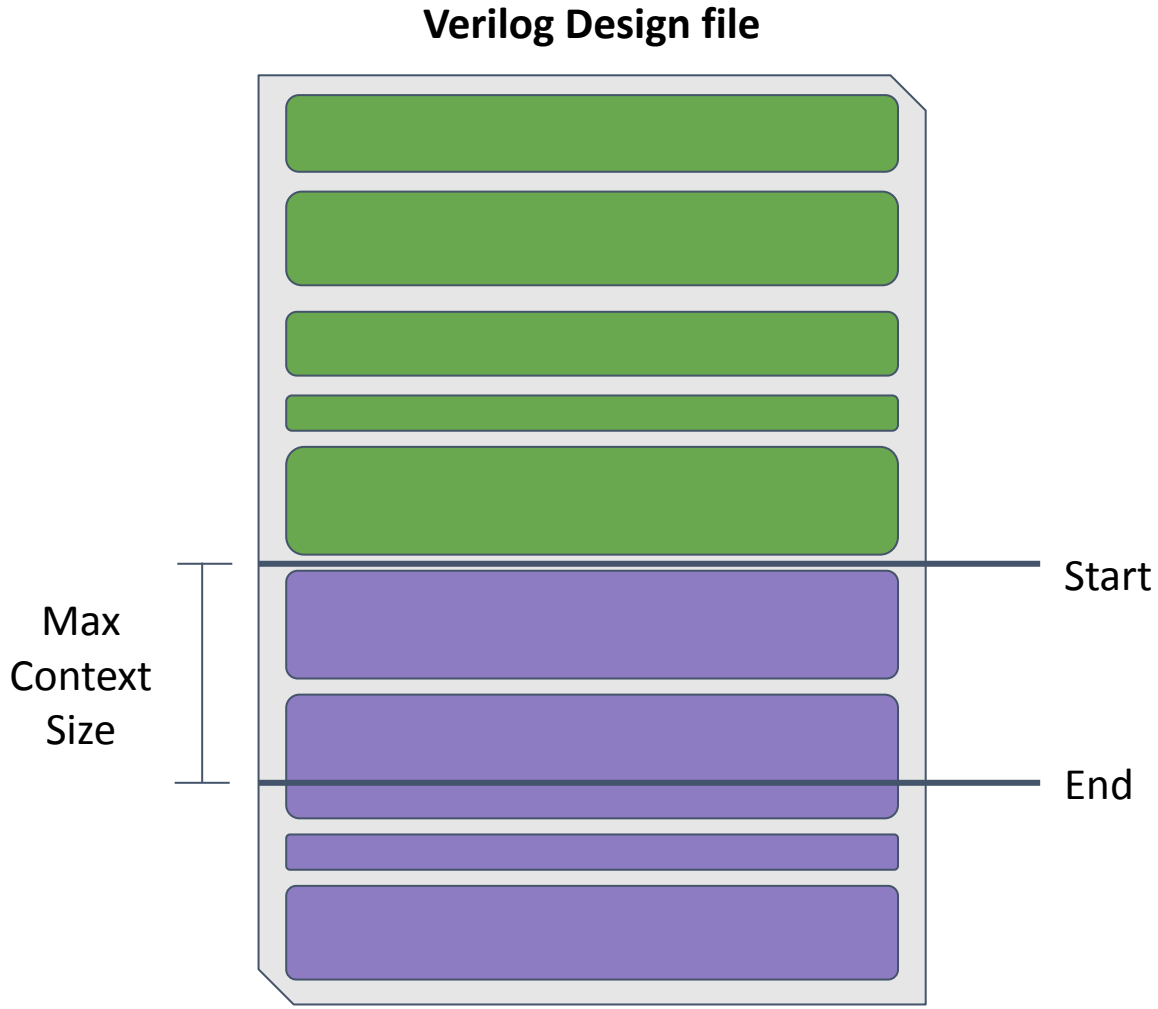


LLM Module Splitter – Chunking



Now, we run into an issue where the region itself is too big for the context size.

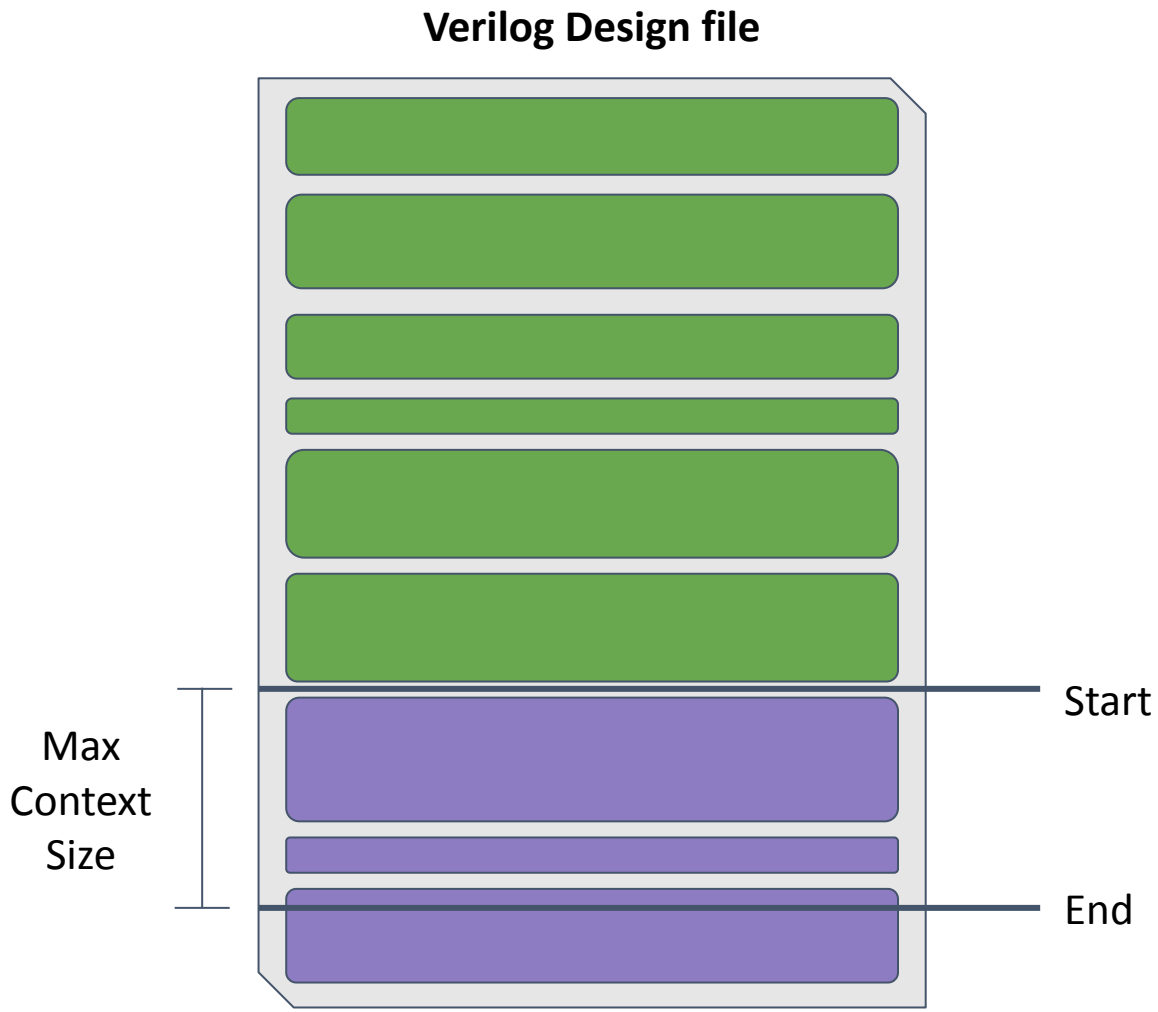
LLM Module Splitter – Chunking



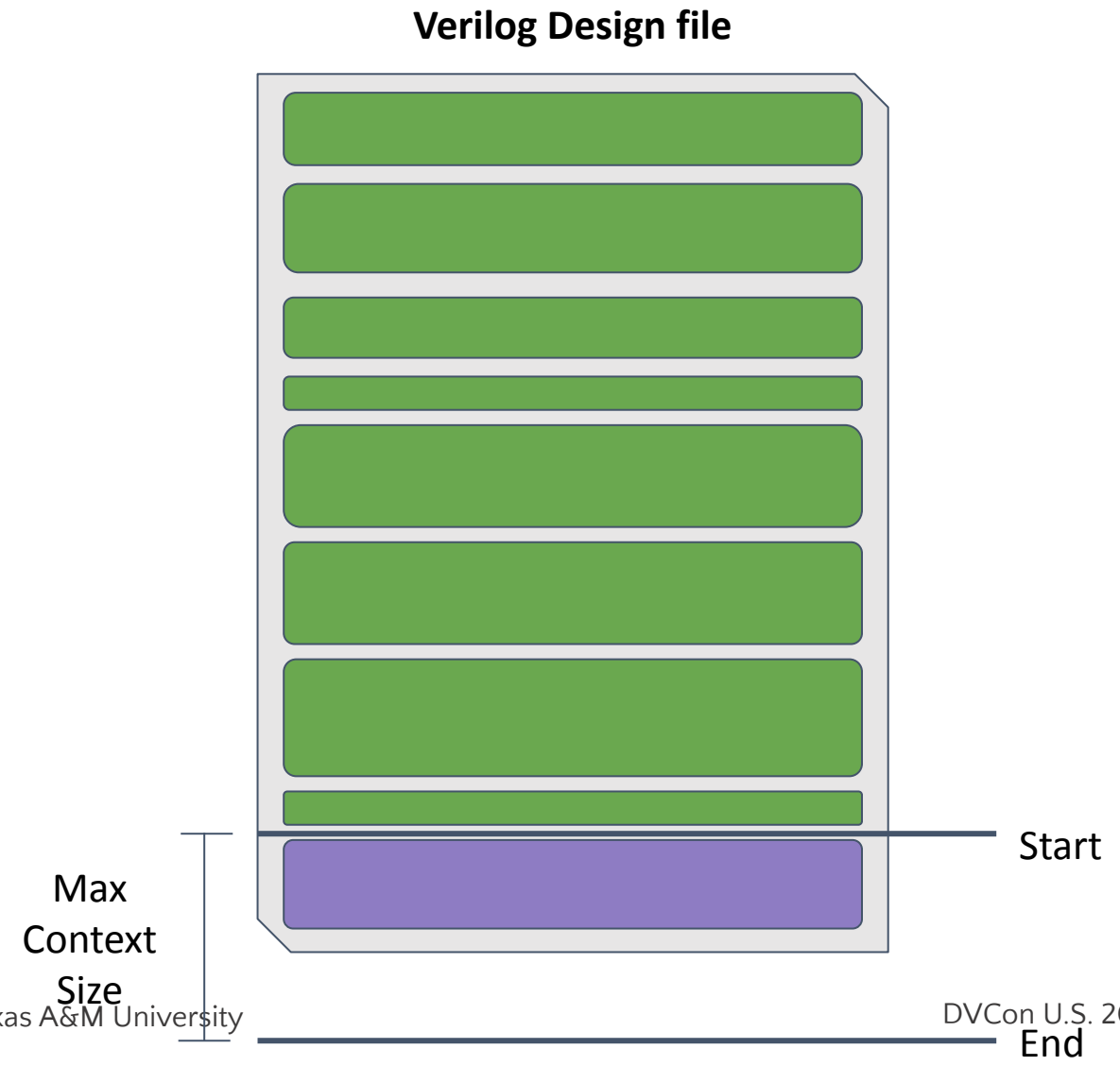
Now, we run into an issue where the region itself is too big for the context size.

Solution: This will actually entirely be avoided as the LLM will simply construct regions within its context size (automatically splitting larger intuitive regions into smaller intuitive subregions).

LLM Module Splitter – Chunking

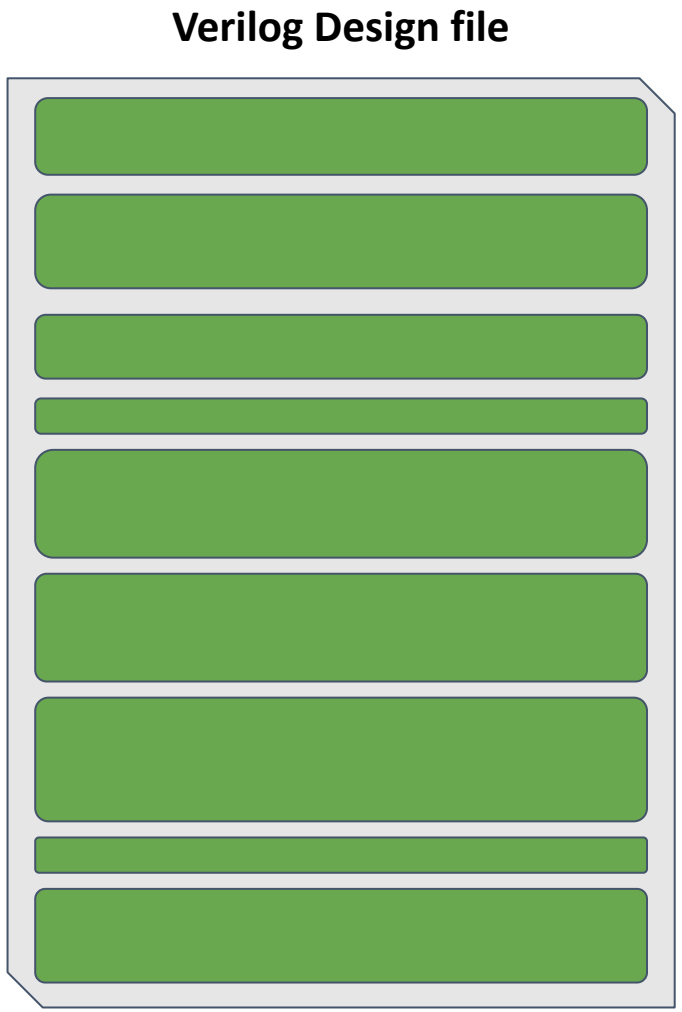


LLM Module Splitter – Chunking



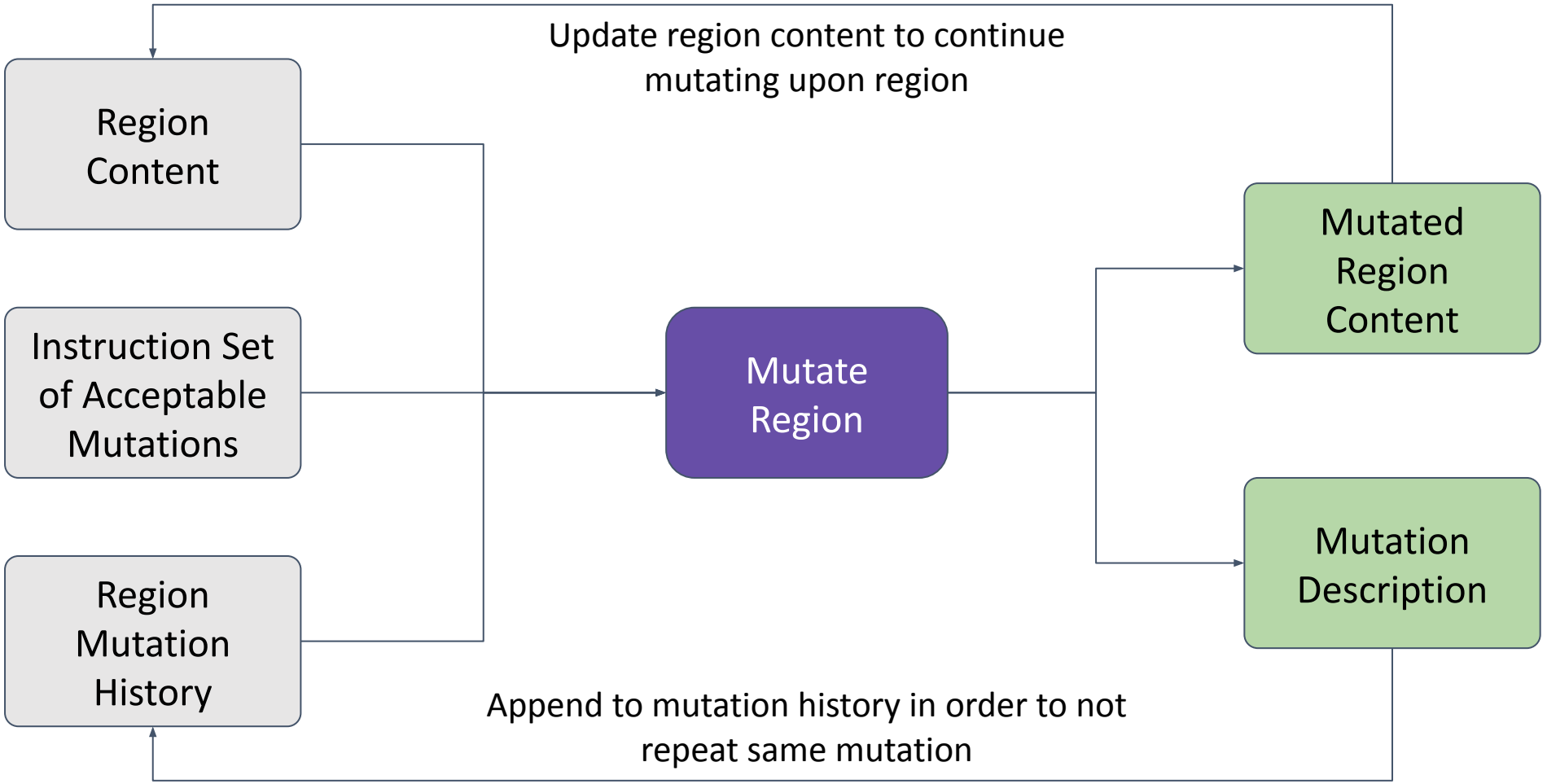
File is appended with EOF token so LLM knows it is not expected to parse further.

LLM Module Splitter – Chunking

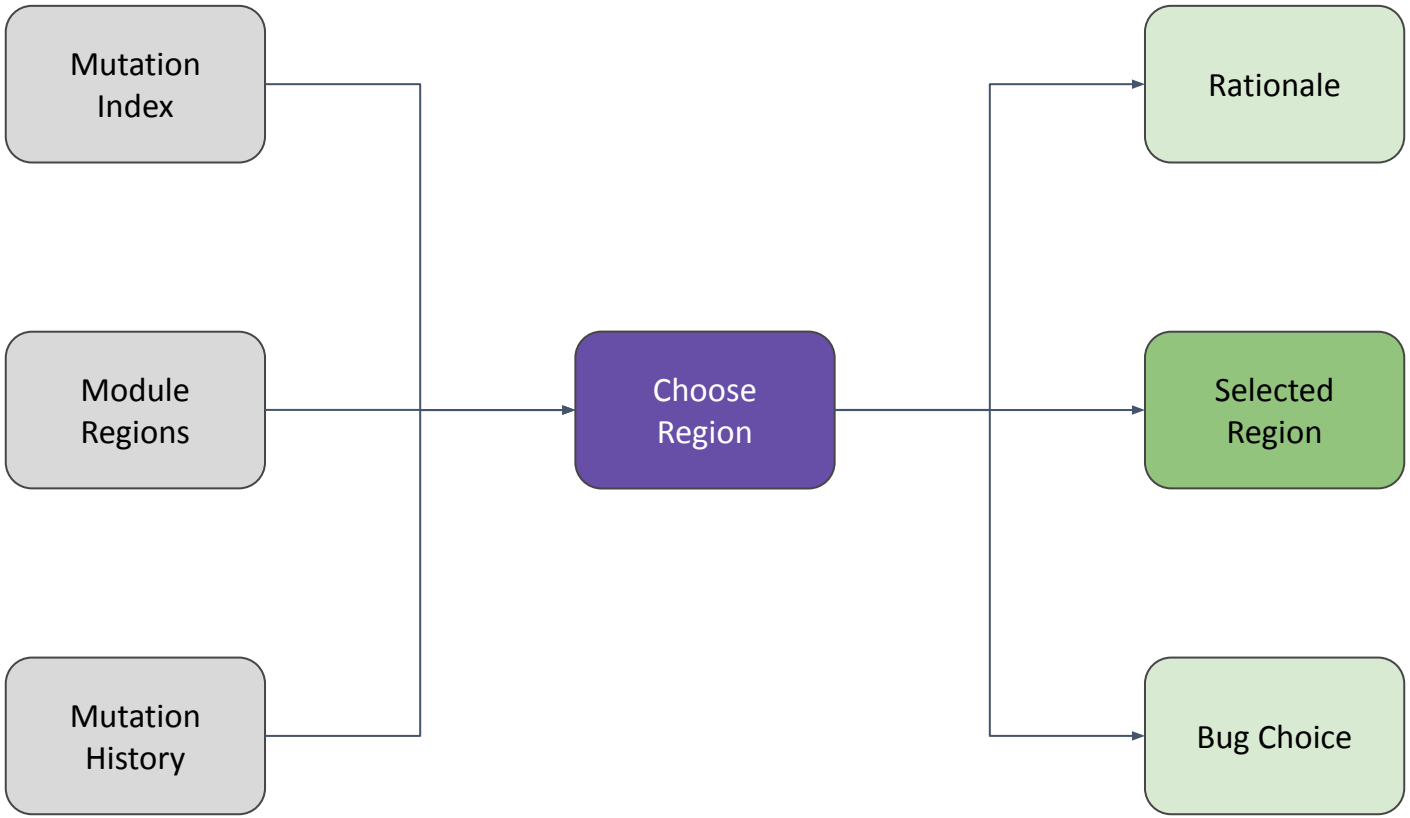


We have now split the verilog design into 9 separate regions. For each, we can perform regional mutation.

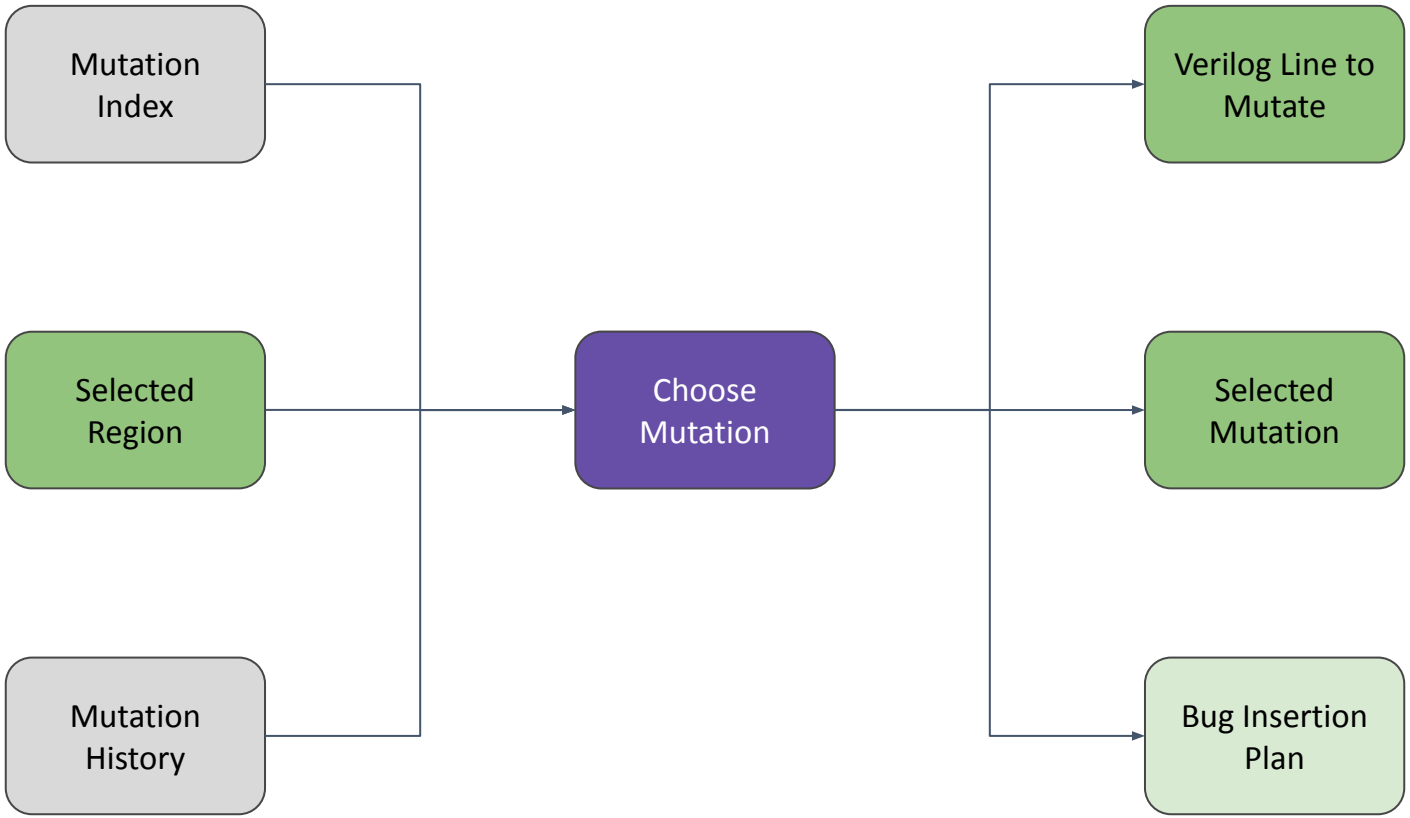
LLM Region Mutator



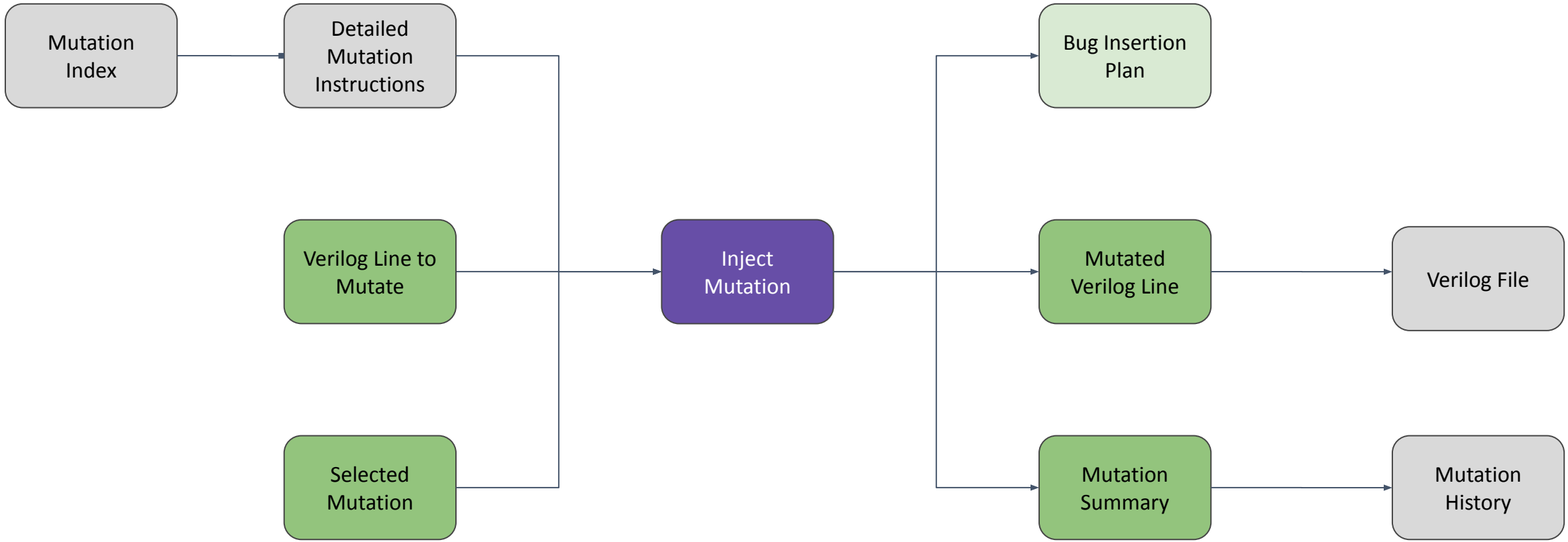
LLM Region Mutator – Step 1: Choose Region



LLM Region Mutator – Step 2: Choose Mutation



LLM Region Mutator – Step 3: Inject Mutation



Demo

Live Demo of Automatic Bug Insertion using LLMs

Download the demo video

↓

SCAN ME

Coverage Acceleration

Overview

Motivation

Random test generation, even with constraints, rarely achieves full coverage

- Generates many redundant test cases that hit already-covered points
- Struggles to reach complex corner cases
- Coverage typically plateaus well below 100%

Directed testing for last-mile coverage is resource-intensive

- Requires deep understanding of design internals
- Engineers must manually craft specialized test cases
- Often takes more than half of total verification time

Machine learning-guided test generation offers key advantages:

- Learns patterns from existing test coverage data
- Intelligently targets uncovered functionality points
- Achieves higher coverage in less time than traditional methods

Methodology: Last-Mile Test Generation

Relevant Variable Identification

- Uses Random Forest for feature selection
- Identify most relevant variables for each functionality group
- Reduce the search space for test generation
- Transform problem into ML feature selection task

Constraint Learning

- A GAN (Generative Adversarial Network) is trained on previously successful test cases
- The GAN learns to generate valid test inputs
- Implicitly capture complex constraints

Methodology: Test Generation Process

Initial Phase

- Random test generation for easy-to-cover points
- Collection of training data for ML models
- Identification of last-mile functionality points

LMT Phase

- Iterative process targeting uncovered points
- Prioritizes groups with most uncovered points
- Combines RF and GAN models for test generation
- Validates and executes generated tests

Variables Pruning using Random Forest

1. Extract log files from the regression
2. Extract AST
3. Extract Cone of influence

Build and relationship between covered cases and the variables.

What variables are contributing to the covered cases?

e.g. if a covergroup gets 80% of functional coverage, then record the variables participate in that coverage.

- Label variable with a probability (0 to 1).
- Variables labeled with a probability above the threshold of **0.7** are considered more for covered cases.

Experimental Setup

DUT: OpenTitan AES hardware IP

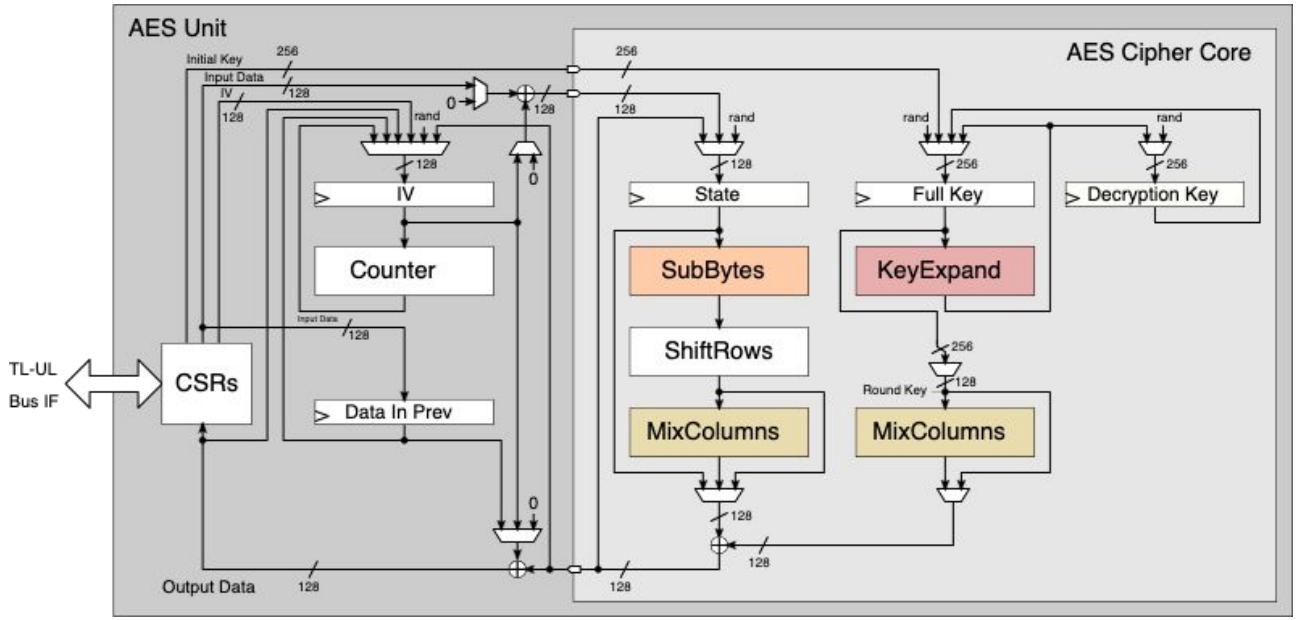
Testbench: UVM compliant testbench

Tools

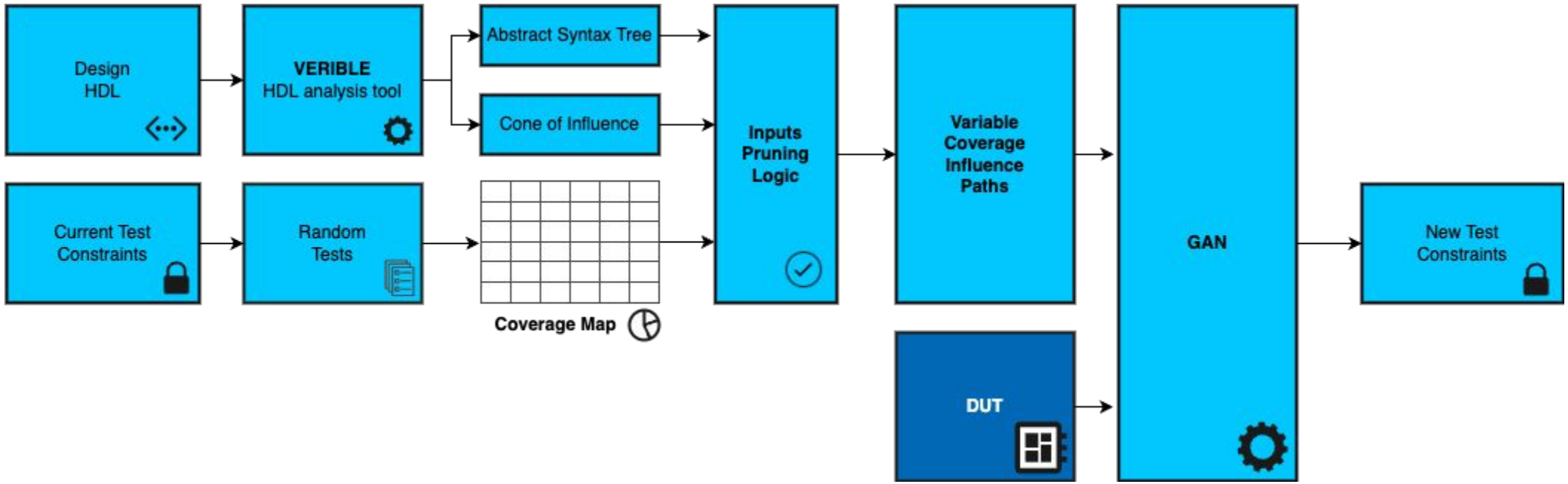
- Verible
- VCDCAT

ML Framework

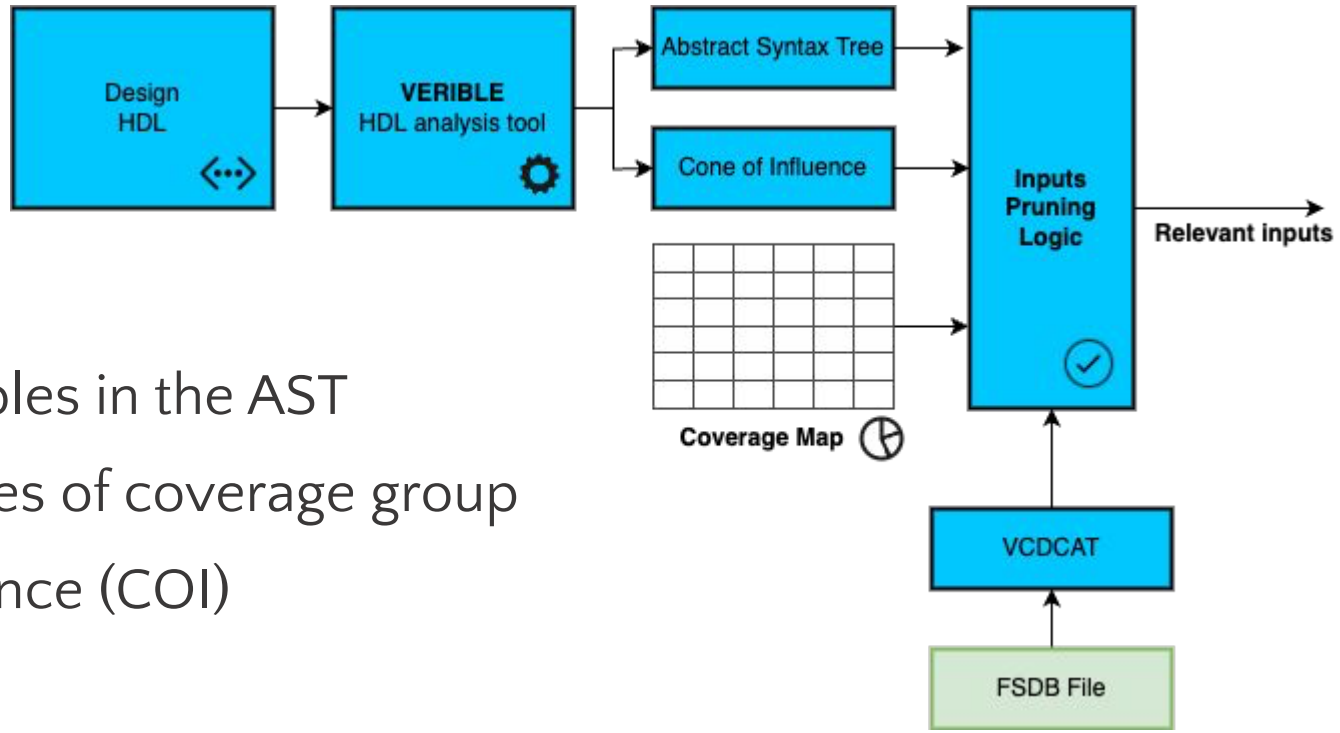
- Generative Adversarial Network (GAN)



Workflow


Design
HDL
VERIBLE
HDL analysis tool
Abstract Syntax Tree
Cone of Influence
Inputs
Pruning
Logic
Variable
Coverage
Influence
Paths
GAN
New Test
Constraints
Current Test
Constraints
Random
Tests
Coverage Map
DUT

Workflow: Relevant Variable Identification

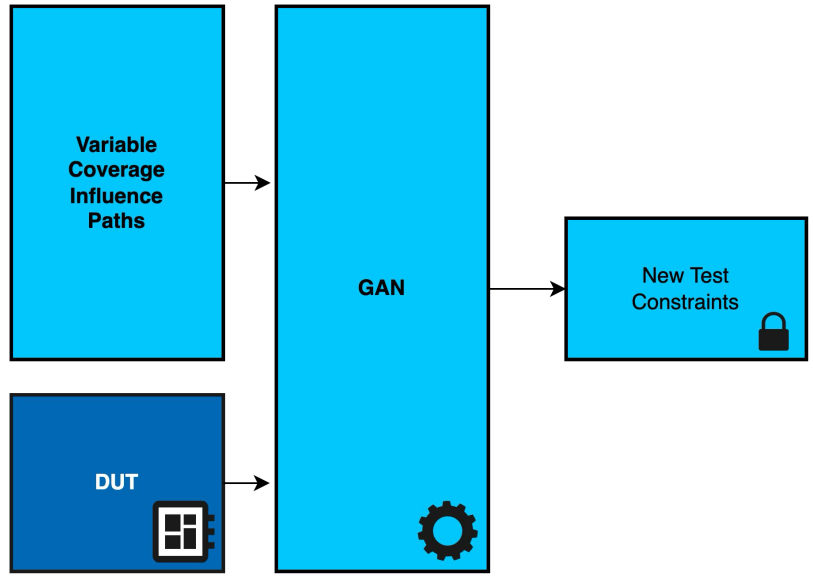


1. Label the signals related to the variables in the AST
2. Label the variables related to the holes of coverage group
3. Label the variables with low importance (COI)

Workflow: GAN for Constraint Learning

A GAN (Generative Adversarial Network) is trained on previously successful test cases

The GAN learns to generate valid test inputs



Assertion Generation using LLMs

Overview

Motivation

- Writing assertions by hand is time-consuming and requires deep expertise
- Translating natural language hardware specifications to formal, precise verilog assertions is subjective and error-prone when performed manually
- More assertions lead to broader coverage, but engineers are limited by constraints
- The use of LLMs to generate assertions can save time, reduce errors, and improve coverage

Methodology – Overview

Assertion Generation

- LLMs have shown great promise in generating high fidelity verilog assertions for manually generated and LLM-generated RTL
- We rely on a 2-step approach to produce assertions based on a hardware feature table with added context about SystemVerilog Assertions and formal verification techniques

Evaluating Assertion Fidelity

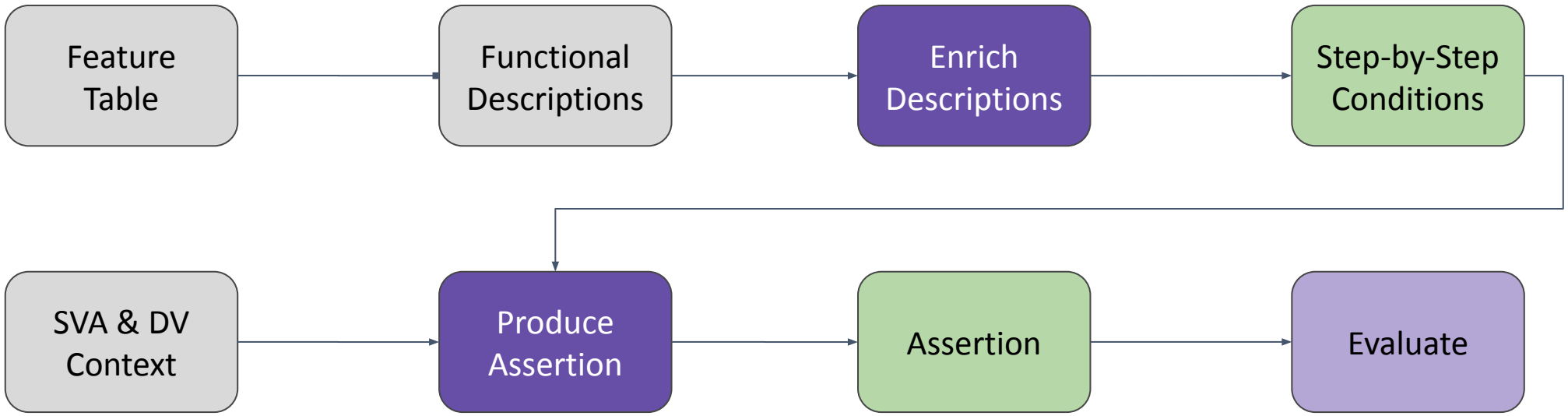
- No standardized method of evaluating an assertion quality
- Utility: Gauged through a separate LLM query
- Syntactical correctness: re-prompt LLM if generated assertion fails to compile
- Functional correctness: JasperGold FPV
  - Check if assertions pass for known bug-free designs
  - Check if assertions fail for known buggy designs

Methodology – Step 1: Start with Hardware Feature Table

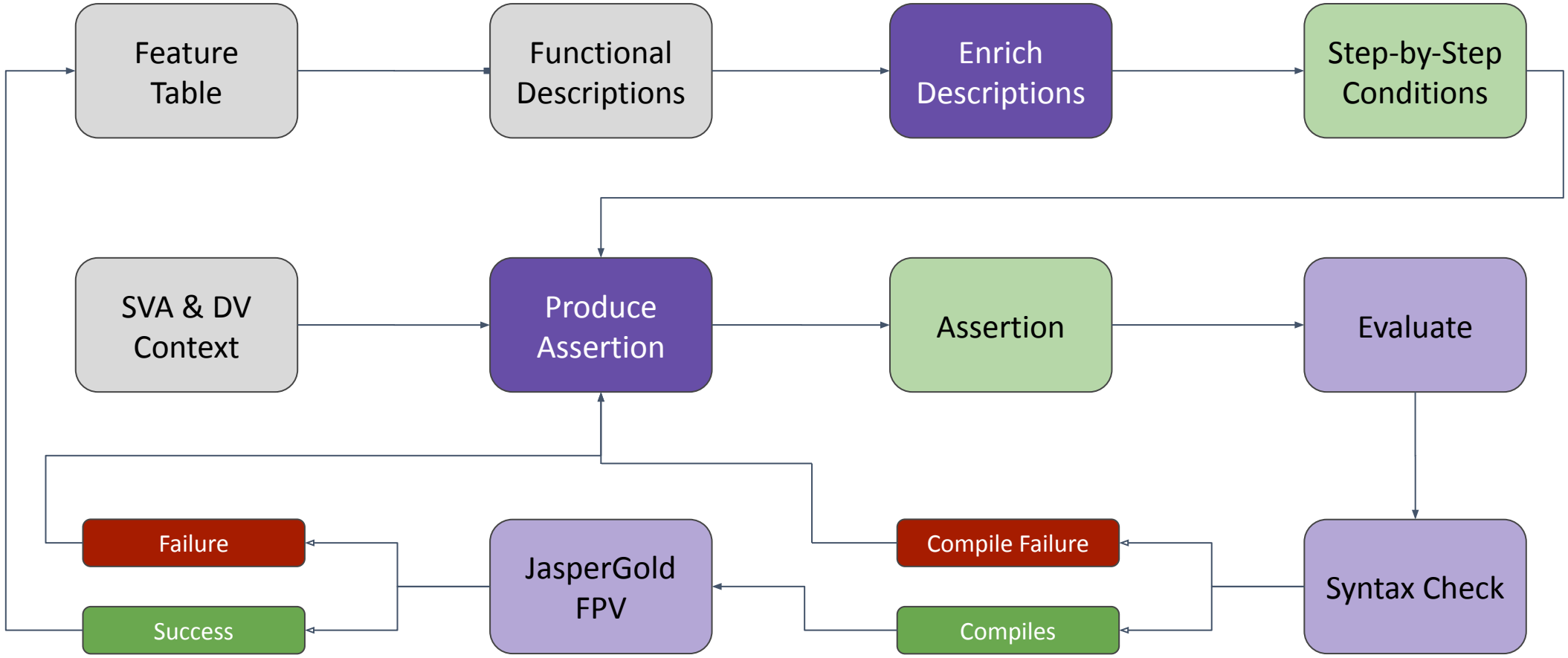
Feature ID	Feature	Description
F1	I2C Protocol Compliance	Ensure compliance with the I2C protocol (START, STOP, ACK/NACK, data transfer).
F2	7-bit and 10-bit Addressing	Proper handling of 7-bit and 10-bit addressing modes.
F3	Multi-Master Arbitration	Handle arbitration loss in a multi-master environment.
F4	Clock Stretching	Ensure proper handling of clock stretching by slower slaves.

Methodology – Step 2: Enrich Functional Descriptions

Methodology – Step 3: Generate Assertion



Methodology – Step 4: Evaluate & Revise



Demo

Live Demo of Assertion Generation using LLMs

Download the demo video

↓

SCAN ME

TEXAS A&M UNIVERSITY

Test Bench Generation using LLMs

Overview

Motivation

Manual efforts are time-consuming, costly, and error-prone

- Can take months for engineers to write detailed stimulus, checkers, & assertions manually
- Human-generated testbenches can contain subtle mistakes or overlook vulnerabilities, leading to false positives / negatives and incomplete coverage

Traditional testbenches miss edge cases, limiting coverage

- Verification engineers traditionally use directed tests and constrained-random techniques which frequently ignore rare state transitions or unlikely scenarios
- These edge cases are often the source of many functional bugs
- Limited coverage can cause huge problems post-silicon

Manually reviewing log files is time-consuming and scales poorly with hardware complexity

- As hardware designs become increasingly complex, verification data scales exponentially
- Engineers must scrutinize huge volumes of massive log files to identify patterns & root cause

Motivation (continued)

LLM automation can significantly improve efficiency & coverage

- Automatically analyze hardware specifications, existing testbenches, and verification plans to produce structured testbenches that align with goals for coverage
- Generate a set of **diverse** stimulus, assertions, & checkers
- Adaptable: allows for rapid testbench updates as design evolves

LLMs are more robust at testbench generation for AI-generated RTL

- AI-generated RTL is becoming more common
- Often features unconventional structures and optimizations
- LLM-generated testbenches are better suited for these paradigms than traditional methods

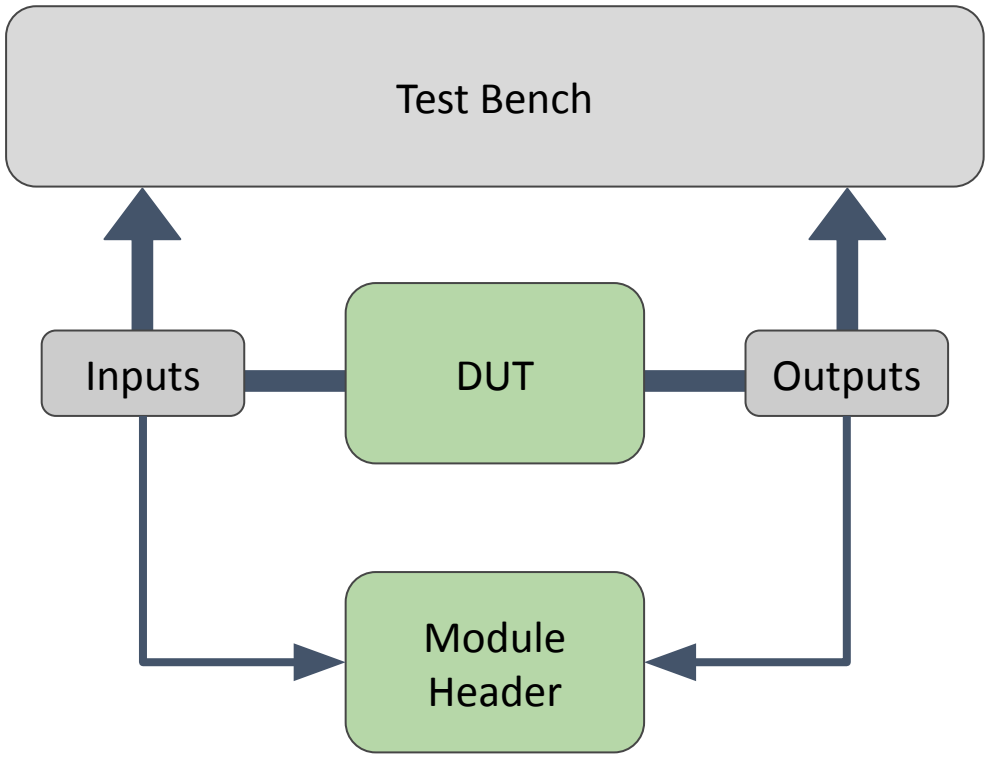
Methodology – Step 1: Prepare Inputs

Provide Design Under Test (DUT)

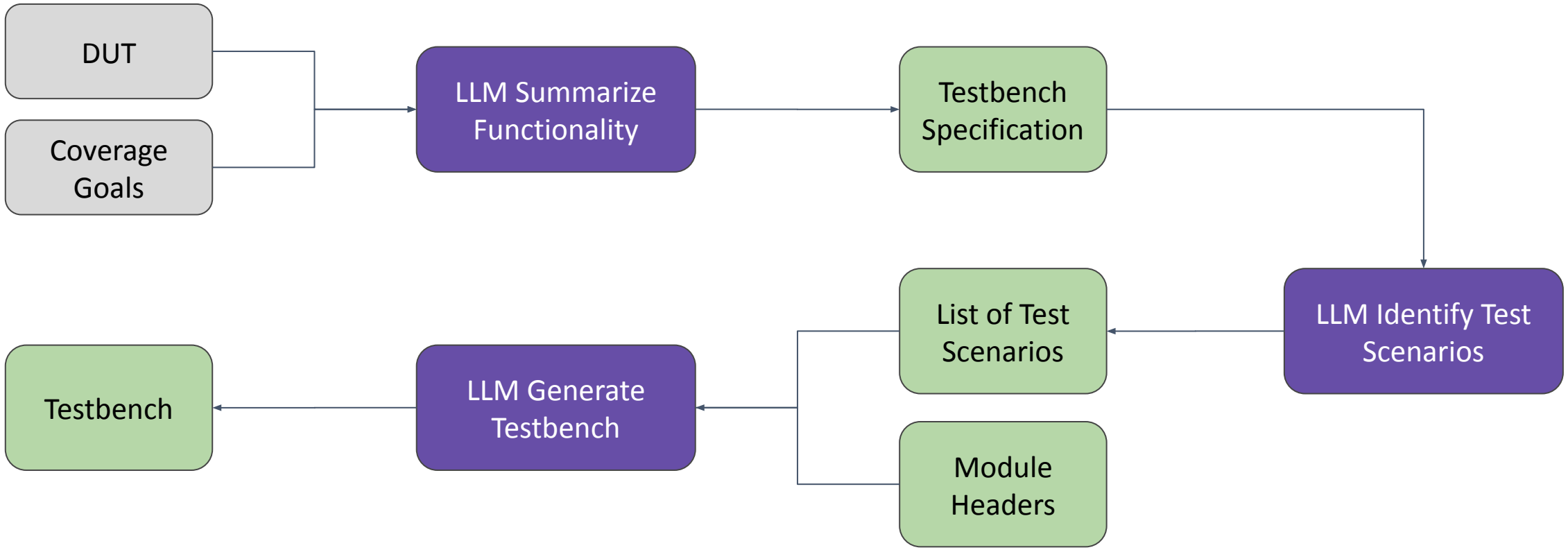
- High-level RTL description

Extract module headers

- Defining I/O ports of the DUT



Methodology – Step 2: Produce Testbench Driver using LLM



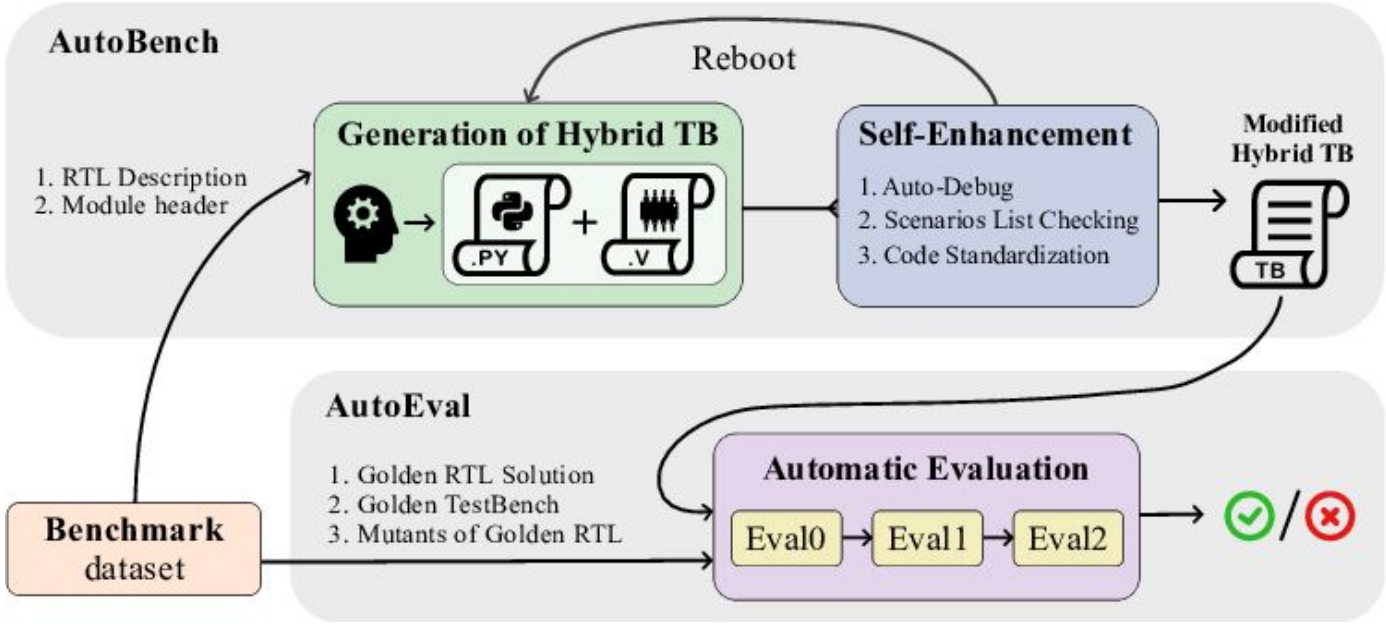
Methodology – Step 3: Evaluate Testbench

Check Syntax

- Ensure the testbench compiles successfully

Functional Correctness

- Run testbench on known bug-free RTL implementation to ensure it passes
- Run testbench on buggy RTL, derived from golden standard, & produce coverage report



Demo

Live Demo of Test Bench Generation using LLMs

Download the demo video

↓

Promotional Statement on Texas A&M University WAVE-CHIP Program

WAVE-CHIP (Workforce Advancement in Verification and Evaluation of Chips) is a **Texas A&M University**'s program to expand the verification workforce. The initiative addresses the shortage of verification engineers by:

1. Developing verification curriculum for community colleges and 4-year universities
2. Creating industry partnerships for hiring pipelines
3. Providing upskilling programs for working professionals

The program brings industry expertise into academia, leveraging Texas A&M's established verification curriculum.

Please contact wavechip@tamu.edu to partner in this critical mission.

TEXAS A&M UNIVERSITY

NSTC WORKFORCE PARTNER

TEXAS A&M UNIVERSITY

Thank you!