Accelerating Pre-Silicon Verification Coverage with Transaction Sequence Modeling

Jayanth Raman, Jackson Wydra, Ximin Shan, Rahul Krishnamurthy,
Michael Yan, Phyllis Hsia, Vikram Narayan, Samir Mittal
Micron, Inc.
San Jose, CA 95134
{jayanthraman, jwydra, ximinshan, rkrishnamurt, binyan, phyllishsia, vnarayan, samir}@micron.com

Abstract

Design Verification (DV) is a crucial stage in the development of integrated circuits. It employs simulation-based techniques to identify and rectify design errors early, prior to silicon implementation, thereby conserving resources through a shift-left strategy. Universal Verification Methodology (UVM) is a standardized framework used in DV. UVM transaction sequences define the simulation that is executed as part of a DV test case. This paper presents a method that uses UVM transaction sequences to train machine learning models that predict the likelihood of hitting a code or functional coverage item. These models are then used to assess if new UVM sequences will increase coverage, and simulations are run only if an increase is likely. This approach saves time and resources by avoiding simulations unlikely to boost coverage, ensuring efficient resource utilization. The technique was applied to the coverage of two Designs-Under-Test (DUTs). The results demonstrate that transaction sequences can be used as input to train machine learning models to predict code and functional coverage. Additionally, the results show coverage acceleration for one of the DUTs.

Index Terms: Design verification, Integrated circuits, Hardware design languages, Pre-silicon, Machine learning.

INTRODUCTION

Design Verification (DV) is a vital phase in the lifecycle of integrated circuit product development. It primarily employs simulation-based testing, which, despite its popularity, is known to be time-intensive and demands substantial engineering resources. The primary objective of DV is to identify and rectify design flaws at the earliest stages, preferably before silicon implementation. This shift-left approach helps to avert or at least mitigate expenses associated with late-stage bug detection and resolution.

Coverage, both code and functional, is a standard metric in DV used to quantify the extent of testing performed on the design. Performing exhaustive testing on industrial designs to achieve full coverage is generally infeasible. Constrained Random Testing (CRT) [1] is a technique used in DV to generate a limited but wide range of test scenarios by applying specific constraints to random inputs. Selecting tests that are more likely to increase coverage and running the simulation only for those tests optimizes machine time, reduces verification duration, and potentially improves coverage metrics. We use the term "test" to refer, in CRT terminology, to a single run of a flavor for a specific seed value. A flavor refers to a specific set of constraints applied to randomized data, essentially defining a particular variation or pattern within the wider range of possible random inputs.

DESCRIPTION

The method presented in this paper identifies whether a set of transactions that a test case is about to execute as a simulation is likely to increase coverage or not. If the set of transactions pertaining to a test case are determined not to increase coverage, then that simulation is not run, thereby shortening verification time. Universal Verification Methodology (UVM) [2] transaction sequences, representing a test case, are used to train Machine Learning (ML) models, which are designed to predict the likelihood of covering a code or functional coverage item. These trained ML models are used to assess whether a new set of UVM sequences would likely result in increased coverage. The new set of UVM sequences are generated as part of the regular constrained-random based DV flow. The actual simulation is only continued if coverage is likely to increase. Given that simulations can be time-consuming, avoiding those unlikely to boost coverage leads to significant savings in time, as well as machine and engineering resources. This strategic approach ensures efficient utilization of resources.

Each transaction sequence can be visualized as a 2D matrix, where the columns represent the fields in the sequence (e.g. buffer ID, key, etc.) and each row corresponds to a timestamp. The two sequences were first merged based on the timestamps of the items in the sequence. This is shown in Figure 1. Each field was represented in binary form, expanding into as many columns as there are bits in that field. For example, an eight-bit field results in eight columns. Additionally, a binary column was added to indicate whether the item originated from the write sequence or the read

Write Transaction Sequence

Timestamp	BufferID	Key	X	Y
—	—	—	—	—
—	—	—	—	—
—	—	—	—	—

Read Transaction Sequence

Timestamp	BufferID	Key	X	Z
—	—	—	—	—
—	—	—	—	—

Merged Transaction Sequence

Cmd	Timestamp	BufferID	Key	X
0	—	—	—	—
1	—	—	—	—
0	—	—	—	—
0	—	—	—	—
1	—	—	—	—

Figure 1. Example transaction sequences (shown here as Write and Read sequences). The timestamps are used to merge the two sequences. An additional column is added with an entry of 0 for write and 1 for read.

sequence. The 2D matrix was then flattened into a 1D vector by concatenating the rows of the matrix in order. Thus, each test generates one 1D vector and a corresponding output label for a give line or coverpoint, forming one sample input-output pair.

TRANSACTION SEQUENCE MODELING METHODOLOGY

There are two phases in the application of this methodology. In the first phase, full simulations are run, UVM transaction sequences and corresponding coverage data are gathered, and machine learning models are trained from the gathered data. In the second phase, the trained models are applied to the UVM transaction sequences generated as part of a test (i.e. a single run of a flavor with a specific seed) to decide whether to run the simulation or not. Imagine that a test or simulation run consists of two parts: generating the sequence(s) and then running the simulation only if the sequence(s) are deemed likely to increase coverage.

A. Phase 1 (Train)

Fig. 1 depicts the steps involved in Phase 1. UVM transaction sequences are generated as part of regular constrained random testing. Simulations are run and coverage metrics are collected for these simulations. The collected

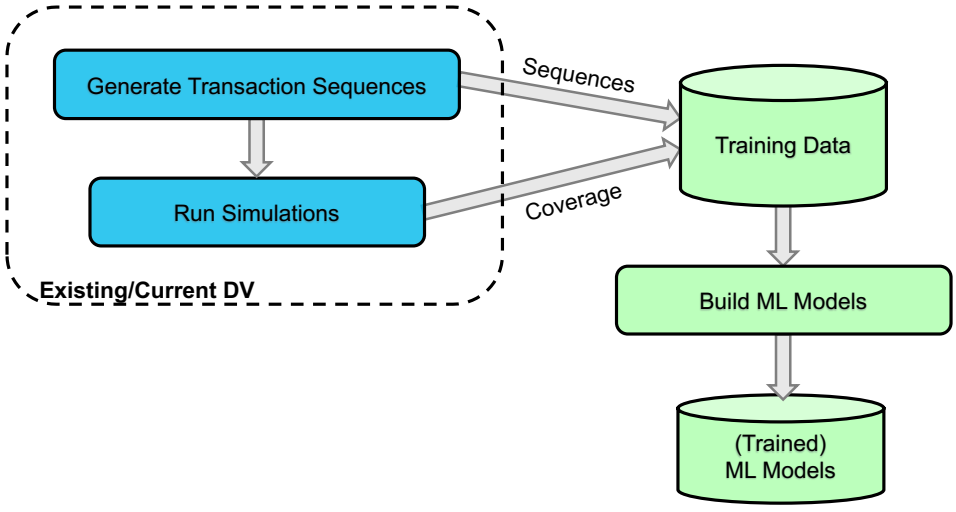

Figure 1. Phase 1 (Train) of the Transaction Sequence 1 Flow.

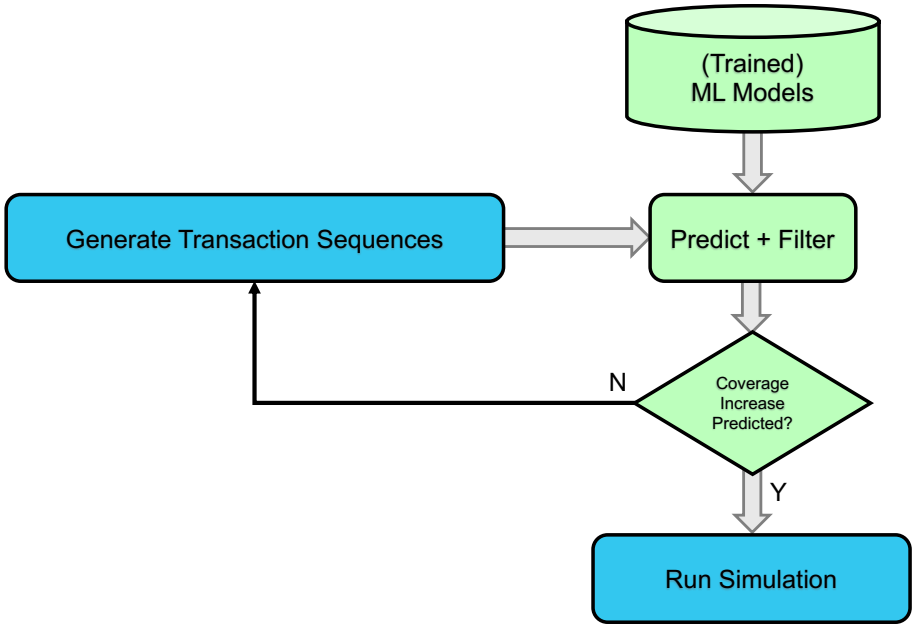

Figure 2. Phase 2 (Apply) of the Transaction Sequence Modeling Flow.

transactions sequences and coverage metrics are used to train machine learning models such as Gradient Boosted models (e.g. XGBoost [3], LightGBM [4]) or Neural Network models [5]. Transaction sequences form the input to the ML models and the binary coverage for the specific line or functional cover bin is the output. Transaction sequences can be composed of different fields. Fields were selected based on having a variance above a threshold (e.g., 80%). Fields were either integer valued or categorical [6]. Integer-valued fields were converted to a binary vector. Categorical fields were converted into indicator variables. Fields were then stacked horizontally. Multiple transaction sequences (e.g. write and read) were interleaved based on their timestamps. When multiple transaction sequences were used, only fields common to both were used. The ML task is classification. Models that meet a minimum performance criterion (e.g., based on area-under-the-curve (AUC) [7] or other ML metric threshold(s)) are retained. The criteria for ML Models that are retained is a tradeoff between the number of simulations that are run and the potential to miss out on covering hard to hit lines or functional cover bins. Hence, the criterion can change from one application to another. For example, retaining models with very low AUC leads to more simulations being performed. Conversely, retaining only models with very high AUC could result in too few simulations, potentially missing certain hard-to-hit lines or functional coverage bins. The trained ML Models are saved for later use in Phase 2. As more tests are run over time as part of the DV process, the trained ML Models may be updated or retrained from scratch.

B. Phase 2 (Apply)

Fig. 2 depicts the steps involved in Phase 2. When running a test case, transaction sequence(s) are generated as a first step using existing DV tests. The DV tests are typically constrained-random tests. Signal constraints, if any, are applied prior to the generation of the test sequences as part of the constrained-random verification setup. Before the tests are simulated, the transaction sequence(s) are input to the previously saved trained ML Models. For transaction sequences for each test, the models are used to predict coverage of the target line or target functional cover bin. For a single line or target bin, the ML model predictions are used to determine if the simulation is to be run or not. If the transaction sequence for a test is predicted to likely not increase coverage, then the simulation for that test is not run, saving simulation time and resources needed to run the simulation.

If there are multiple lines or functional cover bins to be covered, then the coverage is represented in the form of a binary coverage vector. The total coverage is the sum of the elements of the vector. Given a budget of K simulations,

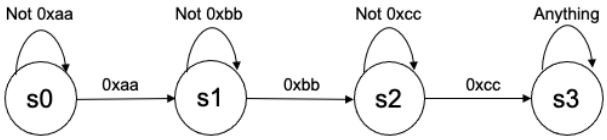

Figure 3. State transitions of a Lock DUT. State s3 is the Unlock state.

N (with N >> K) sequences are generated. The N sequences are ranked and K of them are selected. The ranking and selection strategies would be based on what works best for a given DUT. Depending on the DUT, many strategies are available for picking the K sequences including, (a) picking the top K sequences by total coverage, (b) picking the bottom K sequences by total coverage, (c) a combination of (a) and (b), (d) selecting K sequences based on whether the predicted coverage includes or does not include an instance in the path of execution leading to an uncovered item, (e) maximizing diversity by selecting K vectors that are as different (e.g., by Hamming distance) from each other as possible, etc.

RESULTS

A. Lock DUT

Lock DUT is a Register-Transfer Level (RTL) design that accepts a sequence of code and unlocks only if the sequence matches a predefined pattern. Many Lock DUT variations are possible. The state transition diagram of one such variation of a Lock DUT is shown in Figure 3. The bit width of transactions for this lock is eight. The maximum number of transactions is also set to eight. The line corresponding to s3 can be reached only if the code 0xcc is received after previously first receiving 0xaa and then receiving 0xbb in that order among the previous transactions. For example, the sequence {0x11, 0xaa, 0x22, 0x33, 0xbb, 0x44, 0x55, 0xcc} or the sequence {0xaa, 0x23, 0xbb, 0xcc} would unlock the DUT and cover the s3 line. The DUT poses a significant coverage challenge, more so if the bit width of the codes is large. For example, if the maximum number transactions are three and the bit-width of each transaction is 8 bits, the probability of unlocking with transaction sequences generated uniformly at random is one in about 16.8 million. If the maximum number of transactions is four, then the probability is one in about 5.8 million.

An XGBoost ML model was trained to predict coverage of the line corresponding to state s1 (Phase I). Each input was a sequence of length eight and the corresponding output label was a 0 or 1 depending on whether the target line was covered or not by that sequence. The model was trained on 10K transaction sequences generated uniformly at random with a 70/30 training/validation split. In Phase 2, more transaction sequences of length eight were generated

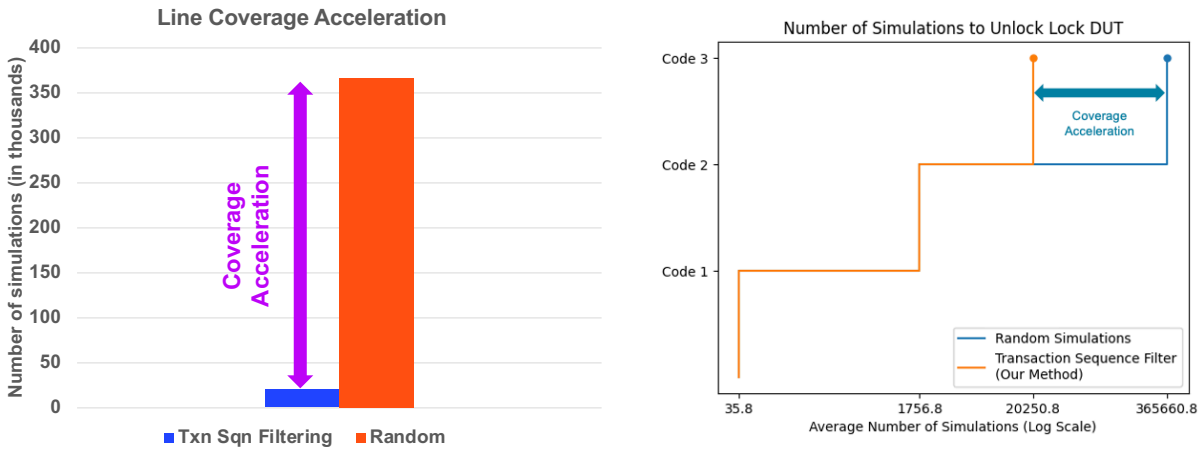

Figure 4. Line coverage acceleration compared to random simulations for the Lock DUT.

TABLE I

Results for line coverage of the Cache DUT. A model was trained for each line. A line is a unique (Instance, Line-Number). The last column (% Tests) is the percentage of tests that cover the corresponding lines. Models for many lines have high AUC values, demonstrating excellent predictive performance for those lines.

	Module	Mean Best AUC	AUC Range	Unique LineN	Unique Inst.	# Unique (Inst., LineN)	% Tests
Poor	Module 1	0.50	0.50—0.51	1	3	3	70%-77%
	Module 2	0.51	0.51—0.51	1	1	1	82.10%
	Module 3	0.51	0.50—0.52	25	1	25	59%-87%
	Module 4	0.51	0.50—0.52	2	2	3	79%-87%
	Module 5	0.51	0.51—0.51	3	1	3	80%
	Module 6	0.56	0.50—0.81	1	48	48	56%-90%
	Module 7	0.64	0.50—0.99	4	5	7	60%-87%
	Module 8	0.70	0.50—0.99	3	5	15	60%-87%
	Module 9	0.75	0.50—1.00	7	20	50	65%-90%
Good	Module 10	1.00	1.00—1.00	1	12	12	75%-76%
	Module 11	1.00	1.00—1.00	1	231	231	54%-56%
	Module 12	1.00	1.00—1.00	7	1	7	54%-56%
	Module 13	1.00	1.00—1.00	7	32	224	54%-56%
	Module 14	1.00	1.00—1.00	1	3	3	75%-76%
	Total			**64**	**365**	**632**	

uniformly at random and each of these sequences were input to the ML model which predicted if the transaction sequence would cover the line corresponding to state s1 or not. Then, simulations were run only for those sequences for which the ML model predicted would hit state s1.

For a code length of 8 and a sequence length of 8, about 366K random simulations are needed on average to unlock the DUT. Using the method described in this paper, about 20K simulations on average were needed to unlock, resulting in an 18x coverage acceleration compared to random simulations. This is depicted in Fig. 4.

B. Cache DUT

Cache DUT is a Micron internal RTL design written in Verilog. For confidentiality reasons, information about this DUT is not shared here. The design can be conceptualized as being composed of multiple modules. Each module can

TABLE II

Results for functional coverage of the Cache DUT. A model was trained for each cover bin. A covergroup consists of multiple coverpoints, which in turn contain multiple cover bins. Models for many cover bins have high AUC values, demonstrating excellent predictive performance for those cover bins.

Covergroup	Num Coverpoints	Num Bins	Mean AUC	Min AUC	Max AUC
Covergroup 1	1	5	0.80	0.50	1.00
Covergroup 2	8	8	0.50	0.48	0.52
Covergroup 3	3	11	0.78	0.50	1.00
Covergroup 4	9	10	0.51	0.49	0.53
Covergroup 5	1	5	0.80	0.50	1.00
Covergroup 6	2	2	0.51	0.51	0.52
Covergroup 7	1	2	0.50	0.49	0.51

be instantiated multiple times, meaning a module can have several instances. A coverage line is uniquely identified by an instance and a line number. When a set of tests are run, if none or very few of the tests cover a line, then there isn't enough data to model that line. On the other hand, if all or most of the tests cover a line, then it is not of interest since it is already covered. Hence, for all the coverable lines in the design, lines that are covered by 10% to 90% of the tests were chosen to be modeled.

Several tests were run as part of the regular DV flow. The testbench was instrumented to save the transaction sequences. Each test generated two sequences: one for write commands and one for read commands. Coverage metrics were collected for each test. Then, for each line under consideration, an XGBoost, LightGBM, and other models were trained for a classification task using the collected transaction sequences as input and the coverage of that line as the output. For a given line, the ML model with the highest AUC was retained as the best model. In most cases either an XGBoost model or a LightGBM model was the best ML model. Additionally, for comparison, we trained a DummyClassifier [8]. The DummyClassifier always resulted in an AUC near 0.5 and is not shown in the results.

Table I presents the results of the best ML models, aggregated by modules in the DUT. The AUC (Area Under the Curve) is a standard ML metric that ranges from 0.5 to 1.0, with higher values indicating better performance. Lines with a high AUC (e.g., greater than 0.9) are selected for the next phase. It can be observed that many lines are well-modeled (AUC close to 1.0), while others are not modeled effectively (AUC close to 0.5).

Table II shows the results of the ML models aggregated by functional covergroups. A functional covergroup contains multiple coverpoints, which in turn contain multiple cover bins. Like line coverage, only bins that are covered by 10% to 90% of the tests were chosen to be modeled. Models for some of the bins perform well and are chosen for the next phase. About 28% of the functional cover bins have a high prediction performance.

SUMMARY

This work shows that transaction sequences can be used as input to train machine learning models to predict code and functional coverage. This work also demonstrates coverage acceleration using transaction-sequence models. We showed transaction-sequence modeling for two DUTs – Lock DUT and Cache DUT. Line coverage was predicted for both the Lock DUT and the Cache DUT, and functional coverage was predicted for the latter. For the Lock DUT, we used the trained models to select transaction sequences to run the simulations. Compared to random simulations, our method achieved an 18x acceleration.

REFERENCES

[1] A. B. Mehta, "Constrained Random Test Generation and Verification", In: *Introduction to SystemVerilog*. Springer: Cham, 2021, pp 325—407. https://doi.org/10.1007/978-3-030-71319-5_13.

[2] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in *IEEE Std. 1800.2-2020 (Revision of IEEE Std. 1800.2-2017)*, vol., no., pp.1-458. 2020.

[3] T. Chen, and C. Guestrin. "XGBoost: A Scalable Tree Boosting System." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785-794, 2016.

[4] G. Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." *Advances in Neural Information Processing Systems* 30, pp. 3146–3154. 2017.

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer. 2006.

[6] "Categorical variable." https://en.wikipedia.org/wiki/Categorical_variable (accessed Dec. 11, 2024).

[7] "Classification: ROC and AUC." https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc (accessed Dec. 11, 2024).

[8] scikit-learn DummyClassifier documentation. URL: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html (accessed Dec. 11, 2024).