# A Scalable Gray-Box Instance-Based Reachability Predictor for Automated DV Regression Scheduling

**Coverage closure is the process of maximizing all the coverage metrics in a Design Under Test (DUT). However, the goal of 100% coverage is hard to achieve due to the large stimuli space and the vast number of simulations required to identify all the possible simulation paths reaching the different DUT modules instantiated. Herein, we propose an Instance-Based Reachability Predictor Model able to guide the test selection and simulate tests expected to contribute to the desired coverage improvement, thus avoiding redundant tests. Moreover, we propose an optimal batch regression scheduling approach to address traditional machine learning limitations applied to heavily parallelized simulation flow. The proposed methodology achieves 32% time saving with respect to Constrained Random Verification (CRV) saturation time and utilizes 40% less resources with an average improvement of 0.74% code coverage and 2.74% functional coverage.**

## I. INTRODUCTION

The traditional Design Verification (DV) flow relies on the execution of many different parametrized tests affecting the stimulus provided in input to a DUT. DV engineers create tests and simulate them across the entire lifecycle of a project. A test is created to exercise different portions of the DUT functionality, by controlling directly and indirectly the input stimulus. The use of constrained random variables, random variables combined with user-defined constraints, is a common practice to ensure specific design functionalities to be exercised by a test while at the same time guaranteeing randomization of the stimulus. These tests, and a set of parameters (plusargs) affecting the stimuli generation via CRV, are defined in Testbenches (TBs).

In the lifecycle of a project, DV engineers first aim at maximizing metrics such as functional coverage and assertion coverage for a high-level assessment of the design functionality, and only in the later phase of the project aim at maximizing code coverage metrics such us line, conditions, branch, finite state machine (FSM), and toggle coverage. Ideally, the verification of a design is completed once all the code and functional coverage metrics reach 100%. Nonetheless, this goal is extremely hard to achieve due to the high complexity of the Register Transfer Level (RTL) digital logic and the large input sampling space. To directly exercise portions of a design, directed tests are created ad-hoc by verification engineers to further narrow the constrained input sampling space, executing a lower number of simulations to reach a higher coverage target. However, designing directed tests is a challenging task requiring significant time and human resources.

To help DV engineers address their need, in this work we propose a Reachability-based Test Scheduler to guide the test selection based on the ability of a test to reach hard-to-cover portions of a design. A Machine Learning (ML) model is trained to learn, from the test parameters (plusargs), the ability of a test to reach (cover) a specific RTL instance. Then, an optimal regression scheduler is used to allocate tests over a regression, while dynamically changing the coverage target over time to achieve coverage closure. The scheduler identifies the most effective tests required to reach uncovered portions of a design, and similar to a directed testing scenario, leverages this information by allocating the tests that are estimated to potentially improve coverage.

Our contributions are the following:

a) a Reachability Predictor Model (RPM) used to predict the ability of a given test to reach (cover) a target RTL design instance,
b) a Hierarchical Ranking Strategy ordering the test identified by the predictor RPM in order to accelerate coverage closure,
d) a Test Regression Scheduler optimizing the test allocation across regression batches, balancing the allocation of tests accounting for available resources such as processors, licenses, and simulation time.

Given an initial regression, our methodology relies on test plusargs to train a model, learning the ability of a test to reach specific design instances. The trained model is leveraged to predict the test targeting the rare instances–the hard-to-reach, and partially covered ones. Tests most likely able to reach such instances are selected for a new regression. Then, selected tests are scheduled balancing the resource usage and avoiding bottlenecks related to long running tests. Once the new regression data are collected the model is updated and the process is repeated until a target coverage closure, or a time budget is reached.

## II. State of the Art

In the past years, the problem of automated DV has been extensively researched and different methodologies have been proposed to accelerate coverage closure and replace standard CRV flows typically adopted in the industry. Two main categories of methodologies can be identified: the first aiming at directly generating input stimulus, and the second utilizing the existing verification environments, through available TB knobs such as plusargs, constrained spaces etc. To the first category belong approaches leveraging static and dynamic based analysis for test generation [1-3]. These methodologies rely on concolic testing, SMT/SAT solvers and leverage the design property extracted at compile time combined with dynamically generated coverage information. While effective for small designs, these approaches are often unable to scale for large industrial designs. On the other hand, methodologies utilizing the existing TB (e.g., by directly controlling the input constrained space) usually rely on ML models to estimate the impact of plusargs on the resulting coverage. These approaches aim at identifying the most promising tests or plusargs in order to avoid redundant simulations and schedule new tests. Among these methods, different type of ML models have been explored, such as Bayesian Networks [4][5], Genetic Algorithms [6] and Neural Networks [7].

More recently Huang et al. [8] proposed Multi Armed Bandit (MAB) based regression to automatically configure the plusargs with the goal of maximizing the coverage. This approach leverages Bayesian MAB to identify and select the best test configuration to simulate without any knowledge of the DUT. Similarly, Jayasree et al [9] proposed to employ decision trees to estimate the test coverage and simulation time and schedule new regressions.

Our approach, similarly, to Huang et al. [8] and Jayasree et al. [9] leverages model predictions to generate new regression batches, but differently from previous works, leverages the DUT property and hierarchical information, to design an instance-based reachability prediction model estimating the effectiveness of the test over specific targets of the design hierarchy. Moreover, we propose a ML-aware scheduling strategy which accounts for the training, data extraction, and inference of the trained model combined with the parallelization capabilities of industrial DV clusters. Thus, we can provide a solution effectively able to address the needs of real industrial settings and large-scale designs.

## III. Problem Formulation

The goal of our methodology is to identify the optimal set of tests to schedule over a regression in order to maximize a target coverage metric while minimizing the number of required simulations. Due to the intractability of the input space and the complex logic of the RTL design, reaching 100% coverage across all the possible metrics is often infeasible. Thus, verification engineers provide waivers, in the form of exclusion files, to exclude specific coverage targets and constrain the reachable portion of a design. Moreover, due to project timelines, the maximum number of simulations that can be run is often limited to a user defined budget or to a specific timeframe. In this work, we refer to the maximum reachable coverage, or coverage closure, as the coverage that can be obtained after accounting for all the user-specified exclusion files. Similarly, our simulation budget refers to the weekly regression. In this work, a set of tests simulated over a 48-hour timeframe is considered as a weekly regression. This is a common scenario for design verification engineers who run longer batch of simulations over a weekend during ongoing development process.

Verification engineers create a set of different tests *flavors* F to exercise specific design functionality. Moreover, they define *plusargs* P to parametrize the test execution. For each *flavor* in F, a user defined subset of *plusargs* can be leveraged to affect a specific *flavor* functionality. The simulation of a *flavor* and its associated set of *plusargs*, generates a coverage report. The coverage report can be represented, for each coverage metric, as logical vector $c$ of zeros and ones describing the coverage obtained for each coverage target (line, assertion, cover point, branch, etc.). The cartesian product among the set of all the available *plusargs* P, combined with the user-defined flavors constraints, and the set of available *flavors* F defines the input *test* space T, such that $T = P \times F$.

We model *flavors* and *plusargs* using categorical and numerical values to encode the input features. *Plusargs* preprocessing is straightforward. For each numerical *plusarg* integer and binary values are used, while the categorical ones are represented using one-hot encoding. Then, for each *plusarg* having numerical values, data are standardized using mean normalization. Means standardization subtracts the mean and divides the *plusarg* values by it to make all feature values fluctuate around the mean and scaling to unit variance. To define the target classes, we introduce the notion of reachability. Reachability is used to identify the ability of a test to reach a specific set of targets in the coverage vectors. The Methodology section discusses the details regarding the notion of reachability and its use in the context of the model prediction to schedule batches of regression.

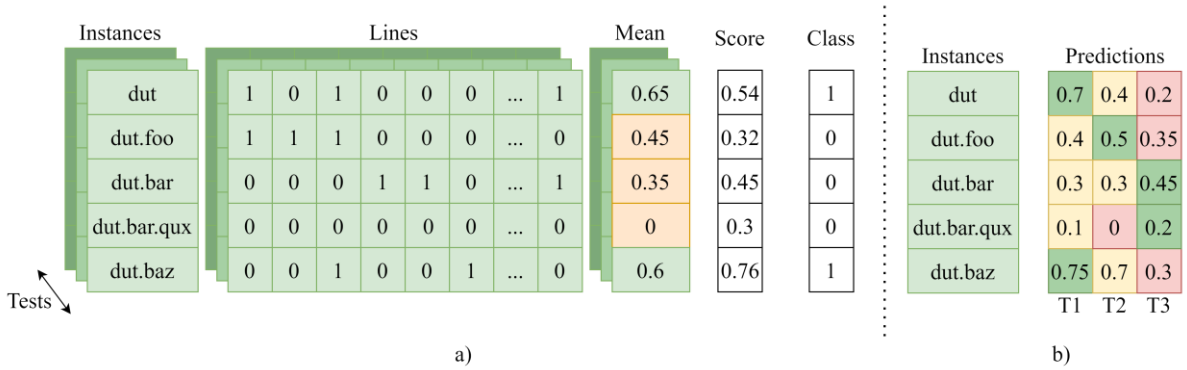


Figure 1. a) For a given number of instances, the mean of coverage information associated to each instance is calculated. Then, the mean coverage for each instance across the test history is derived. This is termed as reachability score. A user defined threshold, in our case of 0.5, it is used to convert scores into classes. b) Prediction sample for the instances from the RPM, showing the confidence of the model of reaching an instance for a given set of plusargs (test).

## IV. Methodology

### A. *Reachability Predictor Model*

In our proposed methodology, we introduce the notion of *reachability*. In a typical testbench, sets of *plusargs* are defined for a given *flavor* of a test. The simulation of such under different seeds leads to different simulation output generating different coverage vectors. We combine the coverage information obtained from the simulations with the RTL design hierarchy to define *reachability* as the probability vector that a *test* t has of covering one or more targets in the design hierarchy. Targets can be defined at different levels of granularity according to the desired metric of interest. While considering code coverage metrics we target instance, and, when considering functional coverage metrics, we target coverpoints. Thus, when selecting a *test*, we aim at selecting one such that it will be the most likely to reach a target instance *i*. Given a history of past simulations and an RTL design hierarchy, for each *test* t, we can identify the probability $R_{i,t}$ associated to each target *i* of being covered by the test *t*, as:

$$R_{i,t} = \frac{1}{N_t} \sum_{k=0}^{N_t} \frac{\sum c_{i,k}}{|C_i|}, \tag{1}$$

where $N_t$ is the number of times test *t* is simulated, $C_i$ is the set of line associated with instance $i \in I$ with *I* being the set of all possible instances, and $c_{i,k}$ is a coverage vector associated to lines, conditions, branches, FSM, assertions, or coverbins for instance *i* obtained from a given simulation *k* of a test *t*. For the rest of this document, we discuss the methodology as referring to code coverage in RTL instances. However, note that the same approach may be similarly applied to coverpoint bins for functional coverage reachability.

Example: Assume that a design has two instances, instance `dut.foo`, and instance `dut.bar`, and both are covered by a single test simulation. Instance `dut.foo` has 10 lines, and only 8 of them are covered during the simulation, while instance `dut.bar` has 25 lines and 10 of them are covered. We say the instance `dut.foo` is 80% reachable by the provided test and instance `dut.bar` is 40% reachable by the test. Across different seeds—simulation of the same test flavor—we average the *reachability* percentages. Figure 1a) shows an example of design hierarchy having five different instances. For each test, the line coverage vector is averaged to associate a coverage score for each instance.

Thus, we can define $\vec{R}_t$ as the reachability vector associated to a *test* t reaching all the available instances *I* as follow:

$$\vec{R}_t = \begin{Bmatrix} R_{i,t} = 1, \ if \ R_{i,t} \geq 0.5 \\ R_{i,t} = 0, \ if \ R_{i,t} < 0.5 \end{Bmatrix}, \qquad \forall \, i \in I \tag{2}$$

Based on the previous example, for test *t*, according to Eq. (2) the instance `dut.foo` is reachable (because 8/10 > 0.5), while instance `dut.bar` is not (10/25 < 0.5).

The goal of our RPM is to predict, given input feature vector of *plusargs* defining a test flavor f, the reachability vector $\vec{R}_t$, such that:

$$\mathrm{RPM}(\mathrm{f}, \mathrm{p}) = \vec{R}_t, \tag{3}$$

where *f* is a specific test *flavor*, and *p* is the associated set of *plusargs* defining the *test.*

Through the RPM inference, we can approximate the reachability of an RTL hierarchy target provided an input *test*—a combination of *plusargs* and *flavor*. Thus, we can leverage historical data to evaluate the impact of input

features in reaching specific regions of a designs and approximate the expected coverage. Using these predictions, we can identify which *tests* to select to reach a given target instance and schedule a batch of tests to simulate over a regression while avoiding noncontributing ones.

Example: given the predicted test vectors in Figure 1b), test $t_3$ is predicted as the most likely to reach instance `dut.bar`, and `dut.bar.qux`, on the other hand, it is not a good candidate for the remaining instances.

### B. *Rarity-Based Hierarchical Traversal Strategy*

Given that RPM is able to predict which *test* is more likely to reach a desired target, a ranking strategy is necessary to define the ordering of the *tests* that will be simulated over a regression. Identifying the right ordering is important since it allows us to minimize the number of redundant *tests* over the same regression batch. Our ranking strategy provides the test ordering aiming at maximizing the cumulative coverage as well minimizing the number of tests required to reach the maximum coverage. Our ranking is defined by the following optimization function:

$$\text{Objective: } \min_{N} \left( \max \sum_{N} c_k \right), \tag{4}$$

where $N$ is the total number of simulations, and $c_k$ is the coverage vector resulting from simulation $k$ of a test.

We leverage RPM predictions $\overrightarrow{P}$ to accelerate coverage closure based on historical data by selecting *tests* having the highest chances to maximize the cumulative coverage. Each element $p_i$ of $\overrightarrow{P}$ represents the probability of instance $i$ of being reached. Given such requirement we rank *tests* according to their ability to reach as many targets as possible. Then, we introduce a threshold to filter the less effective tests by selecting the top-k performing tests. For each prediction vector $\overrightarrow{P_t}$ generated estimating the reachability of test $t$, we apply the thresholding from Eq. 2 to obtain a binary vector of zeros and ones $\overrightarrow{P'_t}$. Then, by concatenating all the $\overrightarrow{P'_t}$ vectors we can define the matrix $P_{T \times I}$ where $T$ is the number of tests and $I$ the number of instances, representing all the test predictions. Top-k tests are thus defined as:

$$K = \text{Top}_k \left( \underset{t}{\text{argmax}} \sum_{i=0}^{I} P_{i,t} \right) \quad \forall\, i, t \;\; s.t. \;\; i \in I, t \in T \tag{5}$$

where top-$k$ tests are identified sorting the column-wise sum of the matrix $P$.

Example: given three tests $t_1$, $t_2$ and $t_3$ and *reachability* probability vectors $p_1, p_2$, and $p_3$, for the DUT's instances in Figure 1 `dut`, `dut.foo`, `dut.bar`, `dut.baz`, and `dut.bar.qux`, with $p_1$ = [0.2,0.6,0.75,0.5,0.2], $p_2$ = [0.5,0.6,0.7,0.4,0.65], $p_3$ = [0.1,0.3,0.5,0.4,0.8], post equation (5) becomes $p'_1$ = [0, 1, 1, 1, 0], $p'_2$ = [1, 1, 1, 0, 1] and $p'_3$ = [0, 0, 1, 0, 1] respectively. The ranking strategy will identify $t_2$ followed by $t_1$, followed by $t_3$ as the optimal test selection given that $t_2$ has the highest chances to cover more targets as compared to $t_1$ and $t_3$, and $t_1$ has higher chances than $t_3$.

While the RPM provides information for instances that were reached by past simulations, there may be targets that were never reached–not even a single line for that given instance was covered. For these instances, which have no coverage information associated, we explicitly leverage the RTL design hierarchy. By traversing the design hierarchy backward—from the bottom to the top—we can identify the closest reachable instance and select the tests able to reach such instance as the most likely ones to simulate in order to reach the original uncovered portion of the design. The set of tests $H_i$, identified as the set of candidate tests able to reach an instance $i$ by traversing the design hierarchy is defined as:

$$H = \underset{t}{\text{argmax}}\, P_{i,t} \quad \forall\, i, t \;\; s.t. \;\; i \in X \subseteq I, t \in T \tag{6}$$

Where, $X$ is the set of instances having minimum distance from the target instances having associated coverage information from past simulations. The *argmax* operator across all the test $t$ provides, for each instance $i$, the most likely test able to *reach* the closest instance. The set $X$ is dynamically updated during the regressions once new instances have been reached.

Example: given three tests $t_1$, $t_2$ and $t_3$ and *reachability* probability vectors $p_1, p_2$, and $p_3$, for the DUT's instances in Figure 1 `dut`, `dut.foo`, `dut.bar`, `dut.baz`, and `dut.bar.qux`, with reachability values as $p_1$ = [0.2,0.3,0.75,0.5,0], $p_2$ = [0.4,0.6,0.7,0.4,0], $p_3$ = [0.1,0.3,0.5,0.4,0] respectively, assuming that `dut.bar.qux` is not reached, then the ranking strategy will identify $t_1$ as the optimal test for reaching instance `dut.bar.qux`, being 0.75 the highest probability associated to a test reaching the target instance.

Moreover, in addition to the likelihood to reach a target instance, our strategy aims at prioritizing the selection of hard-to-reach targets. These targets are instance for which only a small portion of the associated logic has been covered. Based on the historical observations, we can evaluate which targets are difficult to reach based on how often—rarely—these are covered. Instances *rarity* is evaluated directly on historical data used to train the model rather than the prediction. The instance ordering based on *rarity* is defined as:

$$I = \left\{ I_{(1)} \leq \cdots \leq I_{(n)} \right\} \tag{7}$$

Where, $I_{(1)} = \underset{I}{\operatorname{argmin}} \left(\sum_T R_{i,t}\right)$ and $I_{(n)} = \underset{I}{\operatorname{argmax}} \left(\sum_T R_{i,t}\right) \quad \forall i \in I \quad \forall t \in T$. This allows to define an ordering among the instances in according to their *reachability* values observed over the training data.
Then, given a *rarity* ordering, we can identify the ordered set of ranked tests *R* for all the instance in *I* defined as:

$$R = \underset{t}{\operatorname{argmax}} P_{i,t} \; \forall \, i, t \;\; s.t. \;\; i \in I, t \in T \tag{8}$$

The only difference between Equation (6) and (8) is the set from which instance *I* is being selected.

Example: given three tests $t_1$, $t_2$ and $t_3$ and *reachability* probability vectors $p_1$, $p_2$, and $p_3$, for the DUT's instances in Figure 1 `dut`, `dut.foo`, `dut.bar`, `dut.baz`, and `dut.bar.qux`, with $p_1$ = [0.2,0.3,0.75,0.5,0.2], $p_2$ = [0.4,0.6,0.7,0.4,0.65], $p_3$ = [0.1,0.3,0.5,0.4,0.8] respectively, assuming that instance `dut` is the *rare* instance of interest, the ranking strategy will select $t_2$ for reaching instance `dut` as compared to $t_1$ and $t_3$.

Lastly, we define the regression batch set *B* as the union among the identified set of tests *K*, *H* and *R*.

$$B = K \cup H \cup R \tag{9}$$

A regression batch *b* is selected from *B* for a given budget of simulations (# sims) to run over a regression. A randomization factor $\eta$ is accounted in the test selection while creating the batch *b* to guarantee that the entire tests population is considered in the overall process, such that:

$$\text{\# sims} = (1 - \eta)\,|\mathrm{b}| + \eta\,|r| \tag{10}$$

where, $|\cdot|$ is the cardinality operator, |b| is the number of tests selected for the batch $b \subseteq B$, r the entire tests population and $|r|$ is the number randomly selected tests given randomization factor $\eta \in [0,1]$.

### C. Optimal Regression Scheduler

While effective in selecting tests targeting rare and uncovered instances, the proposed flow is unlikely to be effective in a real industrial DV pipeline. The use of DV farm, cluster of dedicated machines running multiple simulations in parallel, enable higher simulation throughput compared to ML driven processes due to the inherent sequential nature of learning processes. For each iteration of the RPM flow, and any other ML iterative process, the longest scheduled test will bottleneck a regression batch. Thus, the adoption of a learning driven flow and their effectiveness may lead to poor results if execution time, parallelization factor, resource constraints and coverage information processing time are not considered.

To address this challenge, we have created an optimal regression scheduler implementing a multiple-constrained knapsack algorithm. The algorithm considers the simulation time, the number of available parallel processes, the number of available simulation tool licenses, and the coverage extraction time to schedule an optimal regression batch over these constraints to be able to maximize the estimated target coverage outcome for a given regression. Moreover, in addition to generating optimal batches, instead of waiting for the end of a regression, we train the model over a sliding window timeframe. By training the model every fixed amount of time, we can mitigate the impact of few long-running simulation while constantly updating the model and producing new regression batches.

## IV. Experiments

We have evaluated our model across three different DUTs: the open-source IBEX design [10], an industrial Cache design, and an Industrial Datapath Scheduler Design. The three designs are characterized by 8370, 7166, 131999 lines of code respectively. Each of these designs have 55, 209, 276 different (unique) tests flavors that have been created for testing. For each of the design we have simulated a standard CRV regression flow sampling uniformly 20 seeds for each test. Similarly, we have adopted our methodology to schedule a regression using in total an equivalent number of simulations but with a different allocation of tests over the entire budget. The test allocation is decided by the algorithm. For each DUT we have repeated the experiment twenty times.

### A. Reachability Predictor Model Performance

*Class Distribution.* To evaluate the model performance over the collected data, we have decided to bin the classes (instances) according to their reachability values and evaluate the performance across these bins. This allows to break down the problem in subclasses such as: hard to reach (0%-40%), medium difficulty (40%-85%), and easy to reach (>85%) ones, where the probability range identify how often these instances are reached. Figure 2 (Top-Left) provides the class distribution across all the DUTs given these binning for the code coverage metric. This analysis provides insight about the data distribution and the need to identify a model performing well across all the classes rather than averaging the overall performance.

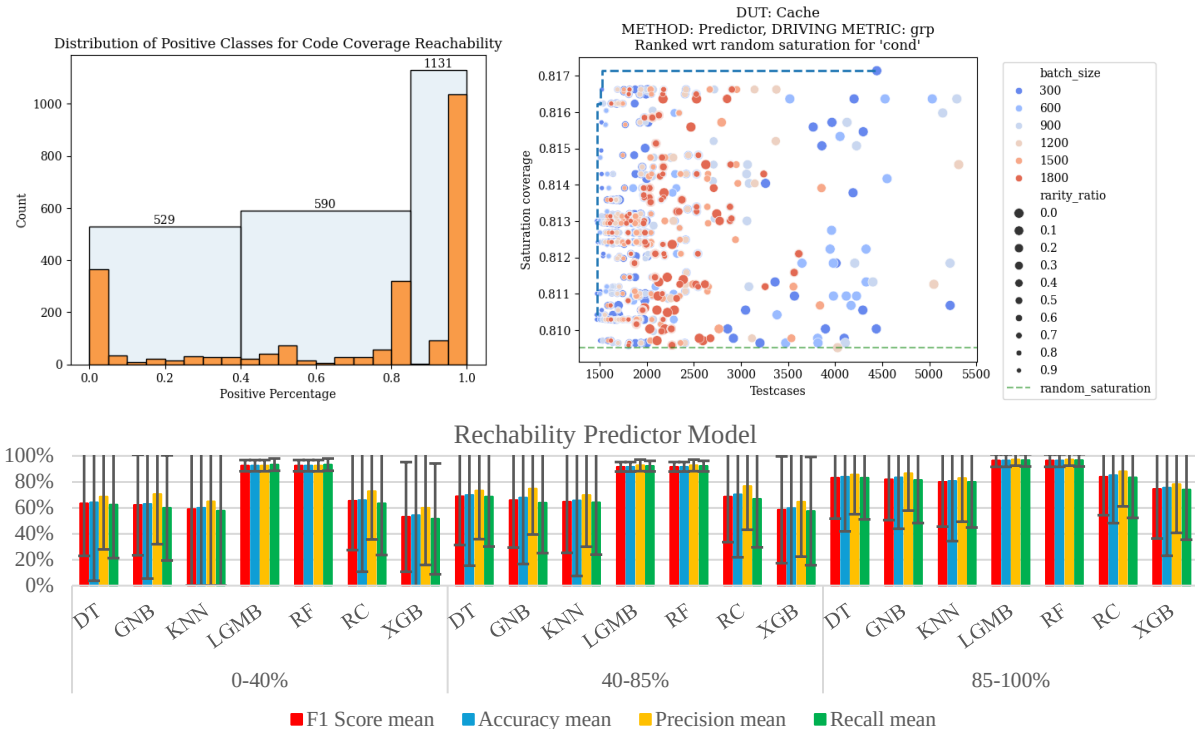


Figure 2. (Top-Left) Instance reachability distribution across the three DUTs. (Top-Right) Example hyperparameter search exploration for batch size and rarity ratio for the Cache design over the group metric. The Pareto-frontier solutions are used to identify the optimal configurations across all the DUTs. (Bottom) Model performance across different type of models for the identified class distributions.

*Model Performance.* We have analyzed the performance of the different classifiers over the identified bin distributions. We have evaluated the performance of Decision Tree (DT), Gaussian Naive Bayes (GNB), K-Nearest Neighbors (KNN), LightGBM (LGBM), Random Forest (RF), Ridge Classifier (RC) and XGBoost (XGB). Figure 2 (Bottom) shows the performance of the different models over the three classes of instances: hard to reach, medium difficulty and easy to reach. Across these classes we can observe how LGBM and RF are the best performing models. They also have negligible difference in the training and inference time. We have adopted LGBM as model for the RPM due to its lower training and inference time.

## B. *Hyperparameter Exploration*

Given the choice of LGBM for the RPM, we have explored the impact of batch size and rarity ratio in Eq. 10 on the overall results, and how these affect the overall performance both in term of final saturation coverage and total number of testcases required to reach saturation. We have simulated the entire flow with different batch sizes (300, 600, 900, 1200, 1500 and 1800), and different rarity ratios (from 0% to 90%), to evaluate the impact of randomness over the flow. Higher values of rarity imply relying more on the RPM and rarity-based traversal rather than random. For each of the coverage metrics evaluated and for each of the DUTs we have identified the set of Pareto-optimal configurations over the testcases and saturation space. Then we have identified the optimal batch size and rarity values that performs better across all the different designs and adopted those for the final evaluation.

Figure 2 (Top-Right) shows an example of hyperparameter search for the Cache design with respect to the conditional coverage metric. The blue dotted line represents the Pareto-optimal frontier across the explored solutions. From the plot it is also possible to observe how the methodology achieves better results than random saturation for all the configurations explored. From the hyperparameter exploration, a batch size of 300 and a randomization factor, rarity_ratio in Figure 2 (Top-Right), of 0.9 have been selected and used for the remaining of the experiments. A *raity_ratio* of 0.9, imply that only 10% of the tests is randomly selected while the remaining 90% is selected by our model. It is important to note that reserving a budget of tests for random selection allows the methodology to potentially reach all the remaining instances, likewise traditional CRV selection, even in case of model misprediction. Nonetheless, the experiment showcases how 10% random selection lead to the optimal results with respect to a higher ratio, highlighting the effectiveness of the model prediction in achieving higher coverage.

Table 1. Reachability Predictor Model and Scheduler (Ranked in the table) performance with respect to CRV coverage saturation and number of simulations required to reach CRV maximum saturation and RPM maximum saturations. Note the Max Sat. column reports the number of simulations required to reach CRV and Ranked saturation respectively. Their maximum achieved coverage is reported in the Coverage Sat. column.

| | | **Coverage Sat.** | | | | **Simulations reaching CRV Sat.** | | | | **Simulations reaching Max Sat.** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DUT** | **Metrics** | CRV | | Ranked | | CRV | | Ranked | | CRV | Ranked | |
| | | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Budget | Mean | Std |
| **DPS** | Assertion | 91.64% | 1.1E-16 | 91.64% | 1.1E-16 | 34 | 41.06 | 22 | 5.08 | 4180 | 22 | 5.08 |
| | Group | 92.27% | 8.4E-04 | **92.69%** | 1.3E-04 | 4051 | 0.00 | 1867 | 344.15 | 4180 | 3357 | 468.69 |
| | FSM | 100% | 0.0E+00 | 100% | 0.0E+00 | 186 | 207.79 | 192 | 36.20 | 4180 | 192 | 36.20 |
| | Branch | 85.53% | 4.3E-04 | **85.62%** | 2.2E-16 | 3032 | 239.30 | 1334 | 123.01 | 4180 | 3692 | 463.30 |
| | Cond. | 91.46% | 1.5E-03 | **91.96%** | 2.2E-16 | 1963 | 34.00 | 357 | 46.41 | 4180 | 3048 | 16.02 |
| | Line | 77.83% | 1.3E-04 | **77.84%** | 1.1E-16 | 3223 | 676.61 | 1347 | 122.08 | 4180 | 3704 | 1019.98 |
| **CACHE** | Assertion | 95.28% | 2.2E-16 | 95.28% | 2.2E-16 | 112 | 127.00 | 439 | 295.50 | 5540 | 439 | 295.50 |
| | Group | 93.89% | 3.0E-03 | **95.11%** | 2.2E-16 | 4921 | 810.28 | 2317 | 133.19 | 5540 | 5032 | 11.47 |
| | FSM | 100% | 0.0E+00 | 100% | 0.0E+00 | 295 | 220.02 | 211 | 119.54 | 5540 | 211 | 119.54 |
| | Branch | 95.98% | 1.2E-03 | **96.53%** | 0.0E+00 | 5496 | 0.00 | 2262 | 47.21 | 5540 | 5495 | 2.49 |
| | Cond. | 81.32% | 8.0E-03 | **84.76%** | 0.0E+00 | 5496 | 0.00 | 2234 | 42.11 | 5540 | 5493 | 2.61 |
| | Line | 99.35% | 6.8E-05 | **99.39%** | 1.1E-16 | 5496 | 0.00 | 2279 | 41.86 | 5540 | 5498 | 2.73 |
| **IBEX** | Assertion | 92.89% | 0.0E+00 | 92.89% | 0.0E+00 | 540 | 11.26 | 641 | 120.34 | 1080 | 641 | 238.46 |
| | Group | 78.04% | 5.3E-03 | **92.85%** | 7.4E-05 | 1076 | 0.00 | 335 | 43.69 | 1080 | 1072 | 1.29 |
| | FSM | **85.09%** | 1.5E-02 | 82.46% | 2.0E-02 | 556 | 1.00 | 1080 | 15.92 | 1080 | 622 | 17.88 |
| | Branch | 91.68% | 7.9E-04 | **92.01%** | 4.9E-04 | 980 | 0.00 | 611 | 174.83 | 1080 | 1007 | 136.45 |
| | Cond. | 84.03% | 2.8E-03 | **90.45%** | 2.1E-03 | 1077 | 0.00 | 351 | 8.07 | 1080 | 1079 | 0.00 |
| | Line | 96.56% | 4.4E-04 | **96.71%** | 2.5E-04 | 744 | 0.00 | 418 | 111.65 | 1080 | 925 | 1.92 |
| **Mean** | Code | 90.74% | 2.50E-03 | **91.48%** | 1.90E-03 | 2378.67 | 114.89 | 1056.33 | 74.07 | 3600.00 | 2580.50 | 151.59 |
| | Func. | 90.67% | 1.53E-03 | **93.41%** | 3.39E-05 | 1789.00 | 164.93 | 936.83 | 156.99 | 3600.00 | 1760.50 | 170.08 |

Table 2. Coverage saturation convergence across DUTs. The table shows across the experiment repetitions how many times CRV and Ranked have been able to reach the maximum code and functional coverage saturations for different budgets of simulations.

| | | **Code** | | | | | | **Functional** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DUTs** | **Methods** | 30% | 50% | 70% | 90% | 99% | 100% | 30% | 50% | 70% | 90% | 99% | 100% |
| CACHE | CRV | 0 | 1 | 13 | 40 | 50 | 50 | 0 | 13 | 35 | 51 | 58 | 60 |
| | Ranked | 100 | 100 | 100 | 100 | 100 | 100 | 49 | 100 | 100 | 100 | 100 | 100 |
| DPS | CRV | 2 | 8 | 26 | 41 | 48 | 50 | 0 | 0 | 2 | 27 | 49 | 50 |
| | Ranked | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| IBEX | CRV | 13 | 100 | 100 | 100 | 100 | 100 | 0 | 0 | 0 | 14 | 46 | 52 |
| | Ranked | 0 | 3 | 14 | 35 | 50 | 51 | 0 | 100 | 100 | 100 | 100 | 100 |

## *C. Results*

*Performance*. Once the optimal hyperparameter values across the DUTs are identified, we have compared the results of our RMP flow with respect to the traditional CRV one. We have evaluated the performance improvement in terms of required simulations and final coverage achieved by CRV and the RPM given two different budgets: the budget required by the CRV flow to reach coverage saturation (unknown to DV engineer until the end of a regression), and the budget required by RPM to reach maximum saturation. In case of CRV this is unknown and therefore capped at the number of required simulations to achieve the maximum budget. This last scenario showcases the net advantage in using our approach over CRV. Table 1 presents the result of our analysis with respect to the different coverage metrics, including both code and functional coverage results, across the three DUTs and the aggregated improvement. On average our methodologies achieve 0.26% higher code coverage, and 1.18% functional coverage compared to traditional CRV. This is higher improvement with respect to the one achieved by Huang et al. [8]. Moreover, our approach achieves on average the same coverage as CRV with 64% less tests for code coverage and 42% less tests for the functional coverage.

Table 3. Comparison table among constrained random running 100 parallel simulations ($CRV_{100}$) and sequential baseline (Seq) w.r.t. $RPMS_{10}$, $RPMS_{60}$, $RPMS_{80}$, and $RMPS_{100}$ where the subscripts indicate the number of required parallel simulation/CPUs/licenses. The table shows for the three DUTs: total simulation time (hh:mm), simulation time required to reach CRV saturation (hh:mm) and acceleration with respect to CRV.

| | **IBEX** | | | **DPS** | | | **CACHE** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Time (hh:mm)** | **Time (hh:mm)** | **Cov.** | **Time (hh:mm)** | **Time (hh:mm)** | **Cov.** | **Time (hh:mm)** | **Time (hh:mm)** | **Cov.** |
| | | **Cov. w.r.t. CRV** | **Accel. % (Time)** | | **Cov. w.r.t. CRV** | **Accel. % (Time)** | | **Cov. w.r.t. CRV** | **Accel. % (Time)** |
| CRV | 00:46 | 00:46 | 0.00% | 10:26 | 10:26 | 0.00% | 04:36 | 04:36 | 0.00% |
| Seq | 20:10 | 12:56 | -4673.33% | 05:53 | 11:13 | -3232.16% | 14:40 | 02:20 | -3595.85% |
| $RPMS_{10}$ | 04:58 | 03:10 | -307.06% | 22:06 | 09:03 | -216.10% | 04:56 | 23:40 | -568.42% |
| $RPMS_{60}$ | 01:25 | 00:45 | 3.40% | 16:46 | 05:36 | 46.22% | 17:56 | 13:35 | 52.58% |
| $RPMS_{80}$ | 01:25 | 00:38 | 17.38% | 07:11 | 04:18 | 58.84% | 12:06 | 10:58 | 61.73% |
| $RPMS_{100}$ | 01:25 | 00:31 | 31.56% | 19:31 | 03:41 | 64.60% | 09:36 | 09:56 | 65.30% |

*Convergence Rate.* In addition to final coverage saturation and number of required simulations, we have also evaluated the effectiveness of the methodology in converging towards the coverage saturation. This analysis allows to evaluate how likely the adopted methodology (CRV vs Ranked) reaches saturation over an increasing budget of simulations. Table 2 provide the convergence rate, across the three DUTs for 30%, 50%, 70%, 90%, 99%, and 100% simulation budget. From the results we can observe that the proposed methodology can converge to maximum functional coverage saturation in 100% of the experiments with less than 50% of the required simulations for all the DUTs. Similar results can be observed for the Cache and DPS over code coverage saturation, while in case of the IBEX, the CRV converge faster to coverage saturation. However, from Table 1 it is possible to observe that at the same time our methodology is able to achieve higher code coverage metrics with less simulations, with the exception of the FSM coverage metric, with respect to the CRV.

### *D. Regression Scheduler*

So far, we have showcased the effectiveness of the proposed RPM in reducing the overall number of simulations required and in improving the final coverage. However, while compared to industrial-grade DV flow it is important to evaluate the impact of the high degree of parallelization available to perform many parallel simulations. Table 3 shows the results of our RPM combined with the optimized regression scheduler (RPMS), with different level of parallelization, with respect to a CRV flow parallelizing simulations across 100 different CPUs (assuming the same number of licenses is available) and a traditional ML flow where a model inference is performed sequentially at the end of each RTL simulation. From the table we can observe how a naïve deployment of a sequential ML flow would result in a dramatic slowdown. While leveraging the proposed optimal scheduler, allows to significantly reduce the time required to reach CRV coverage saturation and thus dedicate the remaining time and resources to achieve better results. In particular, when using 60 parallel jobs our proposed RPMS flow achieves, across the three DUTs 0.74% higher code coverage and 2.74% functional coverage with 32% time saving with respect to CRV saturation time and 40% less resources required. Alternatively, while using the same number of resources (100), the simulation time savings increase up to 53.82% across the Cache, DPS and IBEX designs.

## V. CONCLUSION & FUTURE WORK

Our proposed RPM and optimized scheduling flow can identify the most effective tests reaching uncovered targets in a DUT. Our RPMS achieves 0.74% higher code coverage and 2.74% functional coverage while saving up to 53% of the overall verification time while compared to CRV, or 40% license saving for a 32% faster simulation time.

Our next step is to identify and create new tests by evaluating the contribution of multiple *plusargs* and combining them using generative approaches to further improve coverage. By leveraging generative models to create new test *flavors* we can enable exploration of design regions that the human-generated baseline is not capable of reaching. Currently, our methodology can accelerate the coverage closure according to the testbench capability limited by the existing *plusarg* and *flavor* space. The generation of new *flavors* will entail additional challenges arising from the possible generation of invalid *plusargs* combination, and therefore the need to dynamically learn valid and invalid tests, with the intent of avoiding scheduling tests more likely to fail in regression batches.

In addition to creating new tests, we aim to target different types of designs. The current flow is devised to address coverage closure and help verification of RTL designs. Nonetheless, the proposed approach can potentially be used to

target gate-level designs, and analog ones when targeting functional coverage metrics. The applicability of the methodology to code coverage metrics may be limited and meaningful mostly for RTL designs. While targeting gate level, and analog designs, additional challenges related to the dimensionality of the coverage data may emerge. We believe that the choice of lightweight models such RF or LGMB will not pose a significant challenge for the deployment of such models over larger observation spaces. Nonetheless, larger spaces can be tackled by decomposing the problem into smaller ones leveraging hierarchical information of cover bins, cover items, and cover groups as done by Debarshi et al. [11] while exploring uncharted functional coverage landscape.

Despite all these remaining challenges, we believe this is a first step for scalable AI-driven methodologies in the direction of automating the identification and creation of directed testing in industrial-level DV flows and first silicon success thereof.

## REFERENCES


[1] H. Witharana, Y. Lyu, and P. Mishra, “Directed test generation for activation of security assertions in rtl models,” ACM TOADES, 2021.

[2] Y. Lyu and P. Mishra, “Automated test generation for activation of assertions in RTL models,” in ASP-DAC, 2020.

[3] Y. Lyu, X. Qin, M. Chen, and P. Mishra, “Directed test generation for validation of cache coherence protocols,” IEEE TCAD, 2019.

[4] S. Fine and A. Ziv, “Coverage directed test generation for functional verification using bayesian networks,” in DAC, 2003.

[5] M. Braun, S. Fine, and A. Ziv, “Enhancing the efficiency of bayesian network based coverage directed test generation,” in Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop, 2004.

[6] M. imková and Z. Kotásek, “Automation and optimization of coverage-driven verification,” in Euromicro Conference on DSD, 2015.

[7] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, “Accelerating coverage directed test generation for functional verification: A neural network-based framework,” in Proceedings of the on Great Lakes Symposium on VLSI, 2018.

[8] Huang, Qijing, et al. "Test parameter tuning with blackbox optimization: A simple yet effective way to improve coverage.". DVCon US 2022.

[9] V Jayasree, et al. " Machine Learning for Coverage Analysis in Design Verification" Proceedings of the DVCon US 2021.

[10] Schiavone, Pasquale Davide, et al. “Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications.” 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS 2017)

[11] Debarshi Chatterjee, Spandan Kachhadia, Chen Luo, Kumar Kushal, Siddhanth Dhodhi. “GraphCov: RTL Graph Based Test Biasing for Exploring Uncharted Coverage Landscape.”. DVCon US 2023.