Optimization of Constrained Random Verification using Machine Learning

A Thesis
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY

Sarath Mohan Ambalakkat

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Prof. Gerald Sobelman

May 2018

ACKNOWLEDGEMENT

Foremost, I would like to express my deepest gratitude to my advisor, Prof. Gerald Sobelman for his continuous support, enthusiasm, and immense knowledge. I am deeply indebted to his passionate encouragement. Without his guidance this thesis research would not have been possible. It was an honor and a pleasure working with him.

I submit my heartiest gratitude to Mr. Eldon Nelson for allowing me to extend his work. I would also like to thank Prof. Sachin Sapatnekar and Prof. Rui Kuang, for willing to be a part of my thesis committee and review panel and providing insightful comments on my thesis. Last, but not the least, my joys know no bound in expressing my cordial gratitude to my parents and friends, for their continuous moral support and encouragement.

DEDICATION

This thesis is dedicated to my family and friends.

ABSTRACT

Constrained random simulations play a critical role in Design Verification today. But the effort and time spent to manually update the input constraints, analyzing and prioritizing the unverified features in the design, significantly affect the time taken to converge to the coverage goal. This research work focuses on the optimization of constrained random verification using Machine Learning algorithms, in a coverage-driven simulation using a Universal Verification Methodology (UVM) framework. The optimization will greatly reduce the time a simulation takes to converge to the coverage goal. This research work targets automating the update of the constraints during runtime, abstracting the need for understanding the design to verify it, using Machine Learning. The verification environment is further optimized using techniques including Objective Function, Rewinding and Dynamic Seed Manipulation. The enhanced environment resolves the limitations of the previous efforts at employing these techniques, optimizing the scalability of the environment and enhancing its compatibility at verifying complex combinational designs and sequential designs including Finite State Machines (FSMs).

The optimized verification environment comprises of a SystemVerilog testbench which interfaces and interacts with a TCL environment. The methodology has been empirically demonstrated, with remarkable results showing its superior quality in terms of faster automated coverage closure, efficient final stimulus solution and proposed higher quality of coverage. Multiple Machine Learning algorithms, including a Linear Regression Model and Artificial Neural Networks, have been employed to scale the compatibility of the verification environment, making it capable of autonomously verifying designs of varied behavior. Adequate simulation results to demonstrate the same have been presented in the report.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Overview

Design Verification is becoming increasingly challenging and time-consuming day by day. The transistor sizes are shrinking at a remarkable rate, and hence, there is an exponential increase in the total number of transistors in a chip over the years. This translates to more functionality in the same die area, and in turn to increased design complexity. With such an increase in complexity of designs, the time taken to verify the same is also increasing drastically. In this fast-evolving world, time-consuming Design Verification is becoming a necessary evil, greatly affecting the speed of evolution. The traditional methodology of using directed tests targeting the verification of specific features in the design has become unrealistically time consuming in case of such increasingly complex designs. Constrained random verification hence has become an immediate necessity. But, finding the right combination of constraints to produce the most stressful tests with the widest variety of random stimulus is again a challenge.

Machine Learning has numerous applications and has presented remarkable performance optimizations in various domains [11]. The research work presented in this thesis targets automating the update of constraints, abstracting the need for understanding the design, in coverage-driven constrained random simulations, using Machine Learning algorithms. The main objective is to converge to the coverage goal faster. Functional coverage provides essential feedback for knowing what was tested, the device configuration used and, perhaps most importantly, what still has not been tested. It is this design-independent indispensable information that we are going to exploit to update our constraints autonomously. The optimized verification environment is further enhanced using techniques including Objective Function, Rewinding and Dynamic Seed Manipulation. The SystemVerilog testbench with UVM framework has been configured with multiple machine learning algorithms, including a Linear Regression Model and

Artificial Neural Networks (ANN), to scale the environment, making it capable of autonomously verifying complex designs with varied behavior. Simulation results testing the same using the fully functional testbench is presented in this thesis.

1.2 Design Verification

In today's world, most electronic devices are architected using an SoC design paradigm, integrating predefined hardware and software blocks. As the complexity of designs on an SoC increases, the challenge of verifying the design completely becomes harder [16]. This is the reason for the marked increase in use of Constrained Random Verification as opposed to the traditional testbench approach using Directed Tests. The completeness of the design verification may be defined using multiple metrics; of which, the primary and most extensively used metric is "Coverage." This result-oriented approach to functional verification is hence termed coverage-driven verification. Coverage can be broadly divided into Code Coverage and Functional Coverage.

1.2.1 Code Coverage

Traditionally, the quality of verification was measured using code coverage. Code coverage specifies the extent to which the Design HDL has been exercised. The tools measuring code coverage base their reports on and categorize the same into line coverage, block coverage, conditional coverage, branch coverage, toggle coverage and finite state machine coverage.

However, a 100% code coverage does not necessarily mean that the design has been fully verified. Code coverage does not fully define the completeness and quality of the verification. It has multiple limitations, which includes:

- Inability to identify non-implemented features.
- Inability to measure sequential multi-cycle events.
- Inability to measure the interaction between multiple modules.

1.2.2 Functional Coverage

Unlike code coverage, functional coverage is a user-defined metric that defines the measure of the design functionality that has been exercised by the verification environment. With proper brainstorming, a functional coverage model covering all the features specified in the design specification may be defined. The model may then be used to track whether the important values, sets of values and sequences of values in the design, and interface features and boundary conditions, have been exercised, thereby overcoming the limitations of code coverage.

The coverage model is defined using the SystemVerilog covergroup construct. A covergroup may consist of multiple coverpoints synchronously sampled, either using a clocking event or using an in-built sample() method, as presented below.

```
// Covergroup Definition (SAMPLED USING BUILT-IN "sample()")
covergroup cov_grp;
  cov_p1: coverpoint a;
endgroup

 //Covergroup Instantiation
cov_grp cov_inst = new();

//Covergroup Sampling on "trigger"
@(trigger) cov_inst.sample();

// Covergroup Definition (SAMPLED USING CLOCKING EVENT)
covergroup cov_grp @(posedge clk);
  cov_p: coverpoint in;
endgroup

//Covergroup Instantiation
cov_grp cov_inst = new();
```

The coverpoints defined may be variables or expressions. We may also define cross-coverages between coverpoints. Each coverpoint is associated with bins, which may either be auto-generated, or user-defined. The bins defined may specify the extent to which the state space is expected to be explored or the transitions expected on the coverpoints, or the ignored and illegal values, as presented in the example given below. Bins that have not been hit at least once after the simulation is complete reveal the coverage holes.

```
covergroup cg @(posedge clk);
  c1: coverpoint wr_addr {
              bins b1          = {0,2,7};       // User-defined Bins
              bins b2          = (10=>20=>30); // Transition Bins
              illegal_bins b2 = {41:50};       // Illegal Bins
              ignore_bins  b3 = {[31:40],[51:60]};  // Ignore Bins
            }
  c2: coverpoint rd_addr;  // Auto-Generated Bins
endgroup : cg
```

With the ever-increasing complexity of designs, and therefore the exponential increase in the number of directed tests required for verification, converging to the 100% coverage goal using the traditional approach, even if possible, is unrealistically time-consuming. The increasing need for a faster coverage closure, tackling the exploding state space of modern designs, demands the need for constrained random verification using SystemVerilog testbenches.

1.3 Constrained Random Verification

Up to 60% of the overall design cycle time is estimated to be spent on design verification. Constrained random verification [1] was introduced to enhance the time taken for coverage closure. SystemVerilog testbenches employing constrained random simulations achieve this by addressing the two main problems of the traditional testbench approach, which are procedural, as opposed to

declarative, and enumerative, as opposed to comprehensive. Constraints defined using SystemVerilog constructs [12] are declarative and are closer to the design specification, as they mimic the latter. Moreover, the only reasonable solution to the error-prone and time-consuming enumerative approach of traditional directed testbenches are random simulations which generate unanticipated scenarios which have greater chances of detecting bugs in the design.

To summarize the essential steps in the coverage-driven verification process:

1. Define the coverage model, using covergroups and coverpoints, from the verification plan
2. Debug the verification environment, checkers, and coverage model
3. Perform multiple tests using random seeds until the cumulative coverage converges
4. Update stimulus constraints to target coverage holes and run further tests
5. Analyze and prioritize any unverified features and run directed tests for particularly hard-to-reach coverage holes

To plan and monitor the tested functionality, methodologies based on coverage, checks, and assertions such as the Metric-Driven Design Verification (MDV) [6] were introduced. Advanced methodologies like Universal Verification Methodology (UVM) provide in-built constructs to support constrained random verification [2, 3]. The basic flow of coverage-driven constrained random simulation has been summarized in the figure below.

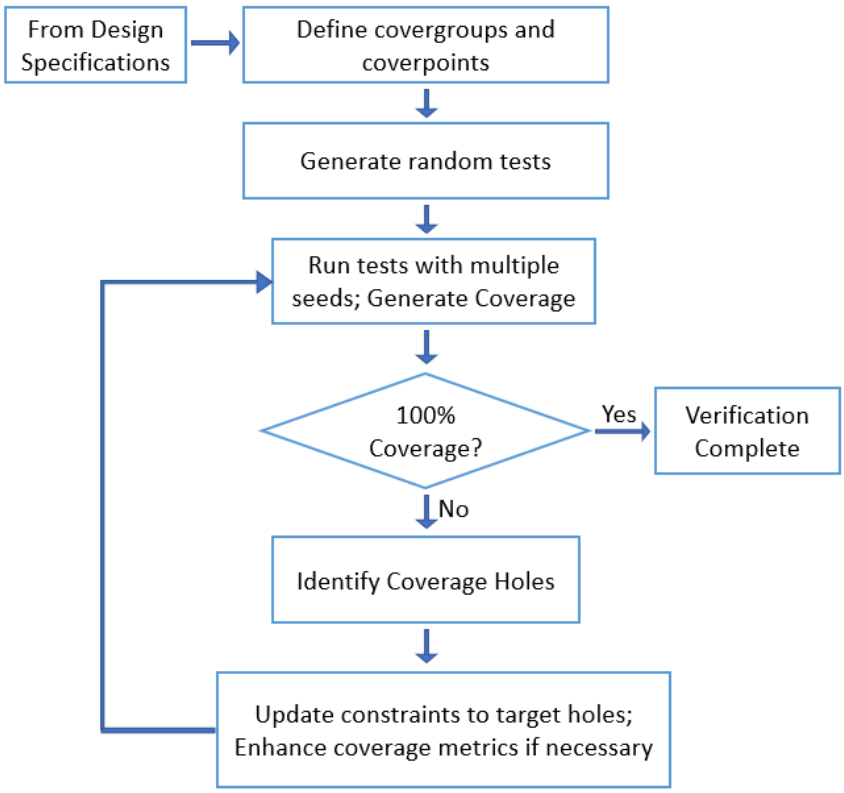

Figure 1: Constrained Random Simulation Flow Chart

1.4 Motivation

Design verification plays a vital role in the design flow in ensuring the correctness of the design. In many situations, the complete verification of a design takes a significant amount of time, perhaps even more time than the design of the module itself. The increasing complexity of designs drastically multiplies the effort necessary to verify the design and to achieve the 100% coverage goal.

Though constrained random simulations have marked advantages over the traditional directed test approach, several factors still limit its performance. The “classical” coverage-driven verification employs design-dependent manual update of constraints. This makes handling the environment challenging and error-prone. Hence, the optimization of the time taken to verify a design using

coverage-driven constrained random simulations is largely limited by the manual update of constraints, in order to target the coverage holes and hit the corner cases.

Furthermore, the manual update of constraints is largely based on the understanding of the design and extracting the design specification. This may adversely affect the efficiency of the stimulus generated, time taken for coverage closure, and quality of coverage. Abstracting the need for understanding the design in order to verify it requires the automation of constraint updates.

Recently, research has been carried out to automate the update of constraints in a coverage-driven simulation. For example, graph-based stimulus generation targets automating the enumeration of minimum number of tests needed to cover the paths through the state space and hence converge to the coverage goal much faster than the "classical" approach [10]. But such a graph-based stimulus generation will require a separate, dedicated tool (like Questa inFact from Mentor Graphics) before its results are fed into a Universal Verification Methodology (UVM) testbench, which limits the applicability of the framework. Similarly, in [14], the efforts at dynamically guiding random stimuli generators rely on an external API, Unified Coverage Interoperability Standard (UCIS), defining the C data structures and the API functions used to access the repository of coverage data.

UVM testbenches implementing constrained random verification are being widely used in practice. The main motivation behind the research described in the thesis is automating the update of constraints, eliminating the need to understand the design in order to verify it. Functional coverage provides essential feedback for knowing what was tested, the device configuration used and, perhaps most importantly, what still has not been tested. It is this design-independent indispensable information that we are going to exploit to update the constraints autonomously. This optimization greatly reduces the time consumed for coverage closure. The proposed method automates the update of the constraints employing Machine Learning algorithms.

1.5 Machine Learning

Machine learning may be defined as the science of getting computers to act without being explicitly programmed [5]. Machine learning is being widely exploited in various domains; for example, in web searches, self-driving cars, image/speech recognition, etc.

The machine learning algorithms model the behavior of a design and generalizes it based on its previous experience. The model is then used for making fairly accurate predictions about outputs of unseen input samples. However, the correctness of predictions made by a good model can depend on multiple factors, including quality of training sets, scalability of the model and most importantly, the machine learning algorithm used to model it.

Machine learning algorithms can be broadly divided into two categories [5]:

1. Supervised Learning Algorithms: Labeled training sets, which ideally represent the state space, train the algorithm to build a model of the state space and learn the behavior of the design. The model approximates the function that best fits the training sets and converges to parameters that best describes the design. The main types of supervised learning problems include regression and classification problems, pertaining to prediction of continuous values and discrete values, respectively.
2. Unsupervised Learning Algorithms: The algorithm is trained using unlabeled training data, i.e. there are no output categories or labels based on which the algorithm models relationships. The algorithm is used for pattern detection and descriptive modeling, i.e. the data points are explored to identify patterns and identical data points are categorized into sets, with each category representing a different set of characteristics.

In [13], the author has demonstrated the applicability of Unsupervised Machine Learning algorithms at optimizing regression tests using the toggle pair coverage as the metric. For our

application we will be using Supervised Learning algorithms, providing necessary labeled training sets to train the Machine Learning model in order to make accurate predictions. We will be using functional coverage as the metric for measuring the performance. The scalability of the model will largely depend on the algorithm employed. The verification environment has been configured with multiple machine learning algorithms including:

1. Linear Regression Model: We may a use a Linear Regression Model to represent the relationships if the inputs and outputs of a design have a linear dependency such as:

   $$X_0*\beta_0 + X_1* \beta_1 = Y$$

   where X_0 and X_1 are the inputs, β_0 and β_1 are the parameters defining the Linear Regression Model, and Y is the output.

   In the given example, the parameters β_0 and β_1, can be estimated using two labeled training sets of the format [X_0, X_1, Y]. Once the parameters have been estimated, the model can be used for predicting the output, Y, for unseen sets of inputs [X_0, X_1]. The model is simple and requires a minimal number of training sets, to make accurate predictions. However, a Linear Regression Model has the limitation that the inputs and outputs must have a linear relation, limiting the scalability of the verification environment.

2. Artificial Neural Networks (ANN): Neural networks are one of the most efficient machine learning algorithms, capable of modeling complex relations between the inputs and outputs [15]. As the name suggests, Artificial Neural Networks are inspired from brain neurons. The general structure of an ANN has been shown in figure below. A neural network primarily consists of three types of layers:

   a. Input Layer: Inputs are fed to this layer.
   b. Hidden Layers: Intermediate layers that are used for defining the complex relations between the inputs and the outputs.
   c. Output Layer: Final output is extracted from these layers.

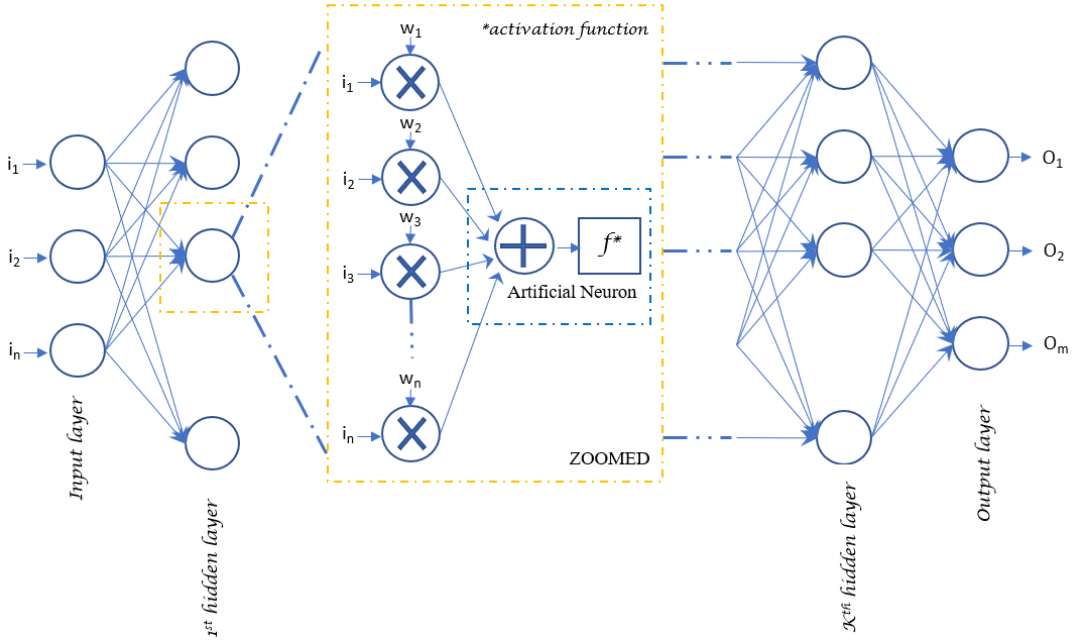

Figure 2: Artificial Neural Network Structure

Similar to the human nervous system, wherein a single neuron takes in inputs from multiple dendrites, every neuron in an ANN collates numerous inputs, i_1, i_2, … i_n multiplied with the weights, w_1, w_2, … w_n and performs a function, called the activation function, on the sum. The output from the neuron is sent to all the neurons in the next layer to which it is connected. The output of each neuron may be summarized as:

$$y = f(\sum_{i=0}^{n} Wi * Ii)$$

Neural Networks are trained using an algorithm termed Back-Propagation [15]. On feeding a training data, the ANN computes the actual outputs, using the weights and the activation functions defining each neuron in the network. The actual outputs are then compared to the expected outputs. The error at the output is propagated back through the network. The weights in the network are adjusted to minimize the error.

Machine Learning has exhibited promising applications in varied domains. In this research, we are trying to exploit its capability to learn a design without explicit programming, to make accurate predictions of outputs for unseen inputs. Specifically, the objective is to predict constraints that can generate unseen outputs targeting previously missed bins.

1.6 Contribution of the Thesis

This research optimizes constrained random simulations by automating the update of constraints eliminating the need of understanding the design to verify it. The verification environment employs machine learning algorithms to analyze and prioritize any unverified features autonomously and to make accurate predictions of input combinations that can generate unseen outputs to exercise these unverified features. This facilitates generating appropriate input stimulus targeting particularly hard-to-reach coverage holes. The optimization overcomes the effort and time taken to manually update constraints and converge to the coverage goal.

The verification environment has been equipped with multiple machine learning algorithms, including a Linear Regression Model and Artificial Neural Networks, to model the predictor. This enhances the scalability of the environment, enhancing its capability at verifying varied designs with different behaviors.

The training sets, used for training the machine learning algorithms, are implicitly generated by the verification environment during the simulation, removing the need for any design-dependent information from the verification engineer.

Functional coverage provides information on the set of values of a variable that has been exercised by the verification environment, and more importantly the values yet to be reached in a constrained random simulation. Machine learning algorithms exploit this design-independent information to make accurate predictions of input combinations used for autonomously updating the constraints

during runtime. The runtime update of constraints has been implemented using efficient techniques that eliminates the need for re-compilation of the simulation environment after its update. Hence, ideally the simulation should autonomously converge to the 100% coverage goal within a single simulation run.

The simulation environment also sustains the capability of random simulations to uncover unanticipated bugs in the design. To enable this, the simulation runs a configurable number of unconstrained random iterations at the beginning.

The research further optimizes the verification environment employing techniques including Objective Function, Rewinding, and Dynamic Seed-Manipulation. The enhanced environment overcomes the limitations of the prior work [4] implementing the techniques, which was incapable of handling complex designs. It was also incapable of verifying sequential designs involving Finite State Machines. This research resolves all these drawbacks in the previous work.

Additionally, the techniques are compatible with advanced methodologies like the Universal Verification Methodology (UVM), widely used in industries these days. Simulation results demonstrating its superior quality, in terms of faster automated coverage closure, efficient final stimulus solution and higher quality of coverage, are presented in this thesis.

2. PRIOR WORK

The basic method of Constrained Random Verification in a SystemVerilog testbench, uses a single seed that is used in the randomization process, which is passed down the entire simulation in a deterministic way. The coverage-driven environment will normally take a long time to converge to the 100% coverage goal in such a scenario. In [4], the author suggests a novel method to optimize the same, through Objective Functions, Rewinding and Dynamic Seed Manipulation. The author shows that the optimization will result in faster automated coverage closure, an efficient final stimulus solution and a higher quality of coverage. It targets automating the simulations and making the verification environment design-independent.

The main ideas used in the method include Objective Function, Checkpointing, Rewinding and Dynamic Seed Manipulation. An objective function may be defined as any function that increases continuously over time as we approach our goal. In a coverage-driven simulation, we may consider coverage, which is a continuously increasing function, to be the objective function. Checkpointing is a vendor-specific but universal capability of saving a simulation point in time with all states intact. The main ideas proposed in the prior work are the following:

1. Exploiting checkpointing to save checkpoints during runtime.
2. Run the simulation for a defined “Interval” from the saved checkpoint.
3. Check for an increase in coverage over the “Interval.”
4. If the objective function has not increased over the defined “Interval,” we rewind the simulation to the last checkpoint and dynamically re-seed the simulation.
5. Repeat steps 2, 3 and 4 until an improvement is observed in the coverage. If the seed resulted in an improved coverage, we update the checkpoint and proceed with the simulation to run the next “Interval” using the latest seed.

Throughout this thesis, the simulation run for the defined “Interval” is referred to as an iteration and the performance of a verification environment is going to be measured in terms of the number of iterations it takes to converge to the coverage goal.

The simulator uses TCL commands to pause the simulation after the “Interval” and invokes TCL functions to evaluate the interval by checking if the objective function, i.e. the coverage, has improved over the “Interval.” If not, TCL restores the last checkpoint, reseeds the simulation and re-runs the simulation with the new seed. Given below is a snippet of a log file for a rejected seed for a scenario where no improvement was observed on the objective function. Highlighted in red is the seed rejected and rewinding to previous checkpoint on observing no progress in coverage from 25% over the interval.

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 37 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 2 at time: 27 ns
INFO STATUS : TCL : 37 ns : NO PROGRESS : false: 25.000000 > 25.000000
REWINDING TO CHECKPOINT {2} at 27 ns
All the Checkpoints created after checkpoint 2 are removed...
------------------------- END eval_loop
```

When an improvement is seen in the objective function, the seed is accepted, the checkpoint is updated and the simulation proceeds to run for the next “Interval.” The loop continues until the objective function is satisfied, i.e. the simulation converges to the 100% coverage. Given below is the snippet of a log file for an accepted seed. Highlighted in green is the seed accepted, on observing an increase in coverage from 0% to 25% over the interval.

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 27 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 2 at time: 17 ns
INFO STATUS : TCL : 27 ns : GOOD : 25.000000 > 0.000000
------------------------- END eval_loop
```

The advantage of the method is the fact that the simulation environment is independent of the design being verified. Irrespective of what the design is, the simulation environment is autonomous and approaches the coverage goal over time. The author of the prior work claims that, irrespective of what the design is, the simulation environment will converge to the 100% coverage over time.

3. DRAWBACKS OF THE PRIOR WORK

Verification using the simulation environment defined in the prior work can get stuck in a simulation loop, in which case the coverage function does not converge to the coverage goal. In such scenarios, the conventional UVM timeout mechanisms will no longer be effective at terminating a simulation, since it is based on simulation time and not the clock time. Therefore, in scenarios where we fail to achieve an improvement on the objective function, the simulation repeats the same simulation period and gets stuck in an infinite loop.

The techniques mentioned in the prior work [4] look promising and work very well for small combinational designs. However, the scalability of the environment is most definitely questionable. This is because the framework of the testbench does not optimize the time taken to converge to the coverage goal. Though the simulation time is optimized using the techniques of objective function, rewinding and dynamic seed-manipulation, the clock time taken to converge to the coverage goal is similar to that of an unconstrained random simulation. As the complexity of the design being verified increases, the size of the state space defining it increases dramatically. Hence, the verification of the design using the unconstrained environment is unrealistically time-consuming.

Rigorous testing was done to check the scalability of the environment, and the environment was experimentally shown to be inefficient to verify complex designs. The observations made are as follows:

1. Verification of a comparator with reconfigurable input widths: The block diagram of the design being verified is given in the figure below:

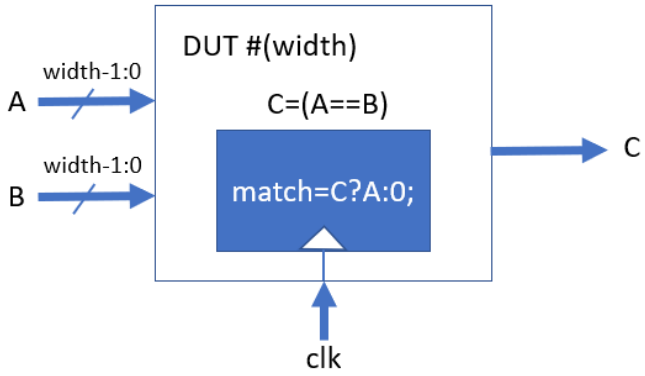

Figure 3: DUT used for testing the original environment (after [4])

The Design Under Test (DUT) is a 2-input parameterized comparator. When the inputs `A` and `B` are equal, the output `C` is asserted. The internal signal `match` is assigned with `A` when `C` is asserted at a positive edge of the clock. Otherwise, it is assigned to be 0. A covergroup has been defined inside the DUT, with the internal signal `match`, as the coverpoint, in order to check if all possible matching values have been generated on `A` and `B` at the positive edge of the clock. The parameterized design is used to check the compatibility of the verification environment for varying input widths. Observations about the simulation results are as follows:

a. When the parameter `width` was set to 1, i.e. for a 1-bit comparator, the simulation results were promising. The simulation converged to 100% coverage in 3 iterations. A snippet of the log file is shown below with the total number of iterations and the coverage goal highlighted:

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 47 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 106767085 at time: 27 ns
INFO STATUS : TCL : 47 ns : GOOD : 100.000000 > 50.000000
INFO STATUS : TCL : MET OBJECTIVE!
------------------------- END eval_loop
UVM_INFO sv/rseed_interface.sv(122) @ 57: reporter [RS]
COVERAGE GOAL MET coverage:  100    max_objective:  100
```

```
INFO STATUS : TCL : ITERATIONS TOTAL = 3
INFO STATUS : TCL : final_report END
```

b. On increasing the width of the comparator, the number of iterations to converge to 100% coverage increased dramatically. When the `width` was set to 3, i.e. for a 3-bit comparator, the simulation converged to 100% coverage in 261 iterations. A snippet of the log file has been presented below with the total number of iterations and coverage goal highlighted:

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 107 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 41496968 at time: 87 ns
INFO STATUS : TCL : 107 ns : GOOD : 100.000000 > 87.500000
INFO STATUS : TCL : MET OBJECTIVE!
------------------------- END eval_loop
UVM_INFO sv/rseed_interface.sv(122) @ 117: reporter [RS]
COVERAGE GOAL MET coverage: 100 max_objective: 100
INFO STATUS : TCL : ITERATIONS TOTAL = 261
INFO STATUS : TCL : final_report END
```

c. For widths greater than 3, a 64-bit mode VCS simulation was so time-consuming that it terminated giving a segmentation fault before converging to the coverage goal. A snippet of the log file with the error highlighted is given below:

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 167 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 110487736 at time: 157 ns
INFO STATUS : TCL : 167 ns : NO PROGRESS : false: 87.500000 >
87.500000 REWINDING TO CHECKPOINT {2} at 157 ns
All the Checkpoints created after checkpoint 2 are removed...
/usr/local/apps/synopsys/vcs/amd64/../gui/dve/bin/dve: line
159:  1264 Segmentation fault      (core dumped) $CMD "$@"
make: *** [sim_synopsys] Error 139
```

A summary of the simulation results for comparators of different widths using the simulation environment in [4] is shown in the table given below:

Table 1: Simulation Results (Original Environment)

#	Width of Comparator	No of Iterations
1	1	3
2	2	60
3	3	261
4	4 or more	Segmentation Fault

These results show that the verification environment is incapable of verifying complex designs. As the width of the comparator increases, the number of iterations taken to converge to the coverage goal is increasing rapidly. Even for a design as simple as a 4-bit comparator, the simulation terminated giving a segmentation fault. The reason for this large increase in the time taken to converge to the coverage goal is that the simulation environment is similar to an unconstrained random simulation, in terms of the state space and the time taken to explore the state space.

2. Verification of Sequential Designs: Additional difficulties were discovered when attempting to apply the prior method to sequential circuits. Consider, for example, a design with a binary tree-structured Finite State Machine (FSM) with a root state and numerous leaf states as shown in the figure below:

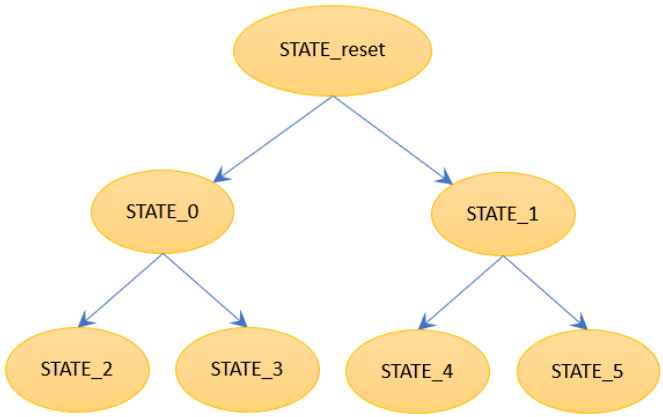

Figure 4: Binary tree structured FSM

In such a scenario, it is obvious that once the simulation reaches one of the leaf states, the simulation will not proceed, since no combination of input will result in an increase in the objective function, irrespective of the input seed on which the simulation interval is run.

In certain cases, there may be scenarios where only very few combinations of inputs can result in a particular state-transition. In such cases as well, the probability of the simulation converging to 100% coverage is very small. Consider the state machine diagram presented in the figure below, where `A` and `B` are 3-bit inputs to the FSM.

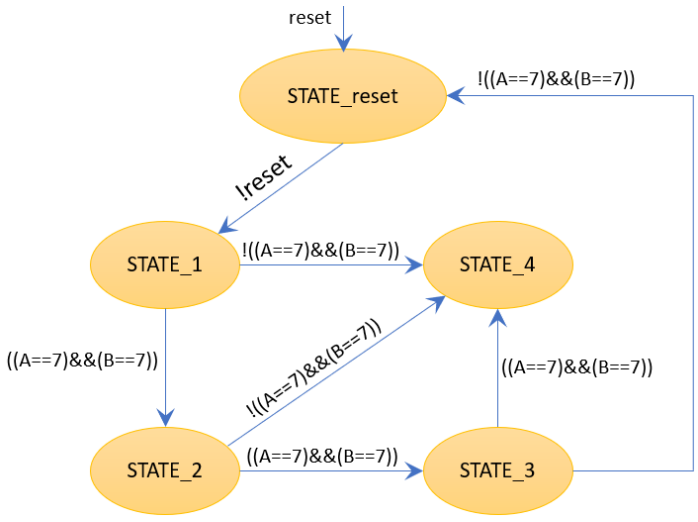

Figure 5: FSM DUT used for testing the original environment

After reset de-assertion, only when `A` and `B` are both 7 will the state machine go from STATE_1 to STATE_2, i.e. 63 out of 64 times the State Machine goes from STATE_1 to STATE_4. A similar situation occurs for the other labeled transitions. Therefore, the probability of the sequence STATE_1 -> STATE_2 -> STATE_3-> STATE_4 to occur is 1 out of (64*64*64); which is negligibly small. Therefore, the odds are that the simulation will not converge to a 100% coverage in most cases.

These observations were experimentally demonstrated through simulations. The test design was simulated using a SystemVerilog testbench employing the optimizations in [4], i.e. Objective Function, Rewinding and Dynamic Seed Manipulation. The most probable transition after the de-assertion of the reset would be from STATE_reset->STATE_1->STATE_4. If the covergroup is defined with the `CurrentState` as the coverpoint (shown below), in such a scenario the simulation would have only covered 3 out of 5 possible states, i.e. the simulation will be stuck at 60% coverage.

```
reg [2:0] CurrentState;
covergroup objective_cg;
  coverpoint CurrentState {
      bins state_initial = {STATE_Initial};
      bins state1 = {STATE_1};
      bins state2 = {STATE_2};
      bins state3 = {STATE_3};
      bins state4 = {STATE_4};
      ignore_bins states567 = {STATE_5, STATE_6, STATE_7};
}
endgroup
```

A snippet of the simulation log is shown below. As expected the simulation does not converge to 100% coverage. The simulation will be stuck at 60% coverage in STATE_4 and terminate giving a segmentation fault after some time (highlighted).

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 37 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 23 at time: 27 ns
INFO STATUS : TCL : 37 ns : NO PROGRESS : false: 60.000000 > 60.000000
REWINDING TO CHECKPOINT {2} at 27 ns
All the Checkpoints created after checkpoint 2 are removed...
------------------------- END eval_loop
UVM_INFO sv/dut.sv(15)@30: reporter [dut_if] AFTER drive regs A: 7 B: 4
UVM_INFO sv/dut.sv(113)@30: reporter [dut] Current State: STATE_4
------------------------- START eval_loop
DEBUG current simulation time is ctime : 37 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 771 at time: 27 ns
INFO STATUS : TCL : 37 ns : NO PROGRESS : false: 60.000000 > 60.000000
REWINDING TO CHECKPOINT {2} at 27 ns
All the Checkpoints created after checkpoint 2 are removed...
/usr/local/apps/synopsys/vcs/amd64/../gui/dve/bin/dve: line 159:  29466
Segmentation fault      (core dumped) $CMD "$@"
make: *** [sim_synopsys] Error 139
```

The research undertaken in this thesis targets resolving all the above-mentioned drawbacks in the environment and developing an autonomous, design-independent, UVM-compatible technique to update the constraints during runtime without having to recompile the updated environment. The next chapter summarizes the structure of the test environment used in the research.

4. TEST ENVIRONMENT

The test environment is a simplified version of the conventional UVM framework. It extends the UVM base classes, `uvm_sequence_item` and `uvm_test`. The environment is set up to include only the necessary components required to facilitate the demonstration. No agents or environments have been used. Though simplified, the test environment employs constrained random verification and demonstrates the optimizations clearly. Given in the figure below is the structure of the SystemVerilog testbench framework used to demonstrate the research.

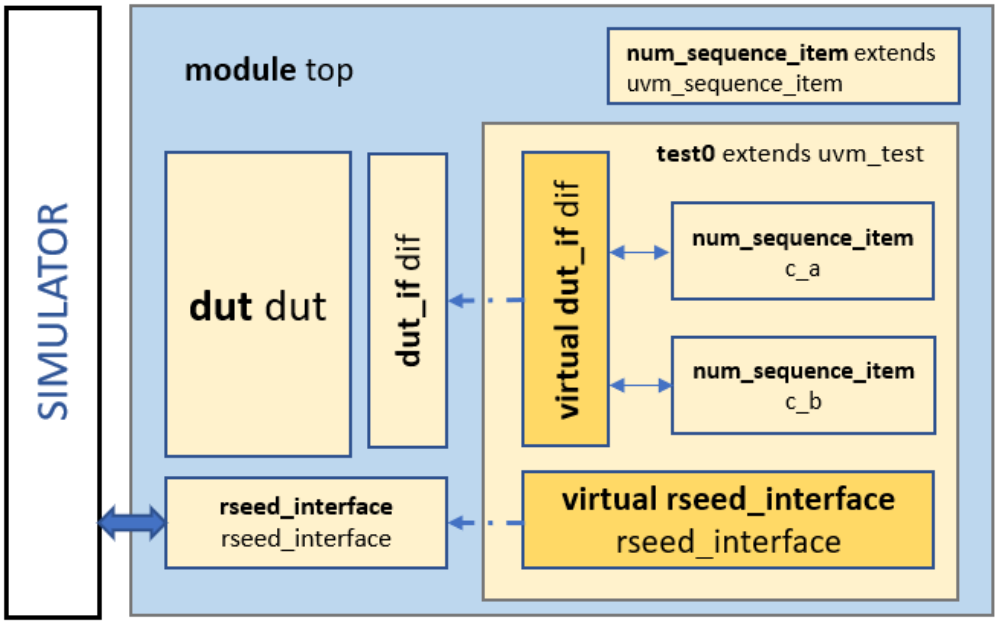

Figure 6: SystemVerilog testbench framework (after [4])

The `dut` is the design being verified; for example, the parameterized comparator discussed earlier, with inputs `A` and `B` of configurable widths. The static module, instantiated in the `top` module, interfaces and interacts with the dynamic testbench through the interface, `dut_if`. The testbench drives the inputs into the `dut` through the `dut_if`. The `test0`, extending `uvm_test`, instantiates the interface virtually. The top module sets the configuration database with the hierarchical path to connect to the interface, using the set command:

```
uvm_config_db#(virtual dut_if#(width))::set(null, "uvm_test_top",
"dif", dif);
```

The command sets the (name, value) pair corresponding to `dut_if`, in the configuration database, which may be accessed from the dynamic classes. Virtual connections may be made to the interface from `test0` using:

```
if (!uvm_config_db#(virtual dut_if)::get(this, "", "dif", dif))
begin
   `uvm_fatal("test0", "Failed to get dut_if")
end
```

`test0` converts the transactions `c_a` and `c_b`, objects of the handle `num_sequence_item` extending the `uvm_sequence_item`, to pin wiggles and drives the inputs of the `dut` through the virtual interface.

An interesting feature in the framework is the `rseed_interface` (i.e. the randomization seed interface). The simulator may use the static interface to trigger SystemVerilog functions that dynamically interrogate and modify the verification environment [4]. The interface contains knobs and functions that control the functionality of the testbench, which may be used to interface with the simulator. The basic structure of the `rseed_interface` which illustrates some of the variables and functions in it is shown below:

```
interface rseed_interface ( input clk, input reset );
   bit      trigger                = 0;
   time     start_time             = 7;
   time     interval_time          = 10;
   bit      final_report           = 0;
   real     coverage_value         = 0;
   int      max_objective          = 100;
   int      unsigned seed;
```

```
    // INCLUDE ALL VARIABLES HERE
    function void get_instance();
       ms.get_instance();
    endfunction
    // Set the Seed of the Singleton
    function void set_seed(int unsigned s);
       seed  = s;
       ms.set_seed(s);
    endfunction
    // Get the Current Coverage Value
    function real get_coverage_value();
       return ms.get_coverage_value();
    endfunction
    // INCLUDE ALL OTHER FUNCTIONS HERE
    initial begin // Initialize variables using in-line arguments
       $value$plusargs("start_time=%d", start_time);
       $value$plusargs("interval_time=%d", interval_time);
       $value$plusargs("ml_enabled=%d", ml_enabled);
       $value$plusargs("fsm_opt_enable=%d", fsm_opt_enable);
       // ADD REMAINING ARGUMENTS
       #(start_time);
       forever begin
          #(interval_time);
          // use variable instead of file to pass coverage value
          coverage_value  =  dut.objective.match.get_coverage();
          ms.set_coverage_value(coverage_value);
          if (reset == 0) // Trigger TCL function to evaluate seed
             trigger  = ~trigger;
          if (ms.get_coverage_value() >= ms.max_objective) begin
             final_report  = 1;
          end
      end
endinterface
```

The above is a sample of the variables and functions defined in the interface. Many of the key features integral to the implementation of the optimization techniques interface with the simulation environment through the `rseed_interface`. Not all SystemVerilog simulators can call functions from dynamic objects in the UVM framework. Hence, the static interface, `rseed_interface`, with the wrapper to all the functions in the classes, provides a static location which may be accessed from the simulator. The dynamic objects may connect to these static locations, using a virtual interface, providing a software communication layer to the simulator. The `set` and the `get` commands, similar to that used in case of `dut_if`, may be used to make virtual connections to the interface through the configuration database.

The simulator can invoke TCL commands to interface and interact with the SystemVerilog testbench. In general, we would like to get the value of a variable from a known location in the SystemVerilog testbench. For example, to get the value of `max_objective` from the SystemVerilog testbench, which defines the maximum value the objective function is expected to be converging to during the simulation, we may use the TCL command:

```
set max_objective [get top.rseed_interface.max_objective]
```

Similarly, we may also set internal signals in the SystemVerilog testbench using TCL commands. For example, to force an internal signal, say `beta_ready` in `rseed_interface` in the SystemVerilog testbench from the simulator, we may use:

```
force top.rseed_interface.beta_ready 1;
```

To call a function in the SystemVerilog testbench, say `set_seed()` passing input arguments from the simulator, we may use:

```
set last_seed [expr {int(rand()*4294967294+1)}]

call top.rseed_interface.set_seed(32'd${last_seed})
```

The above explained test environment can efficiently implement the optimization techniques including, Objective Function, Rewinding, and Dynamic Seed-Manipulation. A brief flow chart of the simulation environment employing these techniques is presented in the figure below:

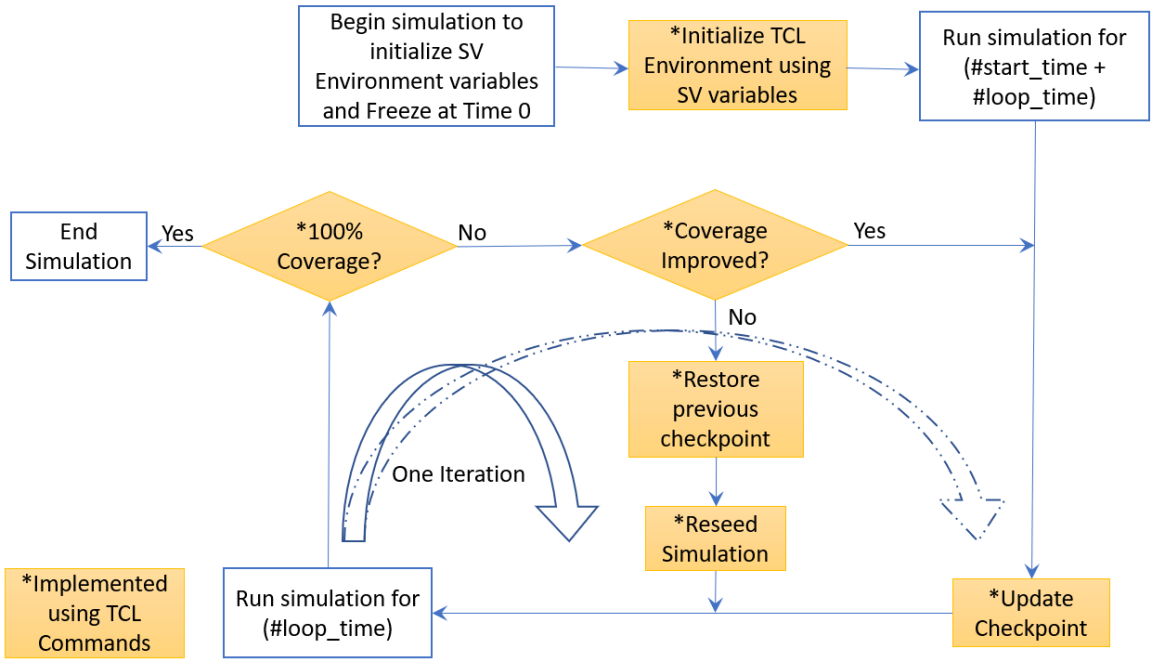

Figure 7: Simulation Flow employing TCL commands to implement Objective Function, Rewinding and Dynamic Seed-Manipulation (after [4])

The simulator initializes the TCL Environment with variables from the SystemVerilog testbench. Once the TCL Environment has been set up, the simulation starts and runs for a configurable duration of time which is equivalent to the `start_time` plus the `loop_time`. After updating the checkpoint in the simulator, the simulation runs for `loop_time`. Triggers from the `rseed_interface` invoke the TCL commands to run the function to evaluate the loop, in order

to check if the objective function has improved. The simulator can access the objective function, i.e. the coverage, by invoking the `get_coverage_value()` function in `rseed_interface`. If the coverage has not improved, we rewind the simulation by restoring the previous checkpoint. We re-seed the simulation and run the simulation loop with the new seed. We pass the random seed generated by the simulator to the `pre_randomize()` function in the `uvm_sequence_item` through the `rseed_interface` and use the built-in UVM function `reseed()` to re-seed a simulation. If an improvement is achieved in the objective function, we update the checkpoint and proceed to the next simulation period until the objective function is satisfied, i.e. the simulation has converged to the coverage goal.

The remaining chapters in the thesis report present the novel optimizations of the test environment proposed in this research and gives the implementation details. Empirical results showing the enhancements achieved through the optimizations are also presented.

5. OPTIMIZATION OF CONSTRAINED RANDOM VERIFICATION USING MACHINE LEARNING ALGORITHM

As the design complexity increases the state space defining the same increases rapidly. Hence, the probability of finding efficient stimuli diminishes. The aim of this research is to autonomously update the constraints during runtime in order to hit bins that have not been previously hit during the simulation, and hence converge to the coverage goal faster. This should ideally resolve the segmentation fault issue too since the simulation will converge much faster and eliminate the need for excess memory, which leads to the segmentation faults. There are multiple challenges that need to be addressed to efficiently implement the optimization proposed and to be able to converge to 100% coverage in one simulation run, including:

1. Autonomous update of constraints to trigger efficient stimuli on the inputs that generate the necessary outputs, so as to achieve an improvement in the objective function.
2. Develop design-independent, efficient techniques to update the constraints during runtime without having to recompile the updated environment.
3. Synchronize the enhancements with the other optimization techniques, including Objective Function, Rewinding and Dynamic Seed Manipulation, and integrating the environment.

All of the above-mentioned challenges have been efficiently resolved in the research work.

Machine Learning, as stated earlier, is the science of getting a computer to act without explicit programming. It uses training sets, which represent the relationships between inputs and outputs in the state space, to model a design. It is this property that we are going to exploit to resolve the scalability issue of the environment. Machine Learning, as the name suggests, learns the design, irrespective of what the design is, i.e. the algorithms are design-independent, resolving one of the challenges inherently. So, the next big challenge is employing the Machine Learning algorithm efficiently to autonomously update the constraints.

Functional coverage provides essential feedback for knowing what was tested, the device configuration used and, perhaps most importantly, what still has not been tested. The idea is to pass this information to the trained machine learning models, in order to identify the inputs that generate outputs that hit the previously missed bins and update the constraints using the same to facilitate the generation of these inputs. Since the aim is to find the function that best fits the state space to make accurate predictions of input stimuli, it is a regression problem. In order to solve the problem, we are going to use supervised machine learning algorithms, using labeled training sets to train the model. The training sets contain information necessary to train the model, which include the inputs and the expected output for the corresponding set of inputs as the label.

Presented below is a stepwise summary of the optimized constrained random simulation flow using machine learning as proposed in this research work:

1. Run a limited number of random simulations.
2. Generate valid training sets from the random simulations.
3. Once the minimum number of training sets necessary to train the Machine Learning model have been generated from the random simulations, train the model.
4. Identify an output bin that has not been hit previously during the simulation and feed the same as input to the Machine Learning model.
5. Use the model to predict the input stimuli that can generate the output.
6. Update the constraints such that the input stimuli identified may be generated and drive the inputs of the DUT with the same.
7. Repeat steps (4), (5) and (6) until all the bins have been hit and the simulation converges to 100% coverage.

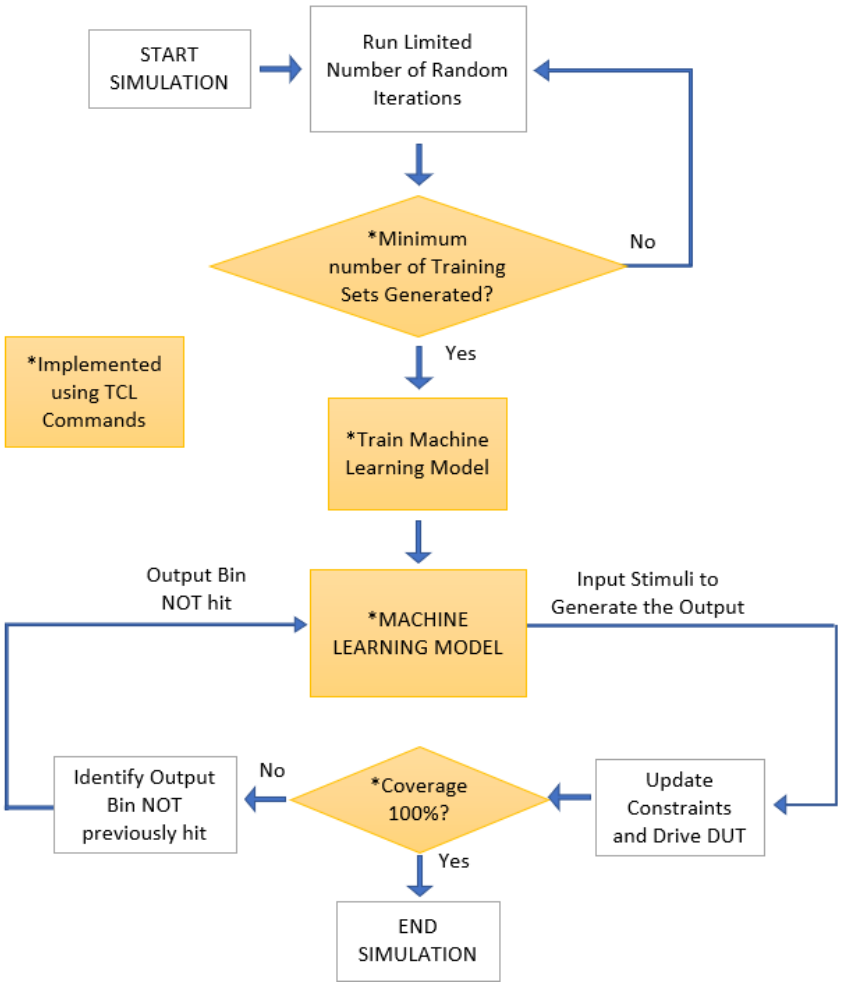

Figure 8: Optimized Constrained Random Simulations using Machine Learning

Irrespective of the Machine Learning algorithm used to model the relationship, the flow will remain the same. However, the accuracy of the prediction will greatly depend on the algorithm used. In order to demonstrate the method, we are going to use two models: a Linear Regression Model and an Artificial Neural Network (ANN).

6. OPTIMIZATION OF THE TEST ENVIRONMENT USING A LINEAR REGRESSION MODEL

For demonstration purposes, we have chosen the design of the comparator having configurable input widths, discussed earlier, as the DUT. A covergroup has been defined inside the DUT, with the internal signal `match` as the coverpoint, to check if all possible matching values have been generated on `A` and `B` at the positive edge of the clock. When `A` equates to `B` at the positive edge of the clock, the output `C` is asserted and `match` is assigned with `A`. Hence, we can see that there is a linear relation between the output, `match` and the inputs `A` and `B`. In such a scenario, a simple Machine Learning algorithm, like a Linear Regression Model will meet the requirements.

6.1 Generating Training Sets

An intriguing feature about this approach is the fact that the environment generates the necessary training sets by itself, making the approach completely independent and mitigating the need for any design-dependent inputs to run the simulation.

Structures defining the information to be contained in the training sets are defined in the environment package. These make addressing the training sets easier and also causes them to be less error-prone. For instance, training sets for modeling the relationships of the parameterized comparator contains information corresponding to the inputs `A` and `B`, the output, `match`, and a flag, `TS_ready`, indicating that the training set is ready, to train the model effectively.

```
typedef struct {
   bit [width-1:0] a;
   bit [width-1:0] b;
   bit [width-1:0] match;
   bit TS_ready;
} training_set;
```

The simulation samples valid data, i.e. when the inputs A and B are equal, during the random simulations and loads the data into the training_set and asserts the corresponding TS_ready signal. We have defined a function in the environment, generate_TS_and_track_hit_bins (presented below) to generate the no_of_TS_required number of training sets, TS and to track the output bins hit. The output bins hit are tracked using an associative array, OUT_HIT [match].

```
function void generate_TS_and_track_hit_bins(
                                        bit [width-1:0] a,
                                        bit [width-1:0] b,
                                        bit [width-1:0] match);

   // Load TS with "no_of_TS_required" number of training sets
   // [Input=0 and Match = 0] cannot be used for training
   if(i<no_of_TS_required && (match!=0))
      begin
        // Generating Training Sets
        TS[i].a        = a;
        TS[i].b        = b;
        TS[i].match    = match;
         TS[i].TS_ready = 1;

         // Tracking Output Bins hit
         OUT_HIT[match] = 1;
         i = i+1;
      end
   else
   begin
      OUT_HIT[match] = 1;
   end
endfunction
```

This function is called whenever the condition, `A==B` is met, to generate the necessary number of training sets and to keep track of the output bins hit. The number of training sets required to train a model can differ based on the design. To allow for this, the simulation is parameterized, with `no_of_TS_required` made to be configurable.

6.2 Training the Model

Now that the training sets are ready, we can use them to efficiently train the Linear Regression Model. We are using a TCL library, `math::linearalgebra,` in order to solve the linear regression problem and compute the parameters, `beta_a` and `beta_b`. Once the training sets are ready, i.e. once the `TS_ready` of `no_of_TS_required` number of training sets are all asserted, the `eval_coeff` function (presented below) from `rclass.tcl` is invoked to evaluate the coefficients.

```
proc ::rclass::eval_coeff {} {
    variable a0
    variable b0
    variable match0
    variable beta_a
    variable beta_b
    # Get the Training Sets
    set a0 [get top.dif.TS_a_0 -radix decimal]
    set b0 [get top.dif.TS_b_0 -radix decimal]
    set match0 [get top.dif.TS_match_0 -radix decimal]
    # Solve Linear Equation
    set beta_a [math::linearalgebra::solveGauss $a0 $match0]
    set beta_b [math::linearalgebra::solveGauss $b0 $match0]
    # Force Beta Values to SV Environment
    force top.rseed_interface.beta_a $beta_a;
    force top.rseed_interface.beta_b $beta_b;
    force top.rseed_interface.beta_ready 1; }
```

The computed parameters `beta_a` and `beta_b`, along with a flag `beta_ready` indicating that the Machine Learning model has been trained, are forced onto variables in the SystemVerilog environment and can now be used to update the constraints and generate inputs based on the expected outputs.

6.3 Updating the Constraints

Once the Machine Learning Model has been trained, i.e. `beta_ready` is asserted, we may use it to accurately predict the input stimuli and to use the same to update the constraints and generate outputs hitting previously missed bins.

We use queues to define in-line constraints, while randomizing the inputs feeding the DUT and dynamically update the queues during runtime to update the constraints. Presented below is the function `rprint` used for randomizing and printing the input `num,` with `num_inside_queue` used to define the inline constraints.

```
// randomize and print
function void rprint();
      this.randomize() with {(num inside num_inside_queue);};
      `uvm_info("CR", $sformatf("num is: %d", num), UVM_LOW)
endfunction
```

To identify the output bins that have not been previously hit we use the associative array keeping track of the output bins hit, `OUT_HIT[match]`. The inputs used for updating the queue, `num_inside_queue,` are computed from the identified expected outputs using the Linear Regression Model parameters i.e. the `beta_value`. The function used for updating the constraints, `update_constraint` is presented below:

```
function void update_constraint(integer beta_value);
      num_inside_queue = {};
      i = 0;
      repeat(2**width)
        if(!(OUT_HIT.exists(i))) begin
           num_inside_queue.push_back(i++/beta_value);
           break;
        end
        else i++;
endfunction
```

This novel implementation used for updating the constraints comes with the advantage that it eliminates the need to recompile the environment every time the constraint is updated. Such an implementation overcomes all the challenges foreseen and hence, the simulation should efficiently converge to the coverage goal autonomously, in a single run.

6.4 Simulation Results for Comparator Design

The simulation results were promising and remarkably good in terms of the quality of input stimulus generated using the optimized constrained random simulations, which in turn translated to minimization of the time taken to converge to the 100% coverage.

The simulation environment also maintains the capability of random simulations to invoke unpredicted bugs in the design. In order to achieve this, the simulation runs a configurable number of unconstrained random iterations in the beginning. The simulation is guided by a plusarg, `+max_rand_sim_count` that defines the number of random simulations run prior to employing the Machine Learning algorithm, even if an improvement is not observed in the Objective Function over the iterations.

To demonstrate the simulation results, a comparator with `width=3` is chosen. A Makefile, with adequate arguments, has been defined to configure the simulation environment efficiently. The simulation is run using:

```
make synopsys WIDTH=3 ML_ENABLED=1
```

`ML_ENABLED=1`, enables the Linear Regression model as the machine learning algorithm. `+max_rand_sim_count` was set to 10. During the random simulations, the valid training sets are generated, and outputs bins hit are tracked (highlighted in the log below). The matching inputs `A=4`, `B=4` will be used as the training set to train the Linear Regression Model.

```
UVM_INFO sv/dut.sv(26)@ 20: reporter [dut_if] AFTER drive regs A: 4 B: 4
UVM_INFO sv/env_pkg.sv(28) @ 26: reporter [ENV_PKG] A and B matching;
Generate training sets; Track output bins hit
rseed_interface.sv, 111 :             begin
------------------------ START eval_loop
DEBUG current simulation time is ctime : 27 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 2 at time: 17 ns
INFO STATUS : TCL : 27 ns : GOOD : 12.500000 > 0.000000
DEBUG stable_count == 0
------------------------ END eval_loop
```

The `stable_count` gives the number of iterations for which an improvement is not observed on the Objective Function. The machine learning algorithm is trained when this value equates to the `max_rand_sim_count` defined. Hence, when `stable_count=10`, the function, `eval_coeff` is called to evaluate the parameters (Highlighted in the log below).

```
------------------------ START eval_loop
DEBUG current simulation time is ctime : 67 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 1067 at time: 57 ns
```

```
INFO STATUS : TCL : 67 ns : NO PROGRESS : false: 50.000000 > 50.000000
REWINDING TO CHECKPOINT {2} at 57 ns
All the Checkpoints created after checkpoint 2 are removed...
DEBUG stable_count == 10
######################### START eval_coeff
INFO STATUS : TCL : Evaluating Coefficients for LR Model at time: 57 ns
Training Sets:
A0 = 4; MATCH = 4
B0 = 4; MATCH = 4
Evaluate Coefficients:
BETA_A = 1.0; BETA_B = 1.0
######################### END eval_coeff
------------------------ END eval_loop
```

Once the parameters, `beta_a` and `beta_b`, are evaluated using the training sets, every successive iteration of the SystemVerilog testbench updates the constraints using the parameters to guide the simulation to generate outputs hitting previously missed bins. Ideally, every iteration is expected to hit a previously missed bin, resulting in an improvement in the objective function. A snippet of the simulation log presenting two iterations is given below. The constraints are updated in every iteration, such that new matching values are generated on A and B every iteration, and hence, an improvement is observed in the objective function (Highlighted in the log),

```
UVM_INFO sv/env_pkg.sv(171) @ 80: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints
UVM_INFO sv/env_pkg.sv(196) @ 80: reporter@@uvm_sequence_item [ENV_PKG]
AFTER UPDATE num_inside_queue contain: '{'h2}
UVM_INFO sv/dut.sv(26)@ 80: reporter [dut_if] AFTER drive regs a: 2 b: 2
UVM_INFO sv/env_pkg.sv(28) @ 86: reporter [ENV_PKG] A and B matching;
Generate training sets; Track output bins hit
------------------------ START eval_loop
DEBUG current simulation time is ctime : 87 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 619 at time: 77 ns
```

```
INFO STATUS : TCL : 87 ns : GOOD : 75.000000 > 62.500000
------------------------- END eval_loop
UVM_INFO sv/env_pkg.sv(171) @ 90: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints
UVM_INFO sv/env_pkg.sv(196) @ 90: reporter@@uvm_sequence_item [ENV_PKG]
AFTER UPDATE num_inside_queue contain: '{'h3}
UVM_INFO sv/dut.sv(26)@ 90: reporter [dut_if] AFTER drive regs a: 3 b: 3
UVM_INFO sv/env_pkg.sv(28) @ 96: reporter [ENV_PKG] A and B matching;
Generate training sets; Track output bins hit
------------------------- START eval_loop
DEBUG current simulation time is ctime : 97 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 619 at time: 87 ns
INFO STATUS : TCL : 97 ns : GOOD : 87.500000 > 75.000000
------------------------- END eval_loop
```

The simulation proceeds until the coverage goal is met. The simulator prints a final report once the simulation converges to the 100% coverage goal.

```
INFO STATUS : TCL : WIDTH OF COMPARATOR = 3
INFO STATUS : TCL : ITERATIONS TOTAL = 24
INFO STATUS : TCL : final_report END
UVM_INFO sv/rseed_interface.sv(143) @ 117: reporter [RS]
COVERAGE GOAL MET coverage:  100    max_objective:  100
```

The simulation converges in 24 iterations (highlighted in the log) as opposed to 261 iterations in the case of the original environment. This represents significant improvement in terms of both the number of iterations required and the time taken to converge to the coverage goal.

The simulation environment was tested using multiple inputs widths. The results are tabulated below, presenting the number of iterations taken to converge to the coverage goal as a function of the width of the comparator.

Table 2: Simulation Results (Optimized Environment employing the Linear Regression Model)

#	Width of Comparator	No of Iterations
1	1	5
2	2	14
3	3	24
4	4	26
5	5	78
6	6	96
7	7	144

The optimized environment resolves the segmentation fault issues observed in the original environment. The increase in the number of iterations for higher widths is expected as it takes longer to generate valid training sets. The technique clearly demonstrates marked advantages over the prior approach. However, a Linear Regression model can only model linear relationships between inputs and outputs. It is impractical to verify designs with non-linear behavior using the environment.

6.5 Simulation Results for Non-Linear Design

To demonstrate the inefficiency of the environment using a Linear Regression Model at verifying non-linear designs, we have chosen the simple non-linear parameterized design shown in the figure below. The design has two inputs, `A` and `B`, with configurable sizes, corresponding to the parameter `width`. The output `C` is asserted when both the inputs have the same value. The internal signal `product` is assigned with the product of `A` and `B`, whenever `C` is asserted at a positive edge of the clock. Otherwise, it is assigned to be 0.

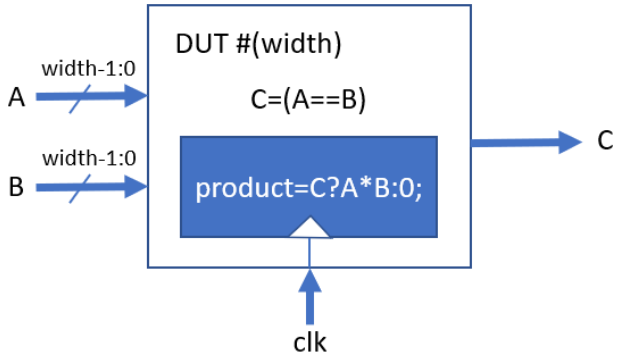

Figure 9: Non-Linear DUT used for testing Optimized Verification Environment

The covergroup is defined with the internal signal `product` as the coverpoint. The covergroup for a design with `width = 4` is presented below. The bins defined check only the basic functionality of the design, since the purpose is to demonstrate the test environment.

```
covergroup objective_cg;
      coverpoint product {
                        bins bin_1   = {1};
                        bins bin_4   = {4};
                        bins bin_9   = {9};
                        bins bin_16  = {16};
                        bins bin_25  = {25};
                        bins bin_36  = {36};
                        bins bin_49  = {49};
                        bins bin_64  = {64};
                        bins bin_81  = {81};
                        bins bin_100 = {100};
                        bins bin_121 = {121};
                        bins bin_144 = {144};
                        bins bin_169 = {169};
                        bins bin_196 = {196};
                        bins bin_225 = {225};
}
endgroup
```

The design is simple but non-linear (since it effectively computes the square when the inputs are equal). With the parameters `beta_a` and `beta_b` computed from the training sets, the model will not be capable of making accurate predictions. A snippet of the log file highlighting the computation of the parameters `beta_a` and `beta_b` using the training set `A=13, B=13` and `product=169` is shown below:

```
######################### START eval_coeff
INFO STATUS : TCL : Evaluating Coefficients for LR Model at time: 127 ns
DEBUG A0 = 13; PRODUCT = 169
DEBUG B0 = 13; PRODUCT = 169
BETA_A = 13.0; BETA_B = 13.0
######################### END eval_coeff
```

The simulation will be stuck in a deadlock wherein the Linear Regression Model keeps making incorrect predictions using the computed parameters. Consequently, the simulation will terminate giving a segmentation fault after some time. A snippet of the log file of a simulation terminating giving a segmentation fault (highlighted) is shown below:

```
DEBUG current simulation time is ctime : 137 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 64 at time: 127 ns
INFO STATUS : TCL : 137 ns : NO PROGRESS : false: 68.750000 > 68.750000
REWINDING TO CHECKPOINT {2} at 127 ns
All the Checkpoints created after checkpoint 2 are removed...
/usr/local/apps/synopsys/vcs/amd64/../gui/dve/bin/dve: line 159: 16805
Segmentation fault      (core dumped) $CMD "$@"
make: *** [sim_synopsys] Error 139
```

Hence, we may conclude that the verification environment employing Linear Regression Model is incapable of verifying non-linear designs. This limitation can be overcome by using Artificial Neural Networks.

7. OPTIMIZATION OF THE TEST ENVIRONMENT USING AN ANN

Artificial Neural Networks (ANN) are non-linear statistical data modeling tools that can model complex relationships between inputs and outputs in a state space. This ability may be exploited to optimize the scalability of our test environment and enhance for verifying complex non-linear designs. Successfully integrating ANNs into the verification environment largely eliminates any design dependency and resolves most challenges pertaining to the capability of the test environment to make accurate predictions for complex designs.

7.1 Modeling the Artificial Neural Network

We are using a TCL extension [7], `fann`, to model the Artificial Neural Networks. This extension supports efficient implementations of Neural Networks with numerous knobs to configure the network. The free open source neural network FANN library supports:

- Multilayer networks with configurable connections, enabling fully, sparse and shortcut type connected networks.
- Backpropagation training which dynamically builds and trains the ANN, using multiple evolving topology training algorithms and configurations.
- Numerous activation functions, including linear, sigmoid, Gaussian, etc.

Depending on the complexity of the relationship between the inputs and outputs of the design, we may configure the ANN in terms of the number of layers, the number of neurons per layer, training algorithm, activation function of each neuron, etc.

```
fann create name layers layer1 layer2 ....
```

The command above creates a new ANN named `name` with `layers` number of layers, and `layer1, layer2,....,` number of neurons per layer respectively, starting from the input layer and towards the output layer.

```
name function layer <0,1,...> activation_function

name function output activation_function
```

These commands may be used to configure the activation function of each of the neurons in the ANN, except for those in the input layer. Each neuron in the hidden layers and the output layer will be assigned with an activation function. The activation function may be selected from the set of available functions.

The FANN library also includes a framework for easy handling of the training sets. We can train the ANN using either of the two commands:

```
name trainondata epochs error input output

name trainonfile filepath epochs error
```

Detailed information regarding the other commands required for advanced configuration of the ANN to model complex design may be found in [7]. Once trained, the neural network may be used on unseen inputs to predict outputs. To run the ANN on unseen inputs, we may use the command:

```
name run input
```

7.2 Simulation Results for Comparator Design

To demonstrate the method and validate the environment, we have chosen first the parameterized comparator design with configurable input widths as the DUT. This simple design, though linear, was chosen to check for the correctness of the technique, and to make a quantitative comparison of the different methods explored in the research.

To demonstrate the simulation results, a comparator with `width`=3 was selected. A Makefile, with appropriate arguments, has been defined to configure the simulation environment efficiently. The simulation is run using:

```
make synopsys WIDTH=3 ML_ENABLED=2
```

ML_ENABLED=2 invokes the Artificial Neural Network as the machine learning algorithm. The technique used to generate the training sets and update the constraints is similar to that explained in previous sections for the Linear Regression Model.

The machine learning algorithm is trained when stable_count equates to the max_rand_sim_count defined. The function train_ANN is called to train the ANN. The function creates the ANN, defines the activation functions, and trains the ANN using the trainondata command, as shown below:

```
fann create ANN 2 1 1
ANN function hidden linear
ANN function output linear
ANN trainondata 500000 0 {TS_inputs} {TS_outputs}
```

A snippet of the log has been presented below. When stable_count=10, the function to train the ANN, train_ANN using two training sets is called (highlighted in the log below).

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 57 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 1140 at time: 47 ns
INFO STATUS : TCL : 57 ns : NO PROGRESS : false: 37.500000 > 37.500000
REWINDING TO CHECKPOINT {2} at 47 ns
All the Checkpoints created after checkpoint 2 are removed...
DEBUG stable_count == 10
************************ START train_ANN
INFO STATUS : TCL : TRAINING ANN at time: 47 ns
Training Sets:
A0 = 4; B0 = 4; MATCH0 = 4;
A0 = 3; B1 = 3; MATCH0 = 3;
************************ END train_ANN
------------------------- END eval_loop
```

Once the Neural Network is trained, every successive iteration of the SystemVerilog testbench updates the constraints using the neural network to guide the simulation to generate outputs hitting previously missed bins. Ideally, every iteration is expected to generate outputs hitting a previously missed bin, resulting in an improvement in the objective function. We use the `run` command from the `fann` library for the same. Presented in the snippet below are two iterations using the `run_ANN` function which predicts the inputs, used to update the queues, from the output bins previously missed (highlighted). Also highlighted is the improvement in the objective function on driving `A` and `B` with the predicted inputs.

```
UVM_INFO sv/env_pkg.sv(205) @ 60: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints (using ANN)
************************ START run_ANN
INFO STATUS : TCL : RUNNING ANN at time: 60 ns
INFO_STATUS : TCL : Output Bin NOT Hit: 1; Predicted Input to Update
Constraint Queue: 1
************************ END run_ANN
UVM_INFO sv/env_pkg.sv(305) @ 61: uvm_test_top [ENV_PKG] AFTER UPDATE
num_inside_queue contain: '{'h1}
UVM_INFO sv/dut.sv(37)@ 61: reporter [dut_if] AFTER drive regs A: 1 B: 1
UVM_INFO sv/env_pkg.sv(29) @ 66: reporter [ENV_PKG] Generate training
sets; Track output bins hit
------------------------- START eval_loop
DEBUG current simulation time is ctime : 67 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 319 at time: 47 ns
INFO STATUS : TCL : 67 ns : GOOD : 50.000000 > 37.500000
------------------------- END eval_loop
UVM_INFO sv/env_pkg.sv(205) @ 70: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints (using ANN)
************************ START run_ANN
INFO STATUS : TCL : RUNNING ANN at time: 70 ns
INFO_STATUS : TCL : Output Bin NOT Hit: 2; Predicted Input to Update
Constraint Queue: 2
```

```
************************ END run_ANN
UVM_INFO sv/env_pkg.sv(305) @ 71: uvm_test_top [ENV_PKG] AFTER UPDATE
num_inside_queue contain: '{'h2}
UVM_INFO sv/dut.sv(37)@ 71: reporter [dut_if] AFTER drive regs A: 2 B: 2
UVM_INFO sv/env_pkg.sv(29) @ 76: reporter [ENV_PKG] Generate training
sets; Track output bins hit
------------------------- START eval_loop
DEBUG current simulation time is ctime : 77 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 319 at time: 67 ns
INFO STATUS : TCL : 77 ns : GOOD : 62.500000 > 50.000000
------------------------- END eval_loop
```

The simulation proceeds until the coverage goal is met. A final report is printed summarizing the simulation once the simulation converges to the 100% coverage goal.

```
INFO STATUS : TCL : WIDTH OF COMPARATOR = 3
INFO STATUS : TCL : ITERATIONS TOTAL = 23
INFO STATUS : TCL : final_report END
UVM_INFO sv/rseed_interface.sv(161) @ 117: reporter [RS]
COVERAGE GOAL MET coverage:  100     max_objective:  100
```

The simulation converged to the 100% coverage goal in 23 iterations (highlighted above) as opposed to 261 iterations in the case of the original environment. It is also similar to the 24 iterations required using the Linear Regression Model.

Simulations were carried out for multiple inputs widths. The results have been tabulated, presenting the number of iterations taken to converge to the coverage goal as a function of the width of the comparator.

Table 3: Simulation Results (Optimized Environment employing ANN)

#	Width of Comparator	No of Iterations
1	1	3
2	2	9
3	3	23
4	4	27
5	5	62
6	6	170
7	7	220

Employing ANNs resolve the segmentation faults observed in the original environment. In terms of number of iterations, the environment using ANN as the machine learning algorithm gives similar trends as when using Linear Regression Model. However, the ANN can be applied in a much wider set of applications, as will be shown in the remainder of this thesis.

7.3 Simulation Results for Non-Linear Design

To demonstrate the efficiency of the environment at verifying non-linear designs, we have chosen the same non-linear parameterized design used earlier (shown in Figure 9). Experimental results presented earlier in the report have shown that it is impractical to verify the non-linear design using a Linear Regression model. Hence, successfully verifying the design using the ANN, accentuates the role of ANNs in scaling the environment.

The Neural Network is trained after the initial random simulations, i.e. when `stable_count` equates to the specified `max_rand_sim_count`. When `stable_count=10,` the function

`train_ANN` is called to load the trained ANN. The function loads the ANN trained using the `trainonfile` command. We are using a sigmoid activation function for the hidden layer, so that the ANN can deal with non-linear functions. Presented below are the TCL functions used to create, configure and train the ANN.

```
fann create ANN 3 1 30 1
ANN function output linear
ANN function layer 0 sigmoid
ANN trainonfile <filepath> 500000 0
```

The trained ANN is loaded, when `stable_count` equates to 10, which indicates that the objective function has been stable for 10 continuous iterations. A snippet of the log presenting the same is given below. Highlighted is the function `train_ANN` used to train the ANN, called when `stable_count` equates to 10.

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 57 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 909 at time: 47 ns
INFO STATUS : TCL : 57 ns : NO PROGRESS : false: 18.750000 > 18.750000
REWINDING TO CHECKPOINT {2} at 47 ns
All the Checkpoints created after checkpoint 2 are removed...
DEBUG stable_count == 10
************************ START train_ANN
INFO STATUS : TCL : TRAINING ANN at time: 47 ns
INFO STATUS : TCL : ANN Trained using Training Set
INFO STATUS : TCL : Trained ANN loaded
INFO STATUS : TCL : Training DONE
************************ END train_ANN
------------------------- END eval_loop
```

Once the trained ANN is loaded, ideally the ANN must be capable of predicting the inputs producing outputs hitting previously missed bins, such that every successive iteration of the

SystemVerilog testbench updates the constraints using the neural network to guide the simulation to achieve an improvement in the objective function. We use the run command from the fann library for the same. Presented in the snippet below are two iterations using the run_ANN function which predicts the inputs, used to update the queues, from the output bin previously missed (highlighted). Also highlighted is the improvement in objective function on driving A and B with the predicted inputs.

```
UVM_INFO sv/env_pkg.sv(206) @ 120: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints (DNN)
************************ START run_ANN
INFO STATUS : TCL : RUNNING ANN at time: 120 ns
INFO_STATUS : TCL : Output Bin NOT Hit: 64; Predicted Input to Update
Constraint Queue: 8
************************ END run_ANN
UVM_INFO sv/env_pkg.sv(305) @ 121: uvm_test_top [ENV_PKG] AFTER UPDATE
num_inside_queue contain: '{'h8}
UVM_INFO sv/dut.sv(37)@121:reporter [dut_if] AFTER drive regs A: 8  B: 8
UVM_INFO sv/env_pkg.sv(29) @ 126: reporter [ENV_PKG] Generate training
sets; Track output bins hit
------------------------- START eval_loop
DEBUG current simulation time is ctime : 127 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 1141 at time: 117 ns
INFO STATUS : TCL : 127 ns : GOOD : 62.500000 > 56.250000
------------------------- END eval_loop
UVM_INFO sv/env_pkg.sv(206) @ 130: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints (DNN)

************************ START run_ANN
INFO STATUS : TCL : RUNNING ANN at time: 130 ns
INFO_STATUS : TCL : Output Bin NOT Hit: 81; Predicted Input to Update
Constraint Queue: 9
************************ END run_ANN
UVM_INFO sv/env_pkg.sv(305) @ 131: uvm_test_top [ENV_PKG] AFTER UPDATE
num_inside_queue contain: '{'h9}
```

```
UVM_INFO sv/dut.sv(37)@131:reporter [dut_if] AFTER drive regs A: 9  B: 9
UVM_INFO sv/env_pkg.sv(29) @ 136: reporter [ENV_PKG] Generate training
sets; Track output bins hit
------------------------ START eval_loop
DEBUG current simulation time is ctime : 137 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 1141 at time: 127 ns
INFO STATUS : TCL : 137 ns : GOOD : 68.750000 > 62.500000
DEBUG stable_count == 0
------------------------ END eval_loop
```

The simulation proceeds until the coverage goal is met. Once the simulation converges to the 100% coverage, a final report is printed, presenting the number of iterations.

```
INFO STATUS : TCL : ITERATIONS TOTAL = 29
INFO STATUS : TCL : final_report END
COVERAGE GOAL MET coverage:  100    max_objective:  100
```

The simulation converges in 29 iterations (highlighted above). Simulations were carried out for multiple inputs widths. The results have been tabulated, presenting the number of iterations taken to converge to the coverage goal as a function of the width of the design.

Table 4: Simulation Results (Optimized Environment employing ANN)

#	Input Width	No of Iterations
1	1	3
2	2	16
3	3	24
4	4	29
5	5	96

These results show that the ANN is capable of making good predictions even for non-linear designs.

8. TEST ENVIRONMENT FOR VERIFYING AN FSM

One key aspect about the verification environment in [4] is that it is based on rewinding the simulation when progress is not observed on the objective function. Adding to the increased complexity, sequential circuits, unlike combinational circuits, can have scenarios wherein no (or very few) combinations of inputs can result in an increase in the objective function. Experimental results presented earlier in the report have shown that the simulation using the environment in [4] can get stuck in a deadlock and it may not converge to 100% coverage in one simulation run in such a scenario.

One simple optimization that can resolve this issue will be to reset the design when the simulation is stuck and the objective function is observed to be stable for a predefined number of clock cycles. The simple optimization can get the simulation out of a possible deadlock condition. To make this simple check, we keep track of the number of iterations the objective function has been stable for. We use `stable_count` during the execution of the function `eval_loop` in `rclass.tcl`, as the indicator. When `stable_count` exceeds a maximum value, the function to generate a reset from within the SystemVerilog environment is invoked. Presented below is the code snippet implementing this approach:

```
# Check if objective function has not improved
if { [expr "$new_objective" == "$old_objective"] } {
      # Increment Count
      incr stable_count;
      if {$stable_count >= $max_rand_sim_count} {
        # Call function to generate reset
        call top.rseed_interface.generate_reset();
        set stable_count 0; }
}
```

The technique can be further optimized by combining the reset generation with a method to constrain the inputs such that previously taken branches are avoided. This implies that the simulation will progress by taking a previously missed branch, thereby likely improving the objective function. The set of valid input combinations that can cause a state transition in an FSM will be small, as compared to the set of all possible combinations at the input. Hence, hardcoding the constraints using this set of valid combinations is a simple and effective solution. An array `array_to_update_queue` holding the set is passed to the function `update_constraint`, used for updating the constraint. A snippet of the driver code calling the functions to generate a reset `drive_reset` and to update the constraints `update_constraint` is given below:

```
// this is the forever loop that represents the UVM driver
forever begin
      @(negedge dif.Clock);
       if (rseed_interface.generate_pulse_on_reset == 1)
      begin
            // Drive Reset on DUT interface
            dif.drive_reset;
            // Function to update constraint
            c_a.update_constraint(array_to_update_queue);
            c_b.update_constraint(array_to_update_queue);
      end
      else
      begin
            // randomize class txns using inline constraints
            c_a.rprint();
            c_b.rprint();
            // Drive DUT inputs
            dif.drive(c_a.get_num(), c_b.get_num());
      end
end
```

Currently the constraints are hardcoded. This may be optimized by automating the constraint update using Neural Networks; but this will require retraining of the ANN in case of mispredictions and is left as future work.

8.1 Simulation Results for FSM Designs

For demonstration purpose we have chosen the same FSM that was used for experimenting with the original environment from the prior work [4], shown in figure below. It was empirically shown that the simulation using the verification environment in [4] gets stuck at 60% coverage and does not proceed any further. The simulation terminates giving a segmentation fault after some time. The reasons for this have been presented earlier in the report.

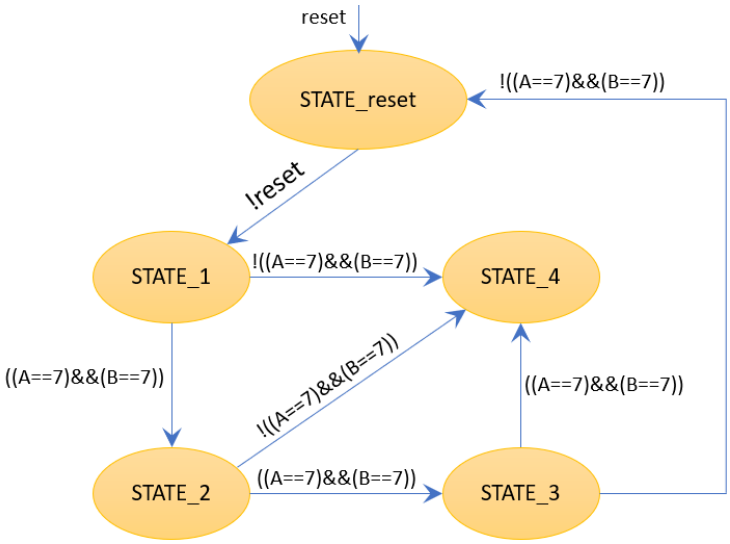

Figure 10: FSM Design used for Testing Optimized Verification Environment

Ideally, the optimized environment is expected to detect such a deadlock and generate a reset in such scenarios. Also, the constraints should be updated to guide the simulation to proceed in the desired direction.

If the simulation is found to be stuck in a state for `n` iterations, where `n` is a configurable parameter, we assume that the simulation is stuck in a deadlock. For the current simulation we have set `n=10`. Hence, when the `stable_count`, which gives the number of iterations for which the objective function has not improved, equates to n, we call the function to generate the reset and update the constraints to guide the simulation. Presented below is the log file of the simulation using the updated environment. The function to generate a reset is called when `stable_count` equates to 10, after which the constraints are updated such that the FSM hits the previously missed states, STATE_2 and STATE_3, and converges to meet the coverage goal (Highlighted in the log below).

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 37 ns
INFO STATUS : TCL : LOCAL REJECTED seed: 819 at time: 27 ns
INFO STATUS : TCL : 37 ns : NO PROGRESS : false: 60.000000 > 60.000000
REWINDING TO CHECKPOINT {2} at 27 ns
All the Checkpoints created after checkpoint 2 are removed...
DEBUG stable_count == 10
DEBUG "Generate Reset" Function called
------------------------- END eval_loop
UVM_INFO sv/dut.sv(20) @ 30: reporter [dut_if] Pulse generated on Reset
UVM_INFO sv/dut.sv(95) @ 35: reporter [dut] Current State: STATE_reset
UVM_INFO sv/env_pkg.sv(127) @ 45: reporter@@uvm_sequence_item [ENV_PKG]
Updating Constraints
UVM_INFO sv/env_pkg.sv(132) @ 45: reporter@@uvm_sequence_item [ENV_PKG]
AFTER UPDATE num_inside_queue contain: '{'h7}
UVM_INFO sv/dut.sv(15)@ 50: reporter [dut_if] AFTER drive regs A: 7 B: 7
UVM_INFO sv/dut.sv(100) @ 50: reporter [dut] Current State: STATE_1
UVM_INFO sv/dut.sv(105) @ 55: reporter [dut] Current State: STATE_2
```

```
------------------------- START eval_loop
DEBUG current simulation time is ctime : 57 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 898 at time: 27 ns
INFO STATUS : TCL : 57 ns : GOOD : 80.000000 > 60.000000
------------------------- END eval_loop
UVM_INFO sv/dut.sv(15)@ 60: reporter [dut_if] AFTER drive regs A: 7 B: 7
UVM_INFO sv/dut.sv(110) @ 65: reporter [dut] Current State: STATE_3
------------------------- START eval_loop
DEBUG current simulation time is ctime : 77 ns
INFO STATUS : TCL : LOCAL ACCEPTED seed: 898 at time: 57 ns
INFO STATUS : TCL : 77 ns : GOOD : 100.000000 > 80.000000
INFO STATUS : TCL : MET OBJECTIVE!
------------------------- END eval_loop
UVM_INFO sv/dut.sv(113) @ 75: reporter [dut] Current State: STATE_4
UVM_INFO sv/rseed_interface.sv(138) @ 77: reporter [RS]
COVERAGE GOAL MET coverage:  100  max_objective:  100
INFO STATUS : TCL : ITERATIONS TOTAL = 13
INFO STATUS : TCL : final_report END
```

The simulation no longer gets stuck in any deadlock. When a deadlock is detected, a reset is generated and the constraints are updated to guide the simulation to proceed in a desired direction. Simulation converges to the 100% coverage, proving that the optimization works as expected.

9. CONCLUSION

With the rapid scaling of technology nodes, more and more functionality is getting packed on to the same-sized die, translating to increasingly complex designs. Accordingly, the verification of such SoCs is becoming a greater challenge, impacting the time-to-market. Constrained random simulations have become mainstream in ASIC and FPGA verification. But the effort spent to analyze and prioritize unverified features and time spent on manually updating stimulus constraints to close the coverage holes introduce a significant amount of unwanted delay in the time-to-market of the product. This manual intervention is the biggest hindrance in the verification cycle of complex designs.

This research work addresses these challenges by introducing an efficient, design-independent technique to autonomously update the constraints using machine learning algorithms in order to converge to the coverage goal faster. The test environment is further optimized using techniques including Objective Function, Rewinding and Dynamic Seed Manipulation. The SystemVerilog testbench, enhanced with multiple machine learning algorithms including a Linear Regression model and Artificial Neural Networks, has been empirically shown to converge to the 100% coverage goal much faster. A link to the GitHub repository with the proposed test environment can be found in [9].

10. FUTURE WORK

The current environment waits for a minimum number of training sets to be generated prior to training the neural network. In case of complex designs which might require a large number of training sets to efficiently train the ANN to make accurate predictions, such a methodology will stall the training and hence will take a longer time to converge to the coverage goal, or might even terminate giving a segmentation fault. Though workarounds have been implemented in the current environment to avoid this stalling, by using the `trainonfile` option to train the ANN, this will require the user to provide valid training sets.

A solution to this would be to optimize the test environment to be capable of retraining the neural network in case of mispredictions. This would eliminate the need to wait until all the training sets are available to train the ANN. It will keep the verification environment completely design-independent, without the need for any design-dependent inputs from the user, and also scale the test environment even further to be capable of verifying much more complex designs.

REFERENCES

[1] Constrained Random Verification, Yuan J.; Pixley, C.; Aziz, A.; 2006, XII, 254p. 72 Illus., Hardcover, ISBN: 978-0-387-25974-5.

[2] Accellera, Universal Verification Methodology (UVM) 1.2 Class Reference (2014)

[3] Accellera, Universal Verification Methodology (UVM) 1.2 User's Guide (2015)

[4] Eldon Nelson, "Improving Constrained Random Testing by Achieving Simulation Verification Goals through Objective Functions, Rewinding and Dynamic Seed Manipulation", Design and Verification Conference (DVCon), Feb 2017.

[5] Panos Louridas and Christof Ebert, "Machine Learning", IEEE Software, (Volume: 33, Issue: 5, Sept.-Oct. 2016), DOI: 10.1109/MS.2016.114.

[6] Carter, Hamilton B., Hemmady and Shankar G., "Metric Driven Design Verification", 2007.

[7] TCL FANN Extension -http://tcl-fann.sourceforge.net/

[8] Dejan Tanikić and Vladimir Despotovic, "Artificial Intelligence Techniques for Modeling of Temperature in the Metal Cutting Process", Metallurgy Yogiraj Pardhi, IntechOpen, September 2012, DOI: 10.5772/47850.

[9] GitHub Repository: https://github.com/sarath-mohan/optimize-constrained-random-using-machine-learning

[10] Nguyen Le and Mike Andrews, "Efficient Bug-Hunting Techniques Using Graph-Based Stimulus Models", Design and Verification Conference (DVCon) US, 2016.

[11] Sheena Angra and Sachin Ahuja, “Machine Learning and its Applications: A Review”, International Conference on Big Data Analytics and Computational Intelligence (ICBDAC), 2017, DOI: 10.1109/ICBDACI.2017.8070809

[12] “IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,” IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009), pp. 1 –1315, 2013.

[13] Stan Sokorac, “Optimizing Random Test Constraints Using Machine Learning Algorithms”, Design and Verification Conference (DVCon), Feb 2017.

[14] Ahmed Yehia, “Faster Coverage Closure: Runtime Guidance of Constrained Random Stimuli by Collected Coverage”, Saudi International Electronics, Communications and Photonics Conference (SIECPC), 2013, DOI: 10.1109/SIECPC.2013.655100.

[15] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning”, Nature, vol. 521, no. 7553, pp. 436–444, May 2015.

[16] Chris Kwok, Priya Viswanathan and Ping Yeung, “Addressing the Challenges of Reset Verification in SoC Designs”, Design and Verification Conference (DVCon), 2015.