Simulation Runtime Optimization of Constrained Random Verification using Machine Learning Algorithms

Sarath Mohan Ambalakkat M.S.
University of Minnesota (ambal006@umn.edu)

Eldon Nelson M.S. P.E.
Synopsys, Inc. (eldon_nelson@ieee.org)

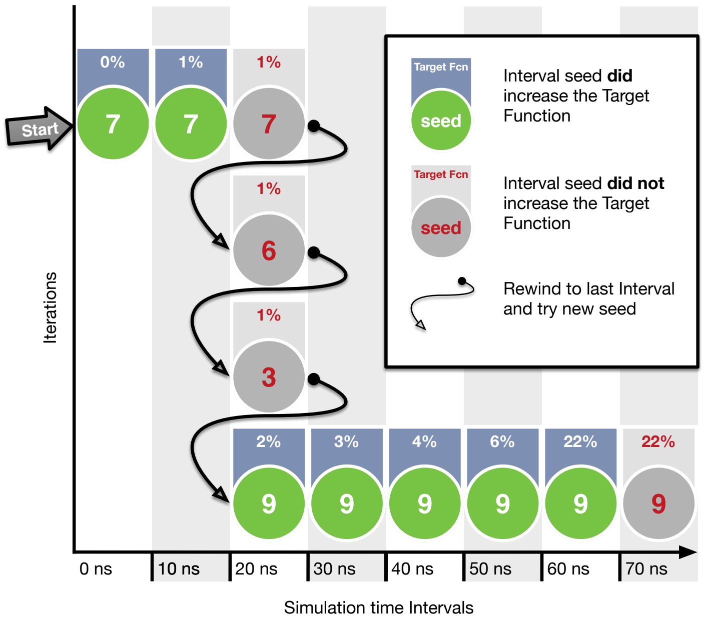
Abstract–**Constrained random simulations play a critical role in design verification today. Analyzing and prioritizing the unverified features in the design significantly affects the engineering time taken to converge a design's coverage goal. This research focuses on optimization of constrained random verification using machine learning algorithms. Machine learning is used to modify randomization constraints during simulation runtime in response to instrumentation of the DUT. Simpler algorithms such as linear regression and more complicated algorithms such as artificial neural networks augment the simulation as it attempts to predict useful stimulus. The proposed enhanced environment resolves some limitations of the previous efforts proposed in a DVCon 2017 paper by Nelson [1] optimizing the scalability of the environment and enhancing its compatibility at verifying complex combinatorial designs and sequential designs including finite state machines (FSMs).**

I. Introduction

The DVCon San Jose paper from 2017 "Improving Constrained Random Testing by Achieving Simulation Design Goals through Target Functions, Rewinding and Dynamic Seed Manipulation" by Nelson [1] was extended to tackle some of its admitted deficiencies. The Nelson [1] paper used introspection of the DUT to see if the stimulus was making an impact in exploring the design; and if there was no impact of that stimulus, the simulation was rewound and new completely random stimulus was explored.

The concept of the "Objective Function" is re-introduced here, which is a measure of a user-specified function queried from within the SystemVerilog environment. An "Objective Function" could simply be defined by the user as: calling the covergroup built-in function `get_coverage()` on a particular covergroup. A more ambitious goal would be adding the resultant `get_coverage()` values from a number of related covergroups and using that as the "Objective Function". The "Objective Function" could be as general as calling the SystemVerilog built-in function `$get_coverage()`, which returns a floating point number between 0 and 100 representing the aggregate value of all covergroups in the design. The "Objective Function" could be any function that returns a number that increases in value as it approaches its goal. Therefore, it is possible that using something as abstract as line coverage of the DUT could be used as the "Objective Function", which could be useful to demonstrate a particular stimulus is activating lines of code not yet exercised.

This research adds different algorithms – and the concept of dynamic constraint modification - to the feedback loop of the Nelson [1] paper which results in efficiency improvements. This research optimizes constrained random simulations by automating the update of constraints during simulation runtime. The method employs machine learning algorithms to analyze and prioritize unverified features autonomously, by using 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 reduces the effort and time taken to manually update constraints and converge to the coverage goal.



Start
0%
1%
1%
7
7
7
1%
6
1%
3
2%
3%
4%
6%
22%
22%
9
9
9
9
9
9
Iterations
0 ns
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
Simulation time Intervals
Target Fcn
seed
Interval seed **did** increase the Target Function
Target Fcn
seed
Interval seed **did not** increase the Target Function
Rewind to last Interval and try new seed

FIGURE 1 COMBINING SEED MANIPULATION WITH SIMULATION REWIND FROM 2017 NELSON PAPER

The provided verification environment has been equipped with multiple machine learning algorithms, including a linear regression model and artificial neural network that model the predictor. This approach improves the scalability of the environment, enhancing its capability at verifying varied designs with different behaviors. The data training sets, used for training the machine learning algorithms, are implicitly generated by the verification environment during the simulation, reducing the need for design-dependent information.

II. DRAWBACKS OF PRIOR WORK

Verification using the simulation environment defined in the prior work Nelson [1] 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 Nelson [1] 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 block diagram of the design being verified is given in the figure below:

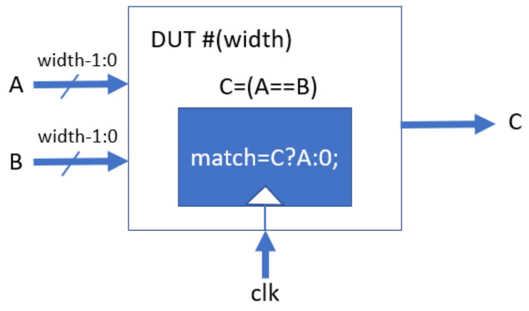


FIGURE 2. DUT USED FOR TESTING THE ORIGINAL ENVIRONMENT (AFTER [1])

The Design Under Test (DUT) is a 2-input parameterized comparator is depicted in
Figure 2. 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. A summary of the simulation results for comparators of different widths using the simulation environment is shown in
Figure 3 (data is median of 11 trials for each width).

Width of Comparator	No of Iterations
1	3
2	34
3	145
4	786
5	3967
6	18958

FIGURE 3. SIMULATION RESULTS (USING ENVIRONMENT IN [1])

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. 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.

III. Optimization of Constrained Random Verification using Machine Learning Algorithms

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.

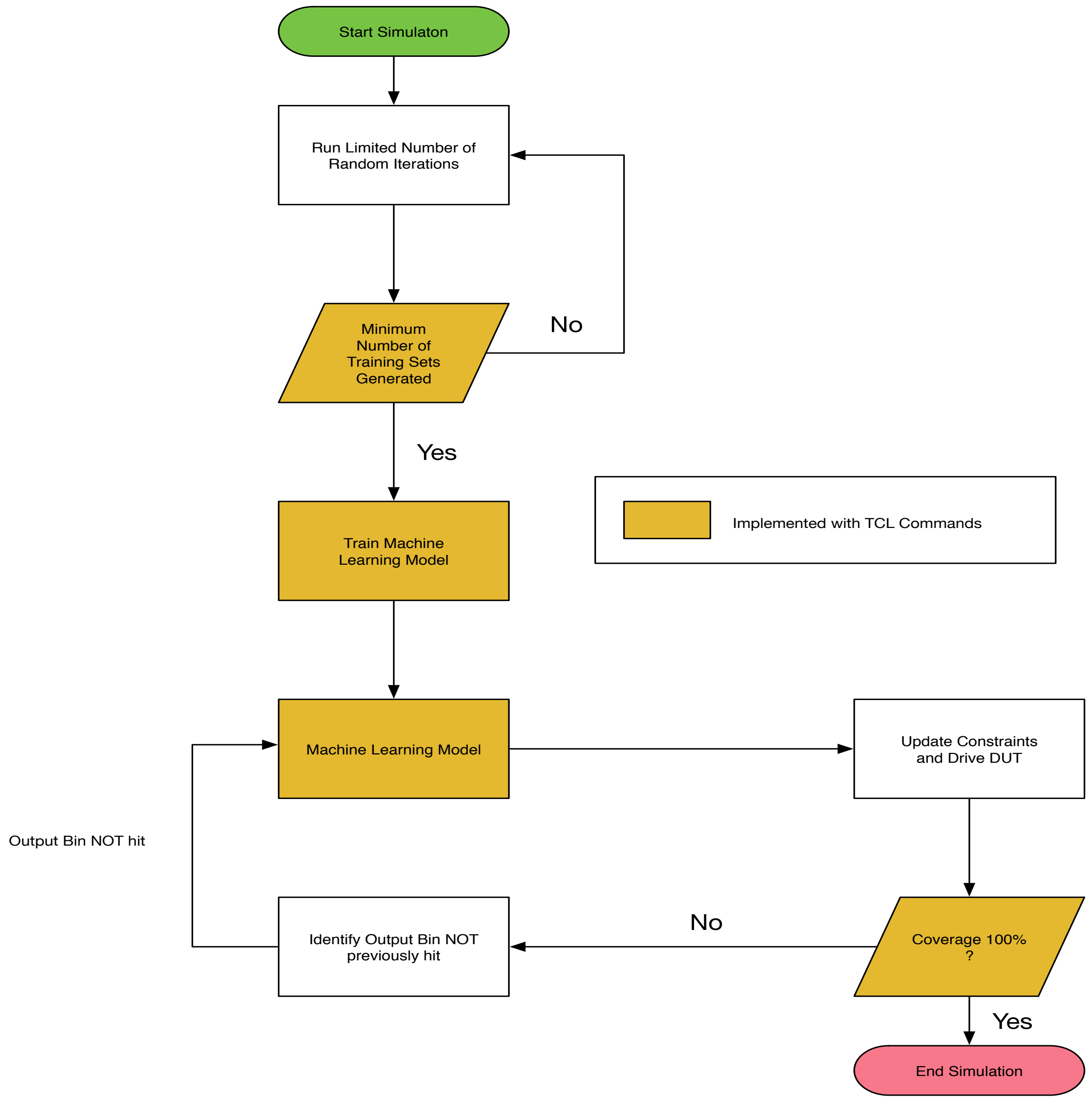
All of the above-mentioned challenges have been efficiently resolved in the research work.

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.



Start Simulaton
Run Limited Number of Random Iterations
Minimum Number of Training Sets Generated
No
Yes
Train Machine Learning Model
Implemented with TCL Commands
Machine Learning Model
Update Constraints and Drive DUT
Output Bin NOT hit
Identify Output Bin NOT previously hit
No
Coverage 100% ?
Yes
End Simulation

FIGURE 4. OPTIMIZED CONSTRAINED RANDOM SIMULATIONS USING MACHINE LEARNING

Irrespective of the Machine Learning algorithm used to model the relationship, the flow in Figure 4 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).

IV. 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.

V. 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, assuring that the simultaneously sampled inputs and outputs are used for training the machine learning algorithm. 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.

VI. 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.

VII. 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.

VIII. 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 simulation 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 145 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.

#	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

FIGURE 5: SIMULATION RESULTS (OPTIMIZED ENVIRONMENT EMPLOYING THE LINEAR REGRESSION MODEL)

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.

IX. 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 to verify 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.

X. MODELING THE ARTIFICIAL NEURAL NETWORK

We are using a TCL extension `fann` [2], 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 [3]. 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
```

XI. Simulation Results for Non-Linear Design

To demonstrate the efficiency of the environment using an Artificial Neural Networks 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.

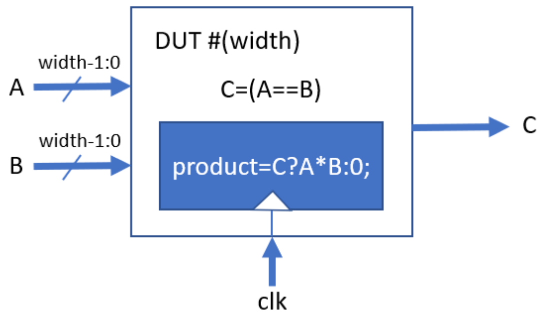


FIGURE 6. NON-LINEAR DUT USED FOR TESTING OPTIMIZED VERIFICATION ENVIRONMENT

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). 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.

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

FIGURE 7 SIMULATION RESULTS (OPTIMIZED ENVIRONMENT EMPLOYING ANN)

Figure 7 shows that the ANN is capable of making good predictions even for non-linear designs.

XII. SUMMARY

This research 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 expands previous technique 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 the previous research result from 2017 with fewer iterations. A GitHub repository [2] is provided with full source code for the examples provided.

Though the design used to prove the technique is quite simple, the environment used to demonstrate the methodology should ideally be capable of handling much more complex combinational designs. However, for verifying sequential designs, the environment will have to be optimized by modelling the Machine Learning algorithms using languages like Python which has many modules, including numpy, scipy and scikit-learn, that can help programmers implement machine learning more efficiently. This has been left as future work.

REFERENCES

[1] E. Nelson, "Improving Constrained Random Testing by Achieving Simulation Design Goals through Target Functions, Rewinding and Dynamic Seed Manipulation," in *Design and Verification Conference*, San Jose, USA, 2017.

[2] S. Ambalakkat, "optimize-constrained-random-using-machine-learning," GitHub, 2018. [Online]. Available: https://github.com/sarath-mohan/optimize-constrained-random-using-machine-learning. [Accessed 2 12 2018].

[3] A. Stergiakis, "tcl-fann," 2008. [Online]. Available: http://tcl-fann.sourceforge.net. [Accessed 3 12 2008].