Reaching 100% Functional Coverage Using Machine Learning: A Journey of Persistent Efforts

Jaecheon Kim[1], Taewook Nam[2], Wonil Cho[3]

Samsung Electronics Co., Ltd.
([1]jcheon05.kim; [2]tw.nam; [3]wonil.cho@samsung.com)
1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18448, Republic of Korea

***Abstract* - With the increasing complexity of IP RTL designs, it has become more difficult for design verification engineers to achieve functional coverage closure based on constrained random verification. This is one of the critical factors in delaying verification time. To address this issue, this paper proposes an automated functional coverage closure technique using machine learning and presents various strategies to enhance the accuracy of machine learning model predictions, enabling quick filling of functional coverage holes. This technique focuses on cross coverage and utilizes a Python-based coverage agent. The proposed technique was evaluated by applying it to the Camera Serial Interface (CSI) design, which is part of the Image Signal Processor (ISP) block in mobile SoC, and achieved functional coverage closure significantly faster than the baseline without much effort from the DV engineer.**

Keywords - Constrained Random Verification, Functional Coverage, Cross Coverage, Machine Learning

I. INTRODUCTION

IP RTL designs are becoming increasingly complex to meet market demands on enhanced features. This complexity makes it more challenging for design verification (DV) engineers to achieve functional coverage closure based on random constrained verification. This leads DV engineers to demand more effort and pain for functional coverage closure. Moreover, it is one of the critical factors in verification turn-around-time (TAT). Many DVCON papers related to functional coverage closure support these facts [1],[2],[3]. These papers demonstrate how to generate meaningful test stimuli in vast input spaces to achieve fast and low-effort functional coverage closure through various approaches. In this regard, this study aligns with these previous works and presents a different approach.

The cross coverage when creating a functional coverage model involves any combination of more than two variables or previously declared coverage points [4]. It helps to verify if the behavior of a specific combination of device under test (DUT) is covered in the test. According to our experience, when the number of tests increase during functional coverage measurement, each coverpoint defined in the coverage model is easily filled, but cross coverage, which is made up of combinations of coverpoints, is often not well filled. To address the cross coverage bins that are not fully filled, DV engineers may analyze the coverage results generated by the tool, run closely related scenarios more intensively, adjust the distribution of related configurations, or assign certain values directly. Regardless of whether there are other methods in addition to those mentioned, analyzing the results and taking action requires a significant amount of effort and time from engineers.

Based on our heuristic approach, we identified cross coverage measurement as a major factor contributing to the functional coverage closure challenge. This study analyzes the causes why cross coverage is not well filled and proposes an automated functional coverage closure using machine learning, which is focused on cross coverage, and introduces a Python-based coverage agent that supports this workflow. This paper also shows the use of various strategies to apply machine learning model predictions to enhance the accuracy of predictions. This ultimately leads to a gradual increase in the confidence of the outcomes created by the coverage agent to fill function coverage holes.

These results allow functional coverage to be filled aggressively and quickly without the intervention of a DV engineer. This approach was evaluated by applying it to the Camera Serial Interface (CSI) design, which is part of Image Signal Processor (ISP) block in mobile SoC. As a result, the proposed approach achieved functional coverage closure substantially faster than the baseline without much effort from the DV engineer.

II. BACKGROUND

In this section, we analyze why cross coverage bins are not well filled in when measuring functional coverage through several experiments. To achieve this, we declared four random variables, test_a, test_b, test_c, and test_d, as shown in Figure 1(a), and made them have an equal weighted distribution. These variables have 2, 6, 5, and 7 possible values, respectively. Therefore, the cross coverage, cross_test_abcd, consists of a total of 420 bins. Figure 1(b) shows the results of 5,000 randomizations of these random variables, with the y-axis representing the frequency at which each combination of random variables was generated out of 5,000 trials, and sorted in descending order along the x-axis. The graph displays an uneven distribution, contrary to expectations. Even though the probability distribution of the random variables is equal, there is a significant difference between the maximum value (25) and the minimum value (3) of the frequency distribution, as evidenced by Figure 1(c). This is because the number of runs (5,000) may not be large enough.

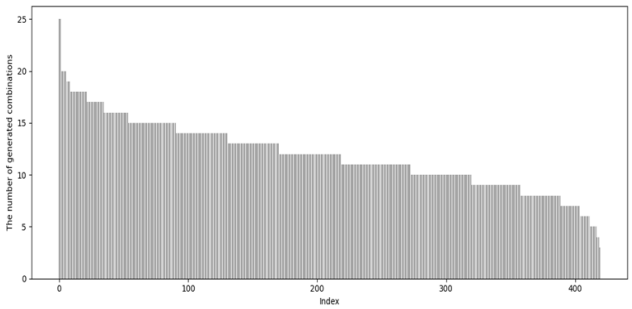
```
constraint test_cr {
   test_a dist {0:=5,1:=5};   // 2 cases
   test_b dist {[2:7]};       // 6 cases
   test_c dist {[8:12]};      // 5 cases
   test_d dist {[13:19]};     // 7 cases
}
```

```
covergroup cg_test;
   option.per_instance = 1;
   option.name = "test_cov";

   cp_test_a : coverpoint test_a { bins b[] = {[0:1]}; }
   cp_test_b : coverpoint test_b { bins b[] = {[2:7]}; }
   cp_test_c : coverpoint test_c { bins b[] = {[8:12]}; }
   cp_test_d : coverpoint test_d { bins b[] = {[13:19]}; }

   cross_test_abcd : cross cp_test_a, cp_test_b, cp_test_c, cp_test_d;
endgroup
```

(a)



(b)

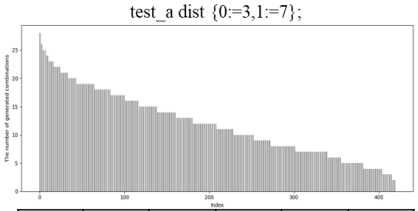
index	test_a	test_b	test_c	test_d	count
0	1	2	12	14	25
1	0	4	12	15	25
2	0	4	9	19	20
417	0	7	8	18	4
418	1	2	8	14	4
419	0	3	10	18	3

(c)

Figure 1. Results of 5,000 randomizations when the weighted distribution is uniform.

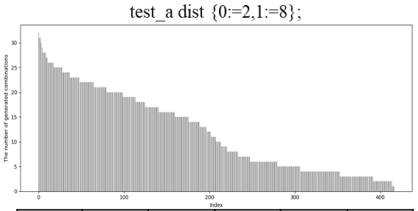
What would happen if one of the four variables had a different weighted distribution among them? Figure 2 illustrates how the min and max values of the frequency distribution of cross coverage combinations vary as the distribution of test_a changes. It can be observed that as the imbalance in the values of test_a increases, the difference between the min and max values of the frequency distribution also increases, and from a ratio of 2:8, it can be seen that certain combinations are no longer generated. Although not indicated in Figure 2, there were 3 combinations that were not generated for the 2:8 case, and 20 combinations for the 1:9 case.

test_a dist {0:=3,1:=7};



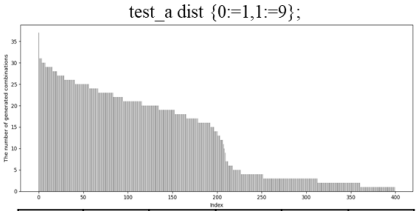
index	test_a	test_b	test_c	test_d	count
0	1	7	10	19	28
1	1	4	10	13	28
2	1	5	8	13	26
417	0	3	10	14	2
418	0	5	12	17	2
419	0	6	9	18	2

test_a dist {0:=2,1:=8};



index	test_a	test_b	test_c	test_d	count
0	1	3	9	13	32
1	1	6	9	15	31
2	1	5	10	15	31
417	0	5	8	15	0
418	0	7	10	17	0
419	0	7	12	14	0

test_a dist {0:=1,1:=9};



index	test_a	test_b	test_c	test_d	count
0	1	5	11	16	37
1	1	2	10	19	31
2	1	7	9	13	31
417	0	7	9	17	0
418	0	7	9	19	0
419	0	7	11	18	0

Figure 2. Results of 5,000 randomizations when the weighted distribution is non-uniform.

From these results, we can see that even if the probabilities of the values that the random variable can take are made equal, a very large number of repetitions are required for the mathematical probability intended by the engineer to be achieved. If the weighted distribution is not even, an even larger number of repetitions may be required. However, biased weighted distribution can be intentional on the part of the engineer or occur unintentionally due to complex random constraints. These results help us understand why simple repetition of simulations does not fill the cross coverage bins well, and these causes ultimately make it difficult to achieve functional coverage closure.

III. SUGGESTED APPROACH WITH EXAMPLE

The main idea of this study originated from the observation that cross coverage cannot be easily achieved when measuring functional coverage, but individual coverpoints can be easily filled, as mentioned earlier. Cross coverage is made up of a combination of coverpoints, each of which is composed of expressions or variables to be covered. If the coverpoints are guaranteed to be composed of random variables, and the cross coverage is composed of these coverpoints, then the cross coverage can be controlled through the random variables. The relationship between them will be best known by the DV engineer who created them, but if a third party can predict these relationships without any help from the DV engineer, they can fill the cross coverage hole instead of the DV engineer.

For this purpose, the coverage agent was developed as a third party, leveraging machine learning algorithms to predict the relationship between cross coverage and constraint random variable, select the test, and regenerate the random variable to fill the coverage hole.

In this section, we illustrate the key concepts and techniques of the proposed approach using a simple constrained random variable and functional coverage model. The example code is further detailed in Figure 3. This helps to clarify the theoretical basis and practical application of the proposed approach.

```
constraint test_cr {
   test_on inside {0,1};
   test_x dist{0:=1, [1:20]:=3, [21:30]:=3, [31:40]:=2, 41:=1};
   test_y dist{0:=1, [1:20]:=4, [21:30]:=4, 31:=1};

   test_mode0 inside {[0:8]};
   test_mode1 inside {[0:3]};
   test_en inside {[0:1]};
   test_bypass inside {[0:1]};
   test_mux inside {[0:4]};

   dummy0 inside {'h20,'h21,'h22,'h23,'h24};
   dummy1 inside {'h0,'h1,'h2,'h3};
   dummy2 inside {'h4,'h5,'h6,'h7};
   dummy3 inside {'h8,'h9,'hA,'hB};
   dummy4 inside {'hc,'hd,'hE,'hF};
}
```

(a)

```
covergroup cg_test;
   option.per_instance = 1;
   option.name = "test_cov";

   cp_test_on       : coverpoint test_on  { bins b[] = {[0:1]}; }
   cp_test_x        : coverpoint test_x   { bins b[] = {[0:41]}; }
   cp_test_y        : coverpoint test_y   { bins b[] = {[0:31]}; }

   cp_test_en       : coverpoint test_en       { bins b[] = {[0:1]}; }
   cp_test_mux      : coverpoint test_mux      { bins b[] = {[0:4]}; }
   cp_test_mode0    : coverpoint test_mode0    { bins b[] = {[0:8]}; }
   cp_test_mode1    : coverpoint test_mode1    { bins b[] = {[0:3]}; }
   cp_test_bypass   : coverpoint test_bypass   { bins b[] = {[0:1]}; }

   cross_test_xy    : cross cp_test_on, cp_test_x, cp_test_y;
   cross_test_mode  : cross cp_test_en, cp_test_mux, cp_test_mode0, cp_test_mode1, cp_test_bypass;
endgroup
```

(b)

Figure 3. Simple Constraint Random Variable and Functional Coverage Example

A. Example Code Analysis

In Figure 3(a), there are a total of 13 random variables, among which test_x and test_y have a non-uniform distribution. Figure 3(b) assumes a combination of IP behaviors controlled by these random variables and includes functional coverage code that covers them. There are a total of 8 coverpoints and 2 cross coverages declared. The variables starting with the name "dummy" are not used in the coverage model.

Table 1(a) shows the changes in coverage rate for cross coverage when running the example code with 1000 iterations of regression. As explained in Section II, even though the cross_test_mode has a uniform distribution, it requires about 6000 to 7000 tests to fill 720 bins. In the case of the cross_test_xy with a non-uniform distribution, even with 10,000 tests executed, it does not fill all 2,688 bins. Although not shown in the table, when cross_test_xy was run continuously without adjusting the weighted distribution of constraints or setting the directed value, it reached 99.74% (2681/2688) after about 25,000 tests.

On the other hand, Table 1 (b) shows the coverage rate for the coverpoints of the first regression run. Unlike cross coverage, it can be seen that cp_test_x and cp_test_y, which have a biased weighted distribution, also fill up easily like other coverpoints.

The accumulated number of tests	cross_test_xy		cross_test_mode	
	Coverage rate (%)	Touched bins / Total bins	Coverage rate (%)	Touched bins / Total bins
1000	30.80	828 / 2688	75.42	543 / 720
2000	50.74	1364 / 2688	94.31	679 / 720
3000	65.36	1757 / 2688	98.33	708 / 720
4000	75.33	2025 / 2688	99.03	713 / 720
5000	81.58	2193 / 2688	99.72	718 / 720
6000	85.86%	2308 / 2688	99.86	719 / 720
7000	88.99%	2392 / 2688	100.00	720 / 720
8000	91.11%	2449 / 2688	100.00	720 / 720
9000	92.67%	2491 / 2688	100.00	720 / 720
10000	94.12%	2530 / 2688	100.00	720 / 720

(a)

Coverpoint name	Coverage rate (%)	Touched bins / Total bins
cp_test_on	100.00	2 / 2
cp_test_x	100.00	42 / 42
cp_test_y	100.00	32 / 32
cp_test_mode0	100.00	9 / 9
cp_test_mode1	100.00	4 / 4
cp_test_en	100.00	2 / 2
cp_test_bypass	100.00	2 / 2
cp_test_mux	100.00	5 / 5

(b)

Table 1. The coverage results after running the example code with regression

B. Type of Inputs for Prediction

Figure 4 (a) simply shows the overall workflow of the suggested approach and the inputs used in each step. In this paper, the Verisium Manager, hereafter vManager, was used to run regression tests, and the Integrated Metrics Center, hereafter IMC, was used to merge coverage results and generate a report.

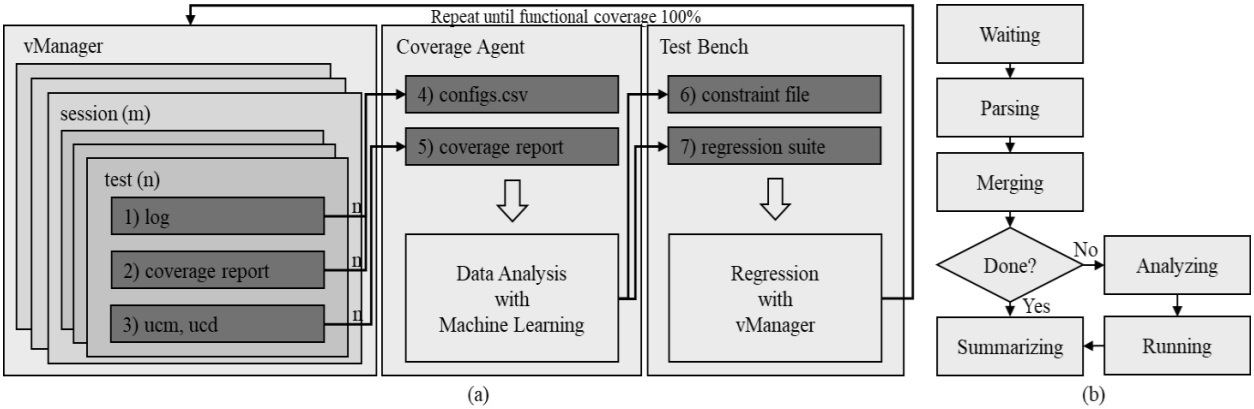


Figure 4. The suggested approach overview and coverage agent workflow

After the initial regression run, 1) log, 2) coverage report and 3) coverage database for each test, are used as input for the coverage agent. The log contains information about constraint random variables, as illustrated in Table 2(a), which were printed using the uvm_field_* macros within the uvm_object_utils macro block in UVM, based on the example code from Figure 3. After registering the random variables with the UVM macro, the variables are printed in a single line as shown in Table 2(b) to make them easier to parse. In addition, to ensure that all variables are printed without omission, we assign -1 to knobs.begin_elements.

```
`uvm_object_utils_begin(test_cfg_c)
    `uvm_field_int(test_on, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_x, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_y, UVM_ALL_ON + UVM_DEC)

    `uvm_field_int(test_en, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_mux, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_mode0, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_mode1, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(test_bypass, UVM_ALL_ON + UVM_DEC)

    `uvm_field_int(dummy0, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(dummy1, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(dummy2, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(dummy3, UVM_ALL_ON + UVM_DEC)
    `uvm_field_int(dummy4, UVM_ALL_ON + UVM_DEC)
`uvm_object_utils_end
```

(a)

```
uvm_default_printer = uvm_default_line_printer;
uvm_default_printer.knobs.begin_elements=-1;
```

(b)

Table 2. UVM code snippets for printing random variables in the log

The coverage report includes the functional coverage measurement results for each test, which were created by the IMC. For instance, Table 3 excerps a part of one of the coverage reports generated for each test using the example code from Figure 3. In the coverage report, we can see the name of the cross coverage, the total number of bins, and the current number of touched bins. We can also determine the number of coverpoints that make up the cross coverage. In this example, the cross coverage named cross_test_xy consists of a total of 2688 bins, with only 1 bin being touched. By examining the bin b[*],b[*],b[*] of cross_test_xy, we can confirm that it is composed of three coverpoints. This means the three random variables are correlated. This information is crucial in determining the number of random variables needed to fill in the cross coverage through machine learning algorithms.

test_cov	21.18%, 0.29% (10/3506)	36 (test_cov.sv) covergroup cg_test;
|--cross_test_xy	0.04% (1/2688)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;
| |--b[0],b[0],b[0]	100.00% (1/1)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;
| |--b[0],b[0],b[1]	0.00% (0/1)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;
| |--b[0],b[0],b[2]	0.00% (0/1)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;
| |--b[0],b[0],b[3]	0.00% (0/1)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;
| |--b[0],b[0],b[4]	0.00% (0/1)	50 (test_cov.sv) cross_test_xy : cross cp_test_on, cp_test_x, cp_test_y;

Table 3. The coverage report example

Finally, the coverage databases of each test are used to generate the overall coverage report. The coverage agent monitors the vManager's operation and, upon completion of a session, merges the log and coverage reports of each test into a CSV file named 4) configs.csv, a tabular data structure in Python, and subsequently merges the coverage databases of each test to generate the 5) overall coverage report. These two files serve as the basis for machine learning-based data analysis in the next steps.

C. Decision Tree Algorithm

Machine learning algorithms are used to predict which random variables should be controlled to fill the hole in a certain cross coverage. The selection of a machine learning algorithm is crucial due to its major characteristics affecting its performance and suitability for specific tasks. The decision tree is used in this study because it is one of the white-box machine learning models [5]. It allows us to understand how each feature contributes to the prediction and how important they are in determining the prediction.

In this paper, the features correspond to random variables, and the target that the model needs to classify are the numbers that are assigned to each cross coverage bin. Machine learning will predict the number assigned to each cross coverage bin for classification, as shown in Table 4. The numbers assigned to the cross coverage can also have a value of 0, indicating that no bins of the cross coverage were touched in the corresponding test.

cross_test_xy	numbering	cross_test_mode	numbering
b[0],b[0],b[0]	1	b[0],b[0],b[0],b[0],b[0]	1
b[0],b[0],b[1]	2	b[0],b[0],b[0],b[0],b[1]	2
b[0],b[0],b[2]	3	b[0],b[0],b[0],b[0],b[2]	3
b[1],b[41],b[29]	2677	b[8],b[3],b[1],b[1],b[2]	718
b[1],b[41],b[30]	2678	b[8],b[3],b[1],b[1],b[3]	719
b[1],b[41],b[31]	2688	b[8],b[3],b[1],b[1],b[4]	720

Table 4. The machine learning prediction results according to the number of regression iterations

The main focus here should not be on whether the machine learning correctly classified the bin numbers, but rather on identifying the features that contributed the most to the classification task. The features with high contributions to the classification are the random variables that are highly correlated with the cross coverage, and they are sorted in order of contribution and stored in a list format.

Table 5 shows the results of predicting which random variables are highly correlated with each cross coverage using a coverage agent that applies detailed descriptions of the example code in Figure 3 after running it for each regression iteration. As the regression iteration progresses, the coverage agent identifies the top 3 and top 5 random variables

associated with each cross coverage for cross_test_xy and cross_test_mode respectively. This approach is based on the predetermined number of random variables required for each cross coverage, as outlined in the coverage report. To gradually observe the process, the coverage agent limited the maximum number of tests that can be configured for a regression suite to 1000. In the predicted results, random variables that are not correlated with cross coverage have been marked in color.

First, looking at cross_test_xy, we can see that in the first iteration, test_x and test_y were correct, but the unrelated dummy1 was predicted. In the next iteration, dummy1 disappears, but another unrelated test_mode0 is predicted.

And by the third iteration, it shows that all three random variables related to cross_test_xy are found correctly. Since the prediction was successful, if the number of tests configuring the regression was not limited, cross_test_xy would have been covered in the third iteration.

However, since the number of bins that were not covered was greater than the number of tests, the next iteration was carried out. An interesting fact is that in the next iteration, it was wrong again. This change is analyzed to be due to the fact that the input features (random variables) and target (cross coverage numbering) values of the machine learning algorithm are selected randomly every iteration. Cumulatively, 80% of the data is used for training, while 20% is randomly selected for evaluating the model's accuracy. However, as previously mentioned, our focus is not on the accuracy of the prediction but rather on identifying the features that were considered crucial in arriving at the correct result.

In the end, the prediction was successful again in the next iteration, and the regression for cross_test_xy was completed as the cross coverage was filled.

Next, for cross_test_mode, it succeeded in the prediction in the second iteration, and since the number of cross coverage bins that were not filled was smaller than the limited number of tests, it was covered and terminated.

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
cross_test_xy	test_cfg.test_x	test_cfg.test_y	test_cfg.test_y	test_cfg.test_y	test_cfg.test_x
	test_cfg.test_y	test_cfg.test_x	test_cfg.test_x	test_cfg.test_x	test_cfg.test_y
	test_cfg.dummy1	test_cfg.test_mode0	test_cfg.test_on	test_cfg.test_mode0	test_cfg.test_on
cross_test_mode	test_cfg.test_mode0	test_cfg.test_mux			
	test_cfg.test_mux	test_cfg.test_mode0			
	test_cfg.test_x	test_cfg.test_en			
	test_cfg.test_y	test_cfg.test_bypass			
	test_cfg.test_en	test_cfg.test_mode1			

Table 5. The machine learning prediction results according to the number of regression iterations

In the simple example code, the coverage agent found the random variable related to each cross coverage out of 13 random variables. However, in the case of the IP used in this paper, about 7,000 random variables were used for verification, and among them, the coverage agent needs to find the random variable related to each cross coverage.

D. Utilizing Machine Learning Predictions

The decision tree algorithm, one of the machine learning algorithms, has been explained in detail, including how it predicts and what it predicts. It is important to determine how to utilize these predictions. The random variables predicted by the machine learning are used to calculate their product sets. The values of the random variables will be the unique values of the values that have been obtained so far while running the tests.

For example, in the case of cross_test_xy, Table 6 shows the results of calculating the product set when the coverage agent successfully predicted in the third iteration of Table 5. In cross_test_xy, the total number of bins matches the number of combinations, and in the table, we can see that each combination is being counted to see if it has been generated before. Based on these result, the coverage agent reflects the combinations that have not been generated in the next regression.

This approach is aggressive because it's an all-or-nothing strategy. If the prediction is successful, it will quickly fill the bin. However, if the prediction fails, the value will be entered into a combination of unrelated random variables. Despite this disadvantage, the meaning of regression offsets it by testing various combinations. Even if it fails, it's like running the regression as usual. Additionally, it's even luckier than the usual regression, where other cross coverages

may be touched. This is the primary reason why machine learning with uncertainty in prediction can be applied to coverage for verification.

This method assumes that all possible values of the random variable should be generated at least once, and that the coverpoint coverage is easily filled. Therefore, this method does not consider cases where a coverpoint never fills up. If the random variable is not uniform and the coverpoint fills late, the performance of the coverage agent will be affected, but ultimately it will fill the cross coverage hole.

Index	test_cfg.test_y	test_cfg.test_x	test_cfg.test_on	Count
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
2685	31	40	1	1
2686	31	41	0	0
2687	31	41	1	0

Table 6. Random variable product set for cross coverage (cross_test_xy)

In the case where the values of the random variables created based on the product set are combinations that cannot be produced by the constraint, what happens? Since the coverage agent does not know the constraint of the random variable, if it puts the unilateral combination of the number of cases for the next regression, randomization can fail. To eliminate this situation, all values of the random variables created by the coverage agent are declared as soft constraints, as shown in Table 7. The values will be assigned to each test using the randomize() with constraint block.

```
//for index 0 in the Table 6.
soft test_cfg.test_y == 0;  soft test_cfg.test_x == 0;  soft test_cfg.test_on == 0;

//for index 2687 in the Table 6.
soft test_cfg.test_y == 31; soft test_cfg.test_x == 41; soft test_cfg.test_on == 1;
```

Table 7. The random variables are declared as "soft"

Therefore, if an invalid combination is assigned, it will be ignored and the random variables will be generated by randomizing them as usual without fail. In this case, it will simply become one of the regression tests and will be used as data for the next prediction.

E. Coverage Agent

The coverage agent, as mentioned earlier, is developed using Python to process files, generate constraint variables using machine learning, select tests, and create a new regression suite. Figure 4(b) simply displays the workflow of the coverage agent. After waiting for the vManager to complete regression session, it parses the aforementioned files, merges them, and checks if the functional coverage is 100% through the overall coverage report. If it is not 100%, it analyzes the coverage, creates a regression suite, and then runs the regression again.

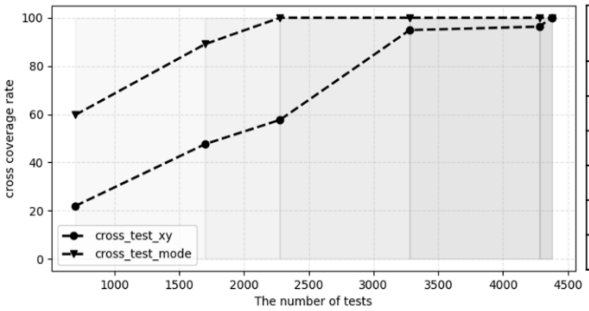
The regression is executed in batch mode, and then the subsequent step, summarizing, is carried out. In this step, the information about which cross coverage remains, how much it is filled, and which random variables are predicted to be related to the cross coverage is summarized and saved as a file. These files allow DV engineer to check the current functional coverage status and ensure that the process is running properly.

Figure 5 below shows the results of applying the coverage agent to the example code in Figure 3, both graphically and in table form. A total of 4,378 tests were run, with a maximum of 1,000 tests per iteration, and 5 regression suites were executed. After the initial 700 test regression, the coverage agent was applied.

In iteration 3, we can observe a significant increase in cross_test_xy from 57.66% to 94.78% due to successful predictions as shown in Table 5. However, since the number of test samples per regression is limited to 1000, it does not reach 100%. In iteration 5, it successfully predicts again and shows that the coverage rate of cross_test_xy reaches 100%.

In iteration 4, the prediction failed, but as mentioned earlier, it is similar to performing a normal regression, so we can see that the numbers increase. These accumulated 4279 tests help the coverage agent make more accurate predictions for the next iteration 5 regression.

In the case of the cross_test_mode, it succeeded in predicting in the second iteration and immediately confirmed that all coverage bins were covered.



Iteration	The accumulated number of tests	Number of tests by coverage agent	cross_text_xy (%)	cross_test_mode (%)
0	700	-	21.95	59.72
1	1700	1000	47.58	89.03
2	2279	579	57.66	100
3	3279	1000	94.87	100
4	4279	1000	96.32	100
5	4378	99	100	100

Figure 5. The results of applying the coverage agent to the example code in Figure 3.

F. Various Strategies

The results in Figure 5 show the ideal behavior of the method proposed in this paper. It seems as if having more data to learn will always lead to better predictions. However, in reality, this is not always the case.

In theory, when predictions fail and regression is executed, the test information from the regression is used as training data to improve the model's prediction accuracy in the next step. In other words, cross coverage that continues to miss will gradually increase its probability of being correct as it progresses to the next step. In practice, however, the situation is not as straightforward. The accumulation of training data may sometimes lead the model to reinforce incorrect predictions rather than correcting them.

At this point, we describe the various cases encountered while applying this approach and explain the strategies that were introduced to address them.

Case 1: when the model's prediction list is shorter than the number of random variables related to cross coverage. The solution is simple. Since this is due to insufficient information, we list up the tests that have touched at least one cross coverage, and then randomly select them and put them back into the regression suite.

For instance, in Table 5, the cross_test_mode requires identifying 5 random variables. However, if there is a lack of training data, meaning an insufficient number of tests, it is possible that the number of random variables identified through machine learning algorithms may be inadequate. In such situations, when the training data is insufficient, the coverage agent searches for scenarios related to the targeted cross coverage and increases the number of tests for those specific scenarios.

Case 2: when the model makes an incorrect prediction and generates a product set of completely unrelated constraint variables, by chance all possible cases of these variables are covered.

For example, in Table 5, there is a higher likelihood that the prediction will be inaccurate due to fewer test repetitions performed during iteration 1. The provided code example was straightforward, so it quickly accesses test_x and test_y, which are highly correlated with cross_test_xy.

cross coverage	Iteration 1
cross_test_xy	test_cfg.test_on
	test_cfg.test_bypass
	test_cfg.test_mux

(a)

Index	test_cfg.test_on	test_cfg.test_bypass	test_cfg.test_mux	Count
0	0	0	0	29
1	0	0	1	27
2	0	0	2	28
17	1	1	2	40
18	1	1	3	23
19	1	1	4	33

(b)

Table 8. One of the worst-case scenarios in the event of a prediction failure

However, in reality, finding the correct answer among numerous random variables is not an easy task. In the provided code example, let's assume that the cross_test_xy prediction at iteration 1 is as shown in Table 8 (a), even though this scenario is unlikely due to its simplicity, but for the sake of explanation.

Then we represented the product set of these three random variables in Table 8 (b). Looking at the table, you can see that all possible cases have been touched. The number of possible cases for test_on, test_bypass, and test_mux are 2, 2, and 5, respectively, resulting in a total of 20 possible cases. When all these cases are covered, the coverage agent does not assign a value to the random variable.

To prevent this, it selects the tests as like case 1 and adds them to the regression suite to prevent incorrect predictions due to lack of learning data.

Case 3: Nevertheless, the incorrect random variables continue to be at the top of the contribution. To tackle this problem, the number of product sets is compared. Cross coverage is composed of the product set of coverpoints, and in some cases, illegal bins or ignore bins may be used. Therefore, the number of product set of random variables must be equal to or less than the actual number of cross coverage bins that need to be filled.

If the product set of the predicted variables is greater than the number of bins, it is an incorrect combination. In this case, it will be passed to the next combination in the candidate list until a combination that meets the condition is found.

For example, if the total number of bins for cross_test_xy is 2668, and the number of predicted cases is less than 2668, then it will not be the correct answer.

Case 4: However, an incorrect prediction combination may continue to occur even though the number of product sets is satisfied.

Table 9 shows the random variables sorted by their contribution in Table 5, which contains twice as many variables as required for cross_test_xy. This is referred to as the candidate list. The columns related to cross_test_xy, test_x, test_y, and test_on are highlighted.

As shown in Table 9, test_on starts appearing from iteration 3. And in iteration 4, when the prediction fails again, test_on is not in the top 3, but it is included in the candidate list.

When applying the coverage agent to practical tasks, even though the data to be learned accumulates, there may be cases where incorrect predictions are made for some cases. However, in all the cases we encountered, the correct answer remained in the candidate list, even though the prediction was incorrect, as shown in iteration 4 of Table 9.

cross_coverage	Index	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
cross_test_xy	0	test_cfg.test_x	test_cfg.test_y	test_cfg.test_y	test_cfg.test_y	test_cfg.test_x
	1	test_cfg.test_y	test_cfg.test_x	test_cfg.test_x	test_cfg.test_x	test_cfg.test_y
	2	test_cfg.dummy1	test_cfg.test_mode0	test_cfg.test_on	test_cfg.test_mode0	test_cfg.test_on
	3	test_cfg.dummy3	test_cfg.test_mode0	test_cfg.test_mode0	test_cfg.dummy4	test_cfg.test_mode0
	4	test_cfg.test_mux	test_cfg.test_mode1	test_cfg.test_mode1	test_cfg.test_on	test_cfg.dummy3
	5	test_cfg.dummy4	test_cfg.test_mux	test_cfg.test_mux	test_cfg.dummy3	test_cfg.test_mux

Table 9. Random variable candidate list for cross_test_xy

In such case, if the cross coverage rate does not change for two regression cycles, the combination is no longer selected, allowing the next combination to be explored.

Case 5: Random variables may have the same value but different names for the convenience of writing constraint expressions in the testbench. They can occur when they are ranked high together on the candidate list. These can interfere with the prediction combination, so it is necessary to make them so that only one remains in the candidate list.

As described above, various strategies enable the coverage agent to select diverse combinations within the candidate list. This is the reason why we insert twice the number of predictions into the candidate list.

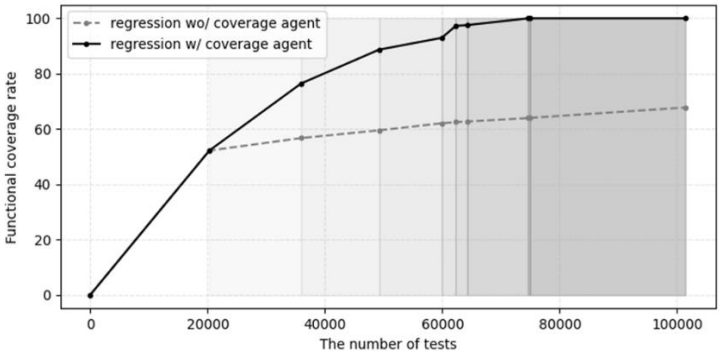
IV. RESULT

Figure 6 compares the results of the original regression, where the new approach was not applied, with the results of the regression using the coverage agent, in terms of functional coverage rate. During the initial regression, 20,290 test cases were executed, and the coverage agent was activated from the next regression.

The table shows the exact numbers, and the second column indicates the number of tests created by the coverage agent for each regression suite. For comparison, the functional coverage rate of the original regression was calculated in proportion to the number of tests created by the coverage agent.

The second row of the second column represents the initial regression, which is not a regression suite generated by the coverage agent, and the last row is also a regression run to compare the figures with the original regression, so it is processed with a dash.

After a total of 74,956 tests, the regression with the coverage agent reached a functional coverage rate of 100%. In contrast, the baseline regression fails to exceed 70% coverage rate, even after more than 100,000 runs, as shown in the last row in the table.



The accumulated number of tests	Number of tests by coverage agent	wo/ coverage agent (%)	w/ coverage agent(%)
20290	-	52.17	52.17
35929	15639	56.68	76.43
49210	13281	59.48	88.66
59896	10686	62.03	92.94
62301	2405	62.45	97.29
64189	1888	62.66	97.53
74585	10396	63.90	99.99
74956	371	63.98	100.00
101450	-	67.74	100.00

Figure 6. Comparison of functional coverage rates with and without coverage agent

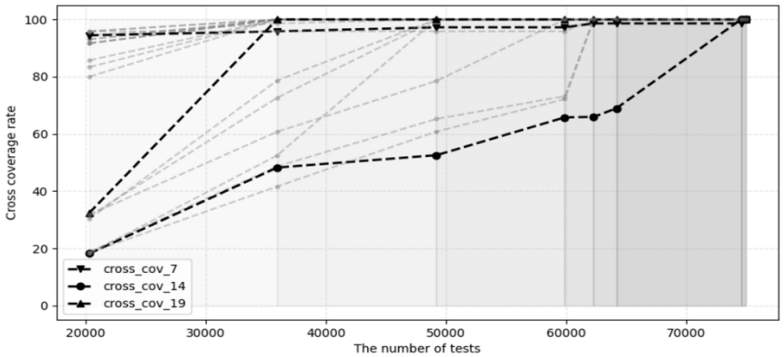
In this work, the IP used has 134 cross coverages and 14,616 values as cross coverage bins. After the first regression, 21 cross coverages were not filled, and the total count of their bin was 7,673.

Figure 7 shows the process of the remaining cross coverage rates being filled to 100% by the coverage agent, and only three representative cross coverages are shown among them.

The cross_cov_19 presents the most ideal case where it successfully predicts using only the information from the first regression and reaches to 100% in the next regression.

The cross_cov_14, on the other hand, continuously failed in predictions but gradually filled up as the number of regression tests increased, eventually succeeding in prediction and reaching 100%.

In the case of the cross_cov_7, it is the worst-case scenario where predictions continue to fail and the coverage rate either barely increases or doesn't increase at all, even with the number of regression tests, but it eventually reaches 100% due to the strategies mentioned earlier.



The accumulated number of tests	cross_cov _7(%)	cross_cov _14(%)	cross_cov _19(%)
20290	94.44	18.17	32.42
35929	95.83	48.23	100
49210	97.22	52.56	100
59896	97.22	65.77	100
62301	98.61	65.93	100
64189	98.61	68.92	100
74585	98.61	100	100
74956	100	100	100

Figure 7. The process of each cross coverage reaching 100%

V. CONCLUSION

In this paper, we propose a novel approach using machine learning for functional coverage closure. We define the factors of the functional coverage closure challenge as cross coverage holes, and to fill them, the coverage agent utilizes the output generated during the regression process, thereby eliminating the need for DV engineers to manually analyze the holes and modify the constraints of random variables. As a result, functional coverage closure can be achieved markedly faster than the original regression without any additional effort from DV engineers.

REFERENCES

[1] Caglayan Yalcin, et al., "An Analysis of Stimulus Techniques for Efficient Functional Coverage Closure" DVCON, 2021
[2] Azade Nazi, et al., "Adaptive Test Generation for Fast Functional Coverage Closure" DVCON, 2022
[3] Jakub Pluciński, et al., "Accelerate Functional Coverage Closure Using Machine-Learning-Based Test Selection" DVCON, 2023
[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language" IEEE 1800-2017
[5] https://scikit-learn.org/stable/modules/tree.html