PSS action sequence modeling using Machine Learning

Moonki Jang, Myeongwhan Hyun, Hyunkyu Ahn, Jiwoong Kim, Yunwhan Kim, Dongjoo Kim

Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18488,
Korea, moonki.jang@samsung.com, mywhan.hyun@samsung.com, hyunkyu3.ahn@samsung.com,
jiwoong7.kim@samsung.com, yunhkim@samsung.com, dj.num1.kim@samsung.com

***Abstract*- In general, one of the most difficult aspects of ML is collecting data for learning. This is because the more learning data, the higher the prediction accuracy of ML. In this regard, we expected that the combination of PSS, which can easily generate numerous tests, and ML, which finds regularity based on collected data, could exert fantastic synergy. And we were able to dramatically increase the verification coverage by being able to freely create concurrent function events that were previously considered impossible at the simulation level through PSS and ML**

I. Introduction

We have an experience in deadlock verification caused by a protocol conflict between PCIe and ACE interface at the pre-silicon level using actual PSS. [1][2] In order to generate deadlock, it was necessary to satisfy a very complex sequence condition in which write transactions of CPU and snoop transactions due to posted writes of PCIe must occur correctly in sequence while the PCIe RC buffer is overflowed. We were able to reproduce the deadlock condition after classifying each deadlock target condition by PSS action and generating scenarios with thousands of random sequence combinations.

However, even in this process, it was necessary for the engineer to continuously check and analyze the results to adjust the random constraints in order to reproduce the target condition, and a considerable amount of time and manpower were required. This is because, as shown in the figure below, we thought that we performed 3 functions with SW at the same time, but on the actual HW, the function was being performed at different times. At that time, we had no choice but to add configurable delay and compare results to ensure that each function was executed at the same time at the actual hw level.

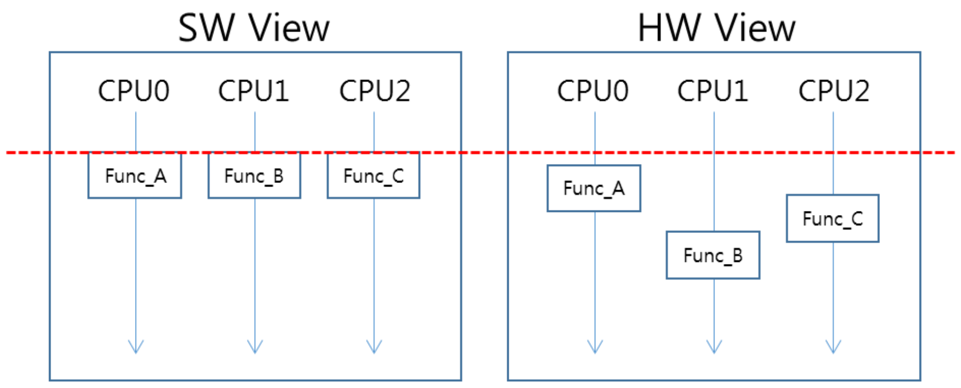


Figure 1. Software sequence vs actual hardware sequence

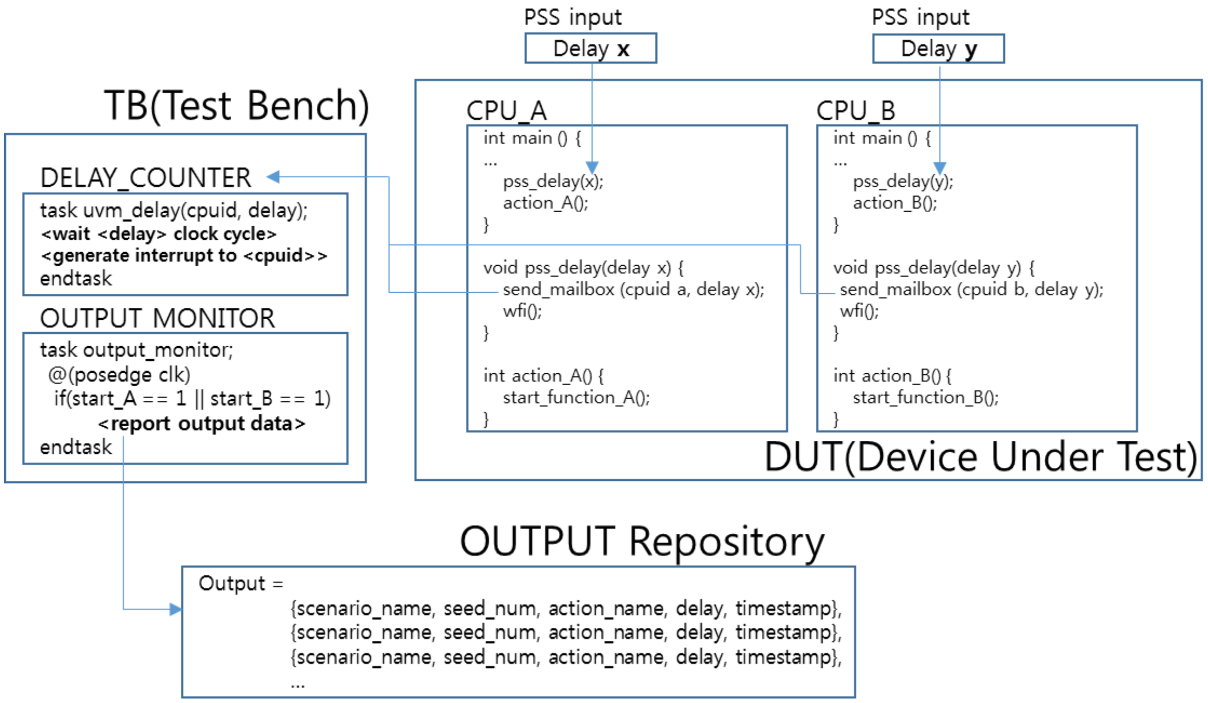
The starting point of this paper was the imagination that AI (Artificial Intelligence) could repeat itself and create an action that satisfies the target condition if only the target condition was specified for each PSS action without the need for an engineer to intervene to analyze and adjust the result

In this paper, we will explain how to apply ML (Machine Learning) to PSS with specific examples.

II. TB structure of PSS Action Sequence model

In general, fatal deadlock issues mainly found at the silicon-level occur in situations where certain conditions occur at the same time as above. And until now, the reproduction of the deadlock condition has tended to rely mainly on random tests, so an environment that can be tested for a long time at high speed was necessary, and it was thought that it was not suitable for verification at the simulation level. However, the simulation environment has the advantage of being able to monitor internal RTL signals that cannot be seen at the silicon level and perform hundreds of tests simultaneously in parallel. To utilize these advantages, we created a PSS action sequence model using configurable delay and signal monitor to enable precise action sequence control.

The figure below shows the structure of the PSS sequence model.



```
task uvm_delay(cpuid, delay);
<wait <delay> clock cycle>
<generate interrupt to <cpuid>>
endtask
```

```
task output_monitor;
 @(posedge clk)
  if(start_A == 1 || start_B == 1)
        <report output data>
endtask
```

```
int main () {
...
   pss_delay(x);
   action_A();
}

void pss_delay(delay x) {
   send_mailbox (cpuid a, delay x);
   wfi();
}

int action_A() {
   start_function_A();
}
```

```
int main () {
...
   pss_delay(y);
   action_B();
}

void pss_delay(delay y) {
 send_mailbox (cpuid b, delay y);
 wfi();
}

int action_B() {
   start_function_B();
}
```

```
Output =
          {scenario_name, seed_num, action_name, delay, timestamp},
          {scenario_name, seed_num, action_name, delay, timestamp},
          {scenario_name, seed_num, action_name, delay, timestamp},
          ...
```

Figure 2. Structure of PSS action sequence model

A. *PSS scenario generation*

The above figure shows how the C code generated through the PSS tool operates in CPU_A and CPU_B, respectively. The delay value, which is a configurable PSS input value, is added in front of the target action as a function called pss_delay. When generating a test in the PSS tool, this pss input value can have a specific value or a random value. You can also set the number of generated tests.

B. *Cycle delay counter*

For sophisticated action sequence control with cycle accuracy, we made and used a cycle delay counter that can adjust the delay in units of clock cycles in the TB. When pss_delay is executed in C code, the delay value and cpu id are stored in a specific memory area used as a mailbox. In the monitoring task of TB, the mailbox is checked every cycle. When a value is transmitted from pss_delay, the cycle delay counter starts to operate. The cycle delay counter counts the clock cycle as much as the delay value and generates an interrupt to the CPU to notify the end of the delay.

The important point here is that the mailbox is checked every cycle and an interrupt is generated at the end of the delay to eliminate the unpredictable cycle loss to create an action delay as much as the desired clock cycle.

C. *Output monitor*

When executing tests generated by PSS tool in simulation environment, include target output file describing target condition as command argument. As for the target condition described in the target output file, it is checked whether the target condition occurs every clock cycle in the output monitor of the TB as shown in the figure above. If the target condition occurs, the output monitor prints the test information, timestamp information, and action information that caused the target condition as a simulation log message. After the simulation is completed, the log analyzer extracts the output monitor message from each simulation log and stores it in the output repository.

D. *Output repository*

The output repository is a space where the information output by the output monitor is collected. The information collected here is used as training data for ML flow, which will be described below.

III. ML IMPLEMENTATION FOR PSS ACTION SEQUENCE MODELING

The purpose of the PSS action sequence model is to adjust the delay to equalize the output timestamp values of different actions. And for this, we applied the ML technique to the sequence modeling process.

The figure below shows the ML sequence modeling flow.

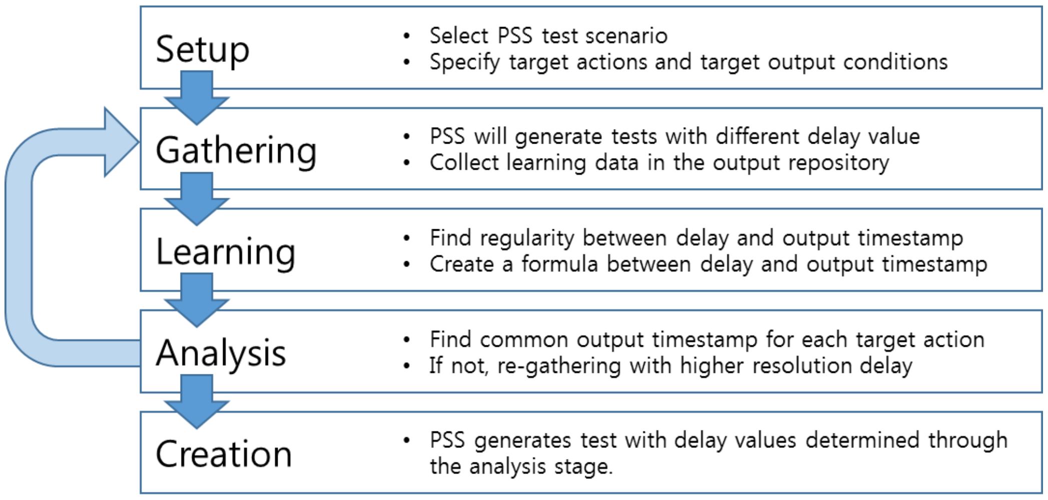

Setup
• Select PSS test scenario
• Specify target actions and target output conditions
Gathering
• PSS will generate tests with different delay value
• Collect learning data in the output repository
Learning
• Find regularity between delay and output timestamp
• Create a formula between delay and output timestamp
Analysis
• Find common output timestamp for each target action
• If not, re-gathering with higher resolution delay
Creation
• PSS generates test with delay values determined through the analysis stage.

Figure 3. ML sequence modeling flow

A. Setup/Gathering stage

In the setup stage, the PSS tool selects the scenario to be used for test creation and selects the target action to control the sequence. And set the pss_delay value to be applied to the test to be created. The settings related to these test generation are reflected when generating the test through the batch mode script supported by the PSS tool.

And the target condition to be used in the output monitor is also selected here. Target condition is managed as a separate file, and when simulation is executed, it is transmitted to testbench through cmd argument and applied to simulation.

In the Gathering stage, the PSS tool creates tests and performs simulations according to configured batch mode script. After running the generated tests, store the output monitor's result report in the output repository.

B. Learning stage

ML engine analyzes the results stored in the Output repository to determine how the delay increase affected the output condition occurrence time and to find the regularity between the two. This process is implemented as a simple ML linear regression algorithm. From the data collected by the output monitor, we can create coordinates with delay and timestamp values.

The table below shows the delay values of Action A and Action B and the timestamp where the actual target condition occurred in the form of coordinates.

TABLE I
EXAMPLE OF GENERATED COORDINATES

Action A	Action B
(0, 1)	(0, 2)
(1, 3)	(1, 5)
(2, 5)	(2, 8)

In the learning stage, the goal is to obtain the following linear equation by identifying the tendency between each coordinate.

$$\boxed{ax + b = y, \qquad a \text{ (slope)} = \text{increment of timestamp / increment of delay}}$$

Figure 4. Linear equation

Accordingly, the linear equation representing the correlation between delay and timestamp for Action_A and Action_B is obtained as follows.

Action_A : $2X_1 + 1 = Y$ (X_1,X_2: delay)
Action_B : $3X_2 + 2 = Y$ (Y : Output timestamp)

Figure 5. Learning result

In the process of linear regression analysis, you can simply obtain a linear equation from the learning data using a python library called SciPy[3] as shown below.

```
from scipy import stats
  x = [x1, x2, x3, x4, x5, x6, x7,...]
  y = [y1, y2, y3, y4, y5, y6, y7,...]
 slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
```

Figure 6. linregress API of SciPy

C. Analysis stage

In the analysis stage, the stage agent uses the formula created in the learning stage to find the delay value where the output condition of the target actions occurs simultaneously. The equation in figure 5 obtained in the learning stage is transformed as follows.

$$2a + 1 = 3b + 2$$

(a : delay of Action_A, b: delay of Acion_B)

Figure 7. Equation for finding common timestamp value

This equation is substituted with a Linear Diophantine equation[4] of the form $ax+by=c$, and the greatest common divisor and solution of the equation are found using the Extended Euclidean algorithm[5] as shown below. However, if c is not a multiple of the greatest common divisor of a and b, this equation cannot be solved according to Bézout's Identity [6].

Diophantine equation : $ax + by = c$

(Euclidean algorithm)

$a = bq_0+r_1$ (q_x : quotient(=a/b) ,r_x : remainder(=a%b))
$b = r_1q_1+r_2$
$r1 = r_2q_2+r_3$
...
$r_{i-1} = r_iq_i+r_{i+1}$ *(When r_{i+1} is 0, the algorithm terminates and r_i becomes GCD(greatest common divisor))*
-> $r_{i+1} = r_{i-1} - r_iq_i$ (1)

(Extended Euclidean algorithm)

In this case, if the coefficient of a is s_i and the coefficient of b is t_i for any r_i, it can be expressed as follows.

$$r_i=s_ia+t_ib$$

Substituting this into (1), we get the following equation.

$$\begin{aligned} s_{i+1}a+t_{i+1}b&=(s_{i-1}a+t_{i-1}b)-(s_ia+t_ib)q_i \\ &=s_{i-1}a-s_iaq_i+t_{i-1}b-t_ibq_i \\ &=(s_{i-1}-s_iq_i)a+(t_{i-1}-t_iq_i)b \end{aligned}$$

s and t are obtained as follows by increasing i until r_{i+1} becomes 0.

$$r_0=a,\ r_1=b,\ s_0=1,\ s_1=0,\ t_0=0,\ t_1=1$$
$$r_{i+1} = r_{i-1} - r_iq_i$$
$$s_{i+1}=s_{i-1}-s_iq_i$$
$$t_{i+1}=t_{i-1}-t_iq_i$$

Figure 8. Extended Euclidean algorithm for Diophantine equation

The Extended Euclidean algorithm finds the greatest common divisors of *a* and *b*, as well as *x* and *y* values that satisfy the equation. The solution obtained in this way is called a particular solution, and using this, a general solution can be obtained that can find numerous combinations of *x* and *y* depending on the integer *k* value.

- Particular solution (d : GCD of a and b)

$$X_0 = s \times c/d$$
$$Y_0 = t \times c/d$$

- General solution (k : integer value)

$$x = x_0 + k * b/d$$
$$y = y_0 - k * a/d$$

Figure 9. Result of Extended Euclidean algorithm

The formula in figure7 can be expressed as $2a\text{-}3b=1$ as shown below. Using the Extended Euclidean algorithm, a particular solution of $a_0=-1$, $b_0=-1$ can be obtained, and a general solution can be obtained using these values.

$2a - 3b = 1$

(Euclidean algorithm)
$3 = 2*1 + 1$
$2 = 1*2 + 0$

(Extended Euclidean algorithm)
$1 = -3*(-1) + 2*(-1)$

-Particular solution :

$$a_0 = -1,\ b_0 = -1$$

-General solution :

$$a = a_0 + k*((-3)/1) = -1 -3k$$
$$b = b_0 - k*(2/1) = -1 - 2k$$

Figure 10. Result of analysis stage

By changing the k value of this general solution, we can obtain numerous combinations of *a* and *b* values that satisfy $2a - 3b = 1$. That is, it is possible to find the pss_delay value that causes Action_A and Action_B to have the same timestamp value using the Extended Euclidean algorithm.

(k,a,b) = (0,-1,-1), (-1,2,1), (-2,5,3), (-3,8,5), (-4,11,7), ...

Figure 11. Result of generation solution

The code implementing the Extended Euclidean algorithm using Python is as follows. Input the values of *a*, *b,* and *c* of $ax+by=c$ and output a particular solution.

```
def diophantine(a, b, c):
    '''returns (x, y) where ax+by=c'''
    r1, r2 = a, b
    s1, s2 = 1, 0
    t1, t2 = 0, 1

    while r2!=0:
        q = r1/r2
        r1, r2 = r2, r1%r2
        s1, s2 = s2, s1-q*s2
        t1, t2 = t2, t1-q*t2
    # gcd: r1
    return (c/r1*s1, c/r1*t1)
```

Figure 12. Extended Euclidean algorithm code

If the expected delay values are obtained through the above processes, repeat the learning data gathering and learning stage based on the values to check whether the previously obtained linear equation is the same and applied, and check whether the same timestamp is obtained as predicted. In the case of the ML linear regression algorithm, since it is important to maintain the linearity of the learning data, a system Verilog-based cycle delay counter is used without any SW loop delay that can hinder this. And even when the delay is completed, it is implemented to notify the completion of the delay with an interrupt that consumes a certain cycle.

D. Creation stage

In the creation stage, the PSS tool will generate a test using the delay prediction results whose linearity has been verified in the analysis stage. In the test generated in the creation stage, the target conditions will be triggered simultaneously by the machine-learned delay value.

IV. CASE STUDY: REPRODUCE PCIE DEADLOCK CONDITIONS

As mentioned earlier, we have experience in PCIE deadlock verification using PSS. In this chapter, we will show how easy it is to reproduce PCIE deadlock conditions using the ML sequence model.

PCIE deadlock is caused by protocol conflict between PCIE and ACE bus interconnector. As shown in the figure below, when the PCIE RC buffer becomes full due to numerous non-posted requests from the CPU, a new non-posted request cannot be received until the previously posted write is completed. And if the CPU continues to issue non-posted requests to PCIe while the PCIE RC buffer is full, the write channel of the bus will be saturated and no more write transactions can be processed. And the posted write transaction delivered as WriteUnique type from PCIE will not be processed until the write transaction generated by the CPU is completed due to the non-blocking requirements defined in the ACE protocol.[7] Eventually, the CPU waits for the PCIE RC buffer to be empty, and PCIE waits for the CPU's Writeback transaction to be completed. We call this a deadlock chain

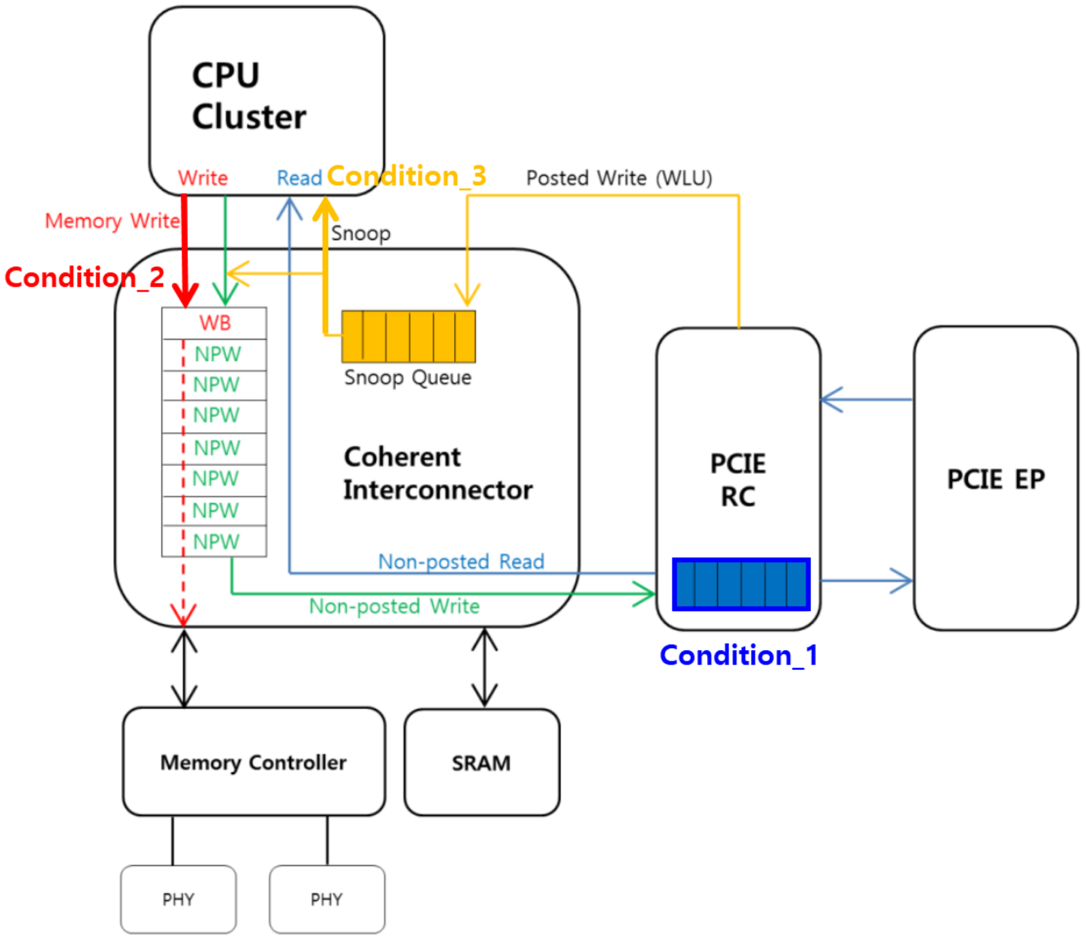


Figure 13. PCIE deadlock flow diagram

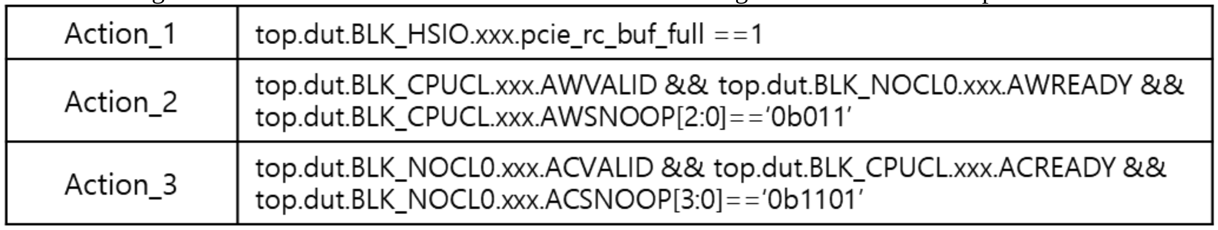
PCIE deadlock appears when the following conditions occur simultaneously as shown in the figure above.

1) Condition_1: Generate PCIE RC buffer full (generated by Action_1)
2) Condition_2: Generate Writeback transaction (Action_2)
3) Condition_3: Snoop generated from posted write of PCIE (Action_3)

We reused the above action as it is, and added the pss_delay action in front of the above action for ML sequence modeling in the test scenario.

A. *PCIE deadlock reproduce: Setup/Gathering stage*

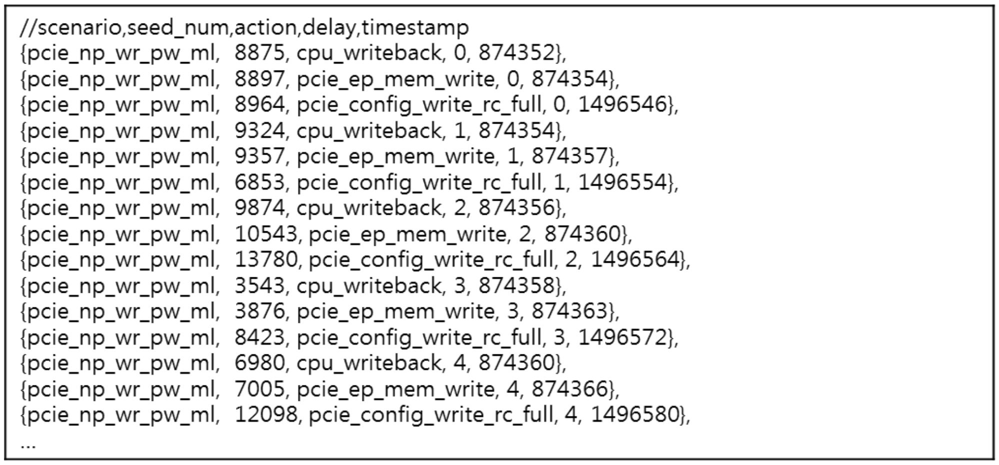
We set the target condition of each action as follows to detect the target condition in the output monitor.



Action_1	top.dut.BLK_HSIO.xxx.pcie_rc_buf_full ==1
Action_2	top.dut.BLK_CPUCL.xxx.AWVALID && top.dut.BLK_NOCL0.xxx.AWREADY && top.dut.BLK_CPUCL.xxx.AWSNOOP[2:0]==‘0b011’
Action_3	top.dut.BLK_NOCL0.xxx.ACVALID && top.dut.BLK_CPUCL.xxx.ACREADY && top.dut.BLK_NOCL0.xxx.ACSNOOP[3:0]==‘0b1101’

Figure 14. Configured output target conditions

And we generated the test by increasing the pss_delay value one step by one from 0 to 100 using the PSS tool. After the execution of the generated test is finished, the learning data stored in the output repository is as shown in the figure below.



```
//scenario,seed_num,action,delay,timestamp
{pcie_np_wr_pw_ml,  8875, cpu_writeback, 0, 874352},
{pcie_np_wr_pw_ml,  8897, pcie_ep_mem_write, 0, 874354},
{pcie_np_wr_pw_ml,  8964, pcie_config_write_rc_full, 0, 1496546},
{pcie_np_wr_pw_ml,  9324, cpu_writeback, 1, 874354},
{pcie_np_wr_pw_ml,  9357, pcie_ep_mem_write, 1, 874357},
{pcie_np_wr_pw_ml,  6853, pcie_config_write_rc_full, 1, 1496554},
{pcie_np_wr_pw_ml,  9874, cpu_writeback, 2, 874356},
{pcie_np_wr_pw_ml,  10543, pcie_ep_mem_write, 2, 874360},
{pcie_np_wr_pw_ml,  13780, pcie_config_write_rc_full, 2, 1496564},
{pcie_np_wr_pw_ml,  3543, cpu_writeback, 3, 874358},
{pcie_np_wr_pw_ml,  3876, pcie_ep_mem_write, 3, 874363},
{pcie_np_wr_pw_ml,  8423, pcie_config_write_rc_full, 3, 1496572},
{pcie_np_wr_pw_ml,  6980, cpu_writeback, 4, 874360},
{pcie_np_wr_pw_ml,  7005, pcie_ep_mem_write, 4, 874366},
{pcie_np_wr_pw_ml,  12098, pcie_config_write_rc_full, 4, 1496580},
...
```

Figure 15. Learning data in the output repository

B. *PCIE deadlock reproduce: Learning stage*

The learning data collected in the output repository is organized in the form of coordinates for each action as follows for linear regression analysis. The bottom of the table shows the linear equation of each action obtained by the linear regression algorithm.

TABLE II
COLLECTED LEARNING DATA

Action 1	Action 2	Action 3
(0, 1496546)	(0, 874352)	(0, 874354)
(1, 1496554)	(1, 876154)	(1, 874357)
(2, 1496564)	(2, 874356)	(2, 874360)
(3, 1496572)	(3, 874358)	(3, 874363)
…	…	…
$y=8x_1+1496546$	$y=2x_2+874352$	$y=3x_3+874354$

C. *PCIE deadlock reproduce: Analysis stage*

In order to reproduce PCIe deadlock, the above Aciton-1, Action_2, and Action_3 must occur simultaneously. In order to find a solution that allows the linear equations with three different variables to have the same result, we first found a general solution for the equations of Action_2 and Action_3 using the Extended Euclidean algorithm as shown below.

- Linear equation of Action_2 : $y=2x_2+874352$
- Linear equation of Action_3 : $y=3x_3+874354$
- Particular solution : $X_2 = 4, X_3 = 2$
- General solution (k : integer value)

$$X_2 = 4 - 3k$$
$$X_3 = 2 - 2k$$

* If k has a value of -103699:

$$X_2 = 4 - 3*(-103699) = 311101$$

$$X_3 = 2 - 2*(-103699) = 207400$$

$$Action\_2: y=2x311101 + 874352 = 1496554$$
$$Action\_3: y=3x207400 + 874354 = 1496554$$

Figure 16. Result of extended Euclidean algorithm for action_2 and action_3

Using this, we searched whether the common timestamp value of Action_2 and Action_3 is in the timestamp measurement value of Action_1. We confirmed that when k has a value of -103699, Action_1, Action_2, and Action_3 have the same output timestamp value of 1496554 as above. Through this process, we could predict that three actions occur simultaneously when $\text{pss\_delay}_{\text{action\_1}}$= 1, $\text{pss\_delay}_{\text{action\_2}}$= 311101, and $\text{pss\_delay}_{\text{action\_3}}$= 207400.

In this process, we obtained a general solution from Action_2 and Action_3, and used the method to find the value obtained therefrom among the timestamp values collected for Action_1. However, it is revealed that it is more efficient to modify the Extended Euclidean algorithm to find a general solution for three variables of *ax_by_cz=y*.

And, as indicated in red of action_1 above, additional delay may occur due to external factors. The most important prerequisite for the sequence model we designed is that it should obtain linear data. For this, factors that can influence from the outside should be removed as much as possible. If non-linear data is included as above, the intercept of the linear equation is changed and it affects the subsequent results. In this case, the linearity of the result range of the analysis stage should be verified again, and then the final result should be passed on to the creation stage.

IV. CONCLUSION

In general, one of the most difficult aspects of ML is collecting data for learning. This is because the more learning data, the higher the prediction accuracy of ML. In this regard, we expected that the combination of PSS, which can easily generate numerous tests, and ML, which finds regularity based on collected data, could exert fantastic synergy. And we were able to dramatically increase the verification coverage by being able to freely create concurrent function events that were previously considered impossible at the simulation level through PSS and ML.

REFERENCES

[1] Accellera,(2018) Portable Test and Stimulus,Version1.0; http://accellera.org/download/standards/portable-stimulus

[2] Jang, M., Kim, J., Chung, H., Huynh, P., Shai, F. (DVCon 2019) Coherency Verification & Deadlock Detection Using Perspec/Portable Stimulus

[3] Python-based ecosystem of open-source software; http://pageperso.lif.univ mrs.fr/~francois.denis/IAAM1/scipy-html-1.0.0/generated/scipy.stats.linregress.html

[4] https://www.britannica.com/science/Diophantine-equation

[5] https://www.math.cmu.edu/~bkell/21110-2010s/extended-euclidean.html

[6] https://mathstats.uncg.edu/sites/pauli/112/HTML/secbezout.html

[7] ARM-EPM-107877 Considerations for Integrating PCI Express with CCI-500 and CCI-550