Closing Functional Coverage With Deep Reinforcement Learning: A Compression Encoder Example

Eric Ohana
Queensland University of Technology, Brisbane, Australia
The University of Bielefeld, Bielefeld, Germany
eric.ohana@hdr.qut.edu.au
eohana@techfak-uni-bielefeld.de

***Abstract*—The simulation based constrained random coverage driven functional verification is one of the most prevalent paradigms used on RTL designs today. While greatly enhancing the verification process, reaching full functional coverage, a necessary step for verification closure, remains a bottleneck. This is mainly due to the necessary manual intervention in the process, consisting of the verification team adjusting stimulus randomization constraints and developing directed tests. This paper tackles this issue by exploring the use and integration of a deep reinforcement learning engine in such a verification environment. Specifically, the technique demonstrated here is a deep learning approach to a Q-Learning variant named Deep Q-Networks, a similar approach to the one developed in the work of DeepMind Technologies to play Atari games [1]. The method is adapted to fit seamlessly into a constrained random coverage driven functional verification environment. The solution development stages along with the simulation results are detailed in this paper. We conclude by making suggestions on which designs may be good candidates for this novel technique.**

Keywords—Simulation Based Constrained Random Coverage Driven Verification, Functional Coverage, Artificial Intelligence (AI), Machine Learning (ML), Reinforcement Learning (RL), Deep Reinforcement Learning (DRL), Feed Forward Neural Network (FFNN), DeepMind, Deep Q-Learning, Deep Q-Networks (DQN), Register Transfer Logic (RTL), RTL Design and Verification, LZW (Lempel Ziv Welch), LZW Compression Encoder, Content-addressable Memory (CAM), Python, PyTorch, SystemVerilog.

I. INTRODUCTION

There have been several endeavors aiming at using ML algorithms for the purpose of optimizing one stage or another of the digital design verification process, see for example [2, 3]. An exhaustive review presenting various ML based techniques, like data mining or inductive logic programming, aiming at generating tests based on the analysis of the simulation coverage data, using supervised learning, has been conducted in [4]. In this paper, we specifically focus on functional coverage closure and reinforcement learning. Reinforcement learning has recently been used in a variety of applications [5] and here we present a deep reinforcement learning based technique that automatically generates stimulus on-the-fly, based on the functional coverage scores achieved during the simulation of an RTL design. We emphasize here that conceptually, RL algorithms are different from supervised learning ones, as the training data for RL algorithms is generated on-the-fly as an inherent part of the learning process itself. This is further discussed in section ***V***.

The parallel in core concepts between RTL verification and reinforcement learning is first clarified along with the role and the integration of a reinforcement learning agent in a modern SystemVerilog based verification environment. The design used in this experiment is an LZW compression encoder written in SystemVerilog RTL. This design is on one hand sufficiently complex to be representative of a typical RTL verification process, and on the other hand, it allows for an unambiguous assessment of the approach's benefits for closing functional coverage. We then present a solution for co-simulating efficiently our deep reinforcement learning agent coded in Python PyTorch and the rest of the verification environment written in SystemVerilog. The deep Q-Learning agent itself, implemented as a DQN, is presented in the section that follows. We report the results of our co-simulations on the LZW compression encoder and show that the use and integration of the deep Q-Learning based technique is efficient in automating the generation of test scenarios in the form of transactions and sequences of transactions that are necessary to hit the most tenacious functional coverage bins of the design, paving the way towards verification closure.

II. RTL VERIFICATION AND REINFORCEMENT LEARNING

An RL system can be thought of as a sequential decision-making process where information between the RL agent and the RL environment is exchanged: at every timestep T, the RL agent executes an action A_T on the RL environment. A_T itself, is a function of the RL environment current state S_T and the reward R_T , which is a scalar value, both obtained from the previous RL agent action A_{T-1} [6]. A_T brings the RL environment to a new state S_{T+1} and a new reward value R_{T+1}. This data is used by the RL agent to learn what subsequent actions shall be selected, in the next timesteps, with the constant objective to maximize the expected future reward as received by the RL environment. The learning process as executed by the RL agent can either estimate a policy function π, which is a direct mapping of an RL environment state to an action, or of a value function. A value function, given a specific policy function π, returns the expected reward of being in a state and taking actions according to π. We focus in this paper on a special value function called the action-value function:

$$Q_\pi(S,A) \rightarrow E(R|A,S,\pi)$$

Formula 1: Action-value function

In words, given a policy π, this function accepts a state and an action and returns the expected reward of being in that state and taking this action [7, 8].

We now transfer to the RTL verification realm, where we specifically focus on the functional coverage closure problem. Firstly, digital simulation can be described as a sequential process which generates a series of input transactions at different timesteps. These input transactions are driven sequentially through a design under test (DUT) and are processed by the DUT. The DUT generates output transactions in return.

The subsequent generation of stimulus, in the form of input transactions to the DUT can be made dependent on these output transactions, or any other effect the previous input transactions had on the design. Therefore, digital simulation can be described as a sequential decision process similarly to an RL system.

We now apply the following translations regarding the RL problem concepts. We first consider a typical modern testbench (TB) as depicted in ***Figure 1*** by focusing on the entities represented by rounded corners rectangles.

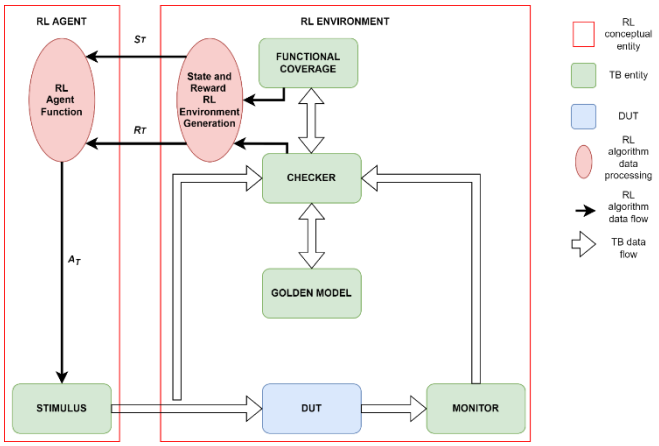

Figure 1: The RL agent and RL environment in a typical test bench

An intelligent test bench entity responsible for generating new stimulus is transformed into our RL agent. The DUT along with the non-stimulus test bench entities form the RL environment. The RL agent generates and drives input transactions to the RL environment. These are its actions. The RL environment generates on-the-fly, a score which is proportional to the functional coverage bin hits that occurred because of a given input transaction. This is the reward. The objective of the RL agent is to hit as many functional coverage bins as possible. This process reaches a termination condition once we reach full functional coverage.

Before we address the RL environment state, we recall here that RL problems are modelled as Markov Decision Processes and need to exhibit the Markov property, meaning that the current RL environment state only, contains enough information for the RL agent to select optimal actions in the view to maximize future rewards [6, 7].

We focus here on functional coverage closure, so our RL environment state is, at every timestep *T*, either a representation of the functional coverage state: what bins have been hit and how many times or a Markov representation of the input transactions that created such a coverage state. The Markov representation is clarified further with our LZW compression encoder example. We also show the translation of the action-value function $Q_\pi(S,A)$ to the functional coverage closure problem.

We emphasize here, that the RL environment state is not the DUT state as represented by the concatenation of its sequential elements. We also remark here that generally a typical functional coverage definition for a given DUT generally includes various categories representing the features of the design. Using our approach, every category shall be represented by its own action-value function $Q_\pi(S,A)$.

III. THE LZW COMPRESSION ENCODER

Our DUT is an LZW compression encoder [8] which architecture is depicted in ***Figure 2*** below:

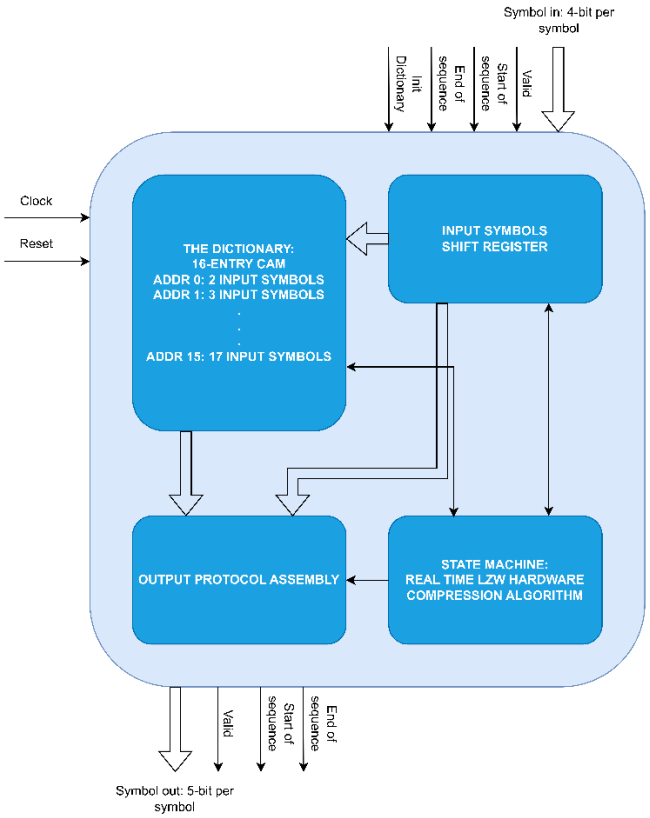

Figure 2:The LZW compression encoder

The LZW compression encoder receives input sequences of 4-bit symbols and uses a 16-address Content-Addressable Memory (CAM) to perform the dictionary-based compression lookup. Hence the compressed output sequence is composed of 5-bit symbols to represent 32 symbols: 16 original input symbols and 16 CAM addresses. The CAM is cleared every time an input sequence starts. It then gets populated anew along with the new input sequence.

To demonstrate the CAM population process, we consider the following sequence of seven 4-bit hexadecimal input symbols:

Start->A,B,A,B,A,B,A<- End

Transaction 1: Input sequence example

Table 1 details the compression and output generation process:

Timestep	Input Symbol 4-bit HEX	CAM[address]	Output Symbol 5-bit HEX
Start: #1	A	-	-
#2	B	CAM[0] = AB	0A
#3	A	CAM[1] = BA	0B
#4	B	Match on CAM[0]	-
#5	A	CAM[2] = ABA	10
#6	B	Match on CAM[0]	-
End: #7	A	Match on CAM[2]	12

Table 1: Compression and output generation processes

The above 28-bit input sequence is compressed into the following 20-bit output sequence:

Start-> 0A,0B,10,12<-End

Transaction 2: Output sequence example

Note that if the output symbol is a CAM address, it is preceded by a '1' otherwise it is preceded by a '0'. As we reset the CAM before every new sequence, the maximum number of symbols that can be written to `CAM[0]` is two, which is the smallest input sequence. We now consider `CAM[1]`: if we get a `CAM[0]` match using the 2nd and 3rd symbol of an input sequence, we can populate `CAM[1]` with a maximum of three symbols. This happens only if the three first input symbols are the same. With the same reasoning, the maximum number of symbols that can be written to `CAM[15]`, which is the highest CAM address, is 17. It will happen with a sequence of the same contiguous input symbols, precisely 137 of them. See ***Figure 3*** for the LZW compression encoder top level waveforms for compressing the sequence above.

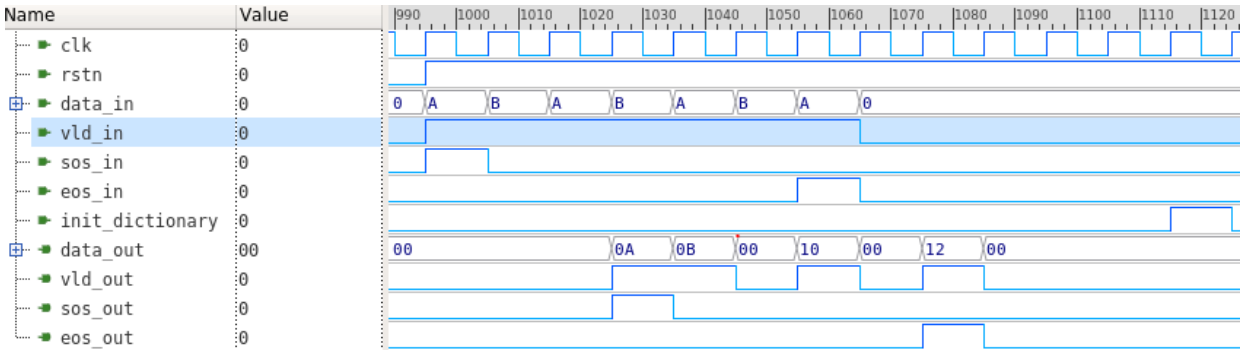

Figure 3: Simulation waveforms of A,B,A,B,A,B,A->0A,0B,10,12

Note: we have used the Aldec Riviera-PRO research edition for all the digital simulations in this research.

Figure 4 below depicts a simplified version of the CAM memory array:

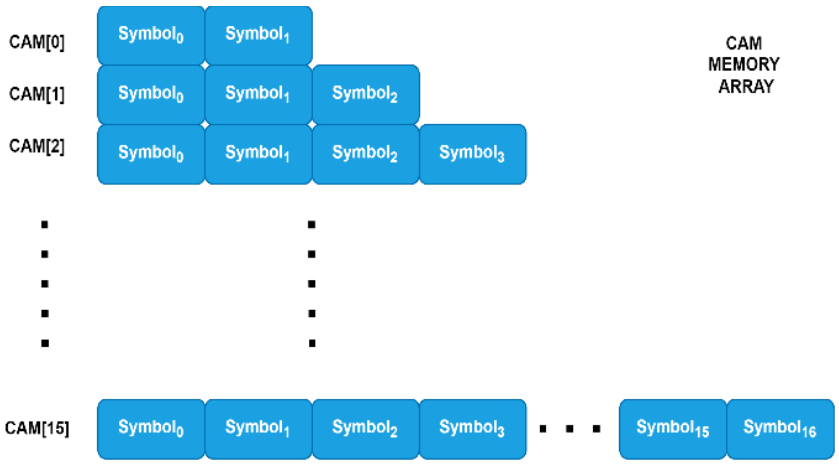

Figure 4: Simplified CAM memory array

We now consider the LZW compression encoder functional coverage and focus on the CAM write functional coverage [9]. We define the bins of this category as follows: for every CAM address, we want to exercise writing every possible input sequence length. For `CAM[0]`, this is length is two. For `CAM[1]`, this length is two or three. For `CAM[15]`, this length is two to 17. We now evaluate the number of functional coverage bins below: the CAM can store 152 input symbols of 4-bit each: two input symbols at `CAM[0]` and up to 17 at `CAM[15]`, hence in total the CAM can store ((2 + 17) / 2) * 16 = 152 input symbols. Now, the smallest input sequence written to the CAM is of two input symbols, so as far as the CAM write functional coverage is concerned, this represents one bin. Hence, we have one bin at `CAM[0]`, two bins at `CAM[1]` and up to 16 bins at `CAM[15]`. This gives us in total ((1 + 16/2) * 16 = 136 CAM write functional coverage bins.

The input sequence patterns necessary for hitting all these functional coverage bins are very specific and reaching them randomly is unlikely, as is shown below using examples.

For instance, the following input sequence pattern can hit the three symbols written to `CAM[3]` functional coverage bin:

Start->A,B,C,C,C,C<-End

Transaction 3: Input sequence example for three symbols in `CAM[3]`

AB is written to `CAM[0]`, BC to `CAM[1]`, CC to `CAM[2]`, CC is matched to `CAM[2]` and finally CCC is written to `CAM[3]`, indeed catching three write locations in `CAM[3]`.

The next pattern can hit the three symbols written to `CAM[4]` functional coverage bin:

Start->A,B,C,D,D,D,D<-End

Transaction 4: Input sequence example for three symbols in `CAM[4]`

AB is written to `CAM[0]`, BC to `CAM[1]`, CD to `CAM[2]`, DD to `CAM[3]`, DD is matched to `CAM[3]` and finally DDD is written to `CAM[4]`, indeed catching three write locations in `CAM[4]`.

During a standard constrained random verification process, it is improbable to hit all the functional coverage bins, by using random input symbol generation. In fact, using a uniform distribution to generate our 16 different input symbols (4-bit), we have managed to hit a maximum of only 28 CAM write functional coverage bins out of the 136 bins, using a substantial amount of input sequences.

Closing the remainder of the CAM write functional coverage bins necessitates, from a verification team, an internal knowledge of the design and the development of customized and directed tests.

We demonstrate here how an RL agent can, by learning from the RL environment state, as per the definition in section ***II***, and the previous functional coverage hits transformed into reward values, generate automatically the input sequences necessary to achieve full CAM write functional coverage. The DUT implementation is resident on GitHub and available upon direct request. See [13].

IV. CO-SIMULATING THE RTL DESIGN AND THE DEEP Q-LEARNING AGENT

Before we detail the implementation of our functional coverage closure DQN agent, we mention here that the Python PyTorch library was used for the purpose of modelling it. PyTorch is an opensource framework, efficient for building deep learning models by providing automatic differentiation [11].

On the other hand, our TB is described in SystemVerilog [10], and the DUT is modelled using the SystemVerilog RTL subset. Therefore, we face the obstacle of co-simulating our DQN agent with our original verification environment.

To address this issue, we developed the architecture described in ***Figure 5***.

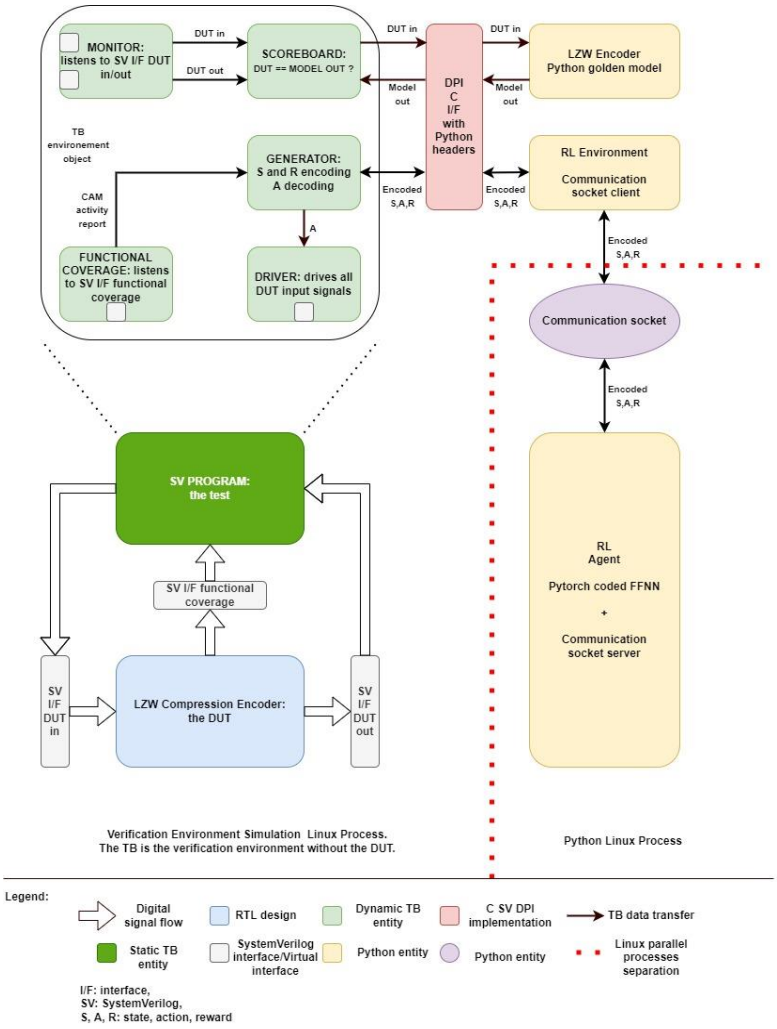

Figure 5: The co-simulation architecture

The main idea behind the architecture is that we have two independent processes which are launched simultaneously and run concurrently. The first process is the digital simulator, as would normally run in an RTL simulation, and the second one runs the DQN agent. The DQN agent awakes from reset in a standby mode, ready for its learning process. It awaits to receive the state from the RL environment and then provides back an action based on the output of an FFNN implemented in the RL agent. The subsequent state and the reward received from the RL environment, as a consequence of this action, are first used to optimize the parameters of our FFNN. The optimization algorithm is the DQN algorithm, as is described in section ***V***. After this optimization step is executed, the same state is then fed to our newly optimized FFNN to generate the new action. This completes a full learning iteration.

In terms of RTL verification, this learning iteration is one input transaction driven to the DUT and the processing of its impact on the functional coverage, in the view of generating subsequent stimulus.

The communication between the digital simulator and the DQN agent is achieved using SystemVerilog direct programming interface (DPI) and C-embedded Python code. The RL agent opens a server socket using the Python socket library and then listens continuously for any activity on this socket.

The RL environment behaves as a client on this socket. Whenever the RL environment state and the reward need to be sent to the DQN agent in order to receive a new action, the RL environment connects to the socket and then the RL agent learning iteration process starts, as it listens continuously to the socket. The socket is depicted as the ellipse in ***Figure 5***.

Implementation wise, we use custom encoding for the state, the reward and the action using size configurable arrays of 8-bit integers. On the SystemVerilog side, we define the following import “DPI-C”:

```
import "DPI-C" function void rl_python(input bit [7:0] data_in [],output bit [7:0] data_out []);

// And on the the C-side, we use embedded Python:

void rl_python(const svOpenArrayHandle data_in,svOpenArrayHandle data_out) {
                        Py_Initialize();
                        // Python script sending state and reward to RL agent
                        // and receiving the action from it.
                        Py_Finalize();
}
```

Code snippet 1:SystemVerilog to/from Python/PyTorch

The sizes of the communication arrays are set on-the-fly during the above two-way communication. The SystemVerilog TB and the communication implementation are resident on GitHub and available upon direct request. See [13].

V. THE DEEP Q-LEARNING AGENT

Figure 6 shows the full implementation for the DQN agent:

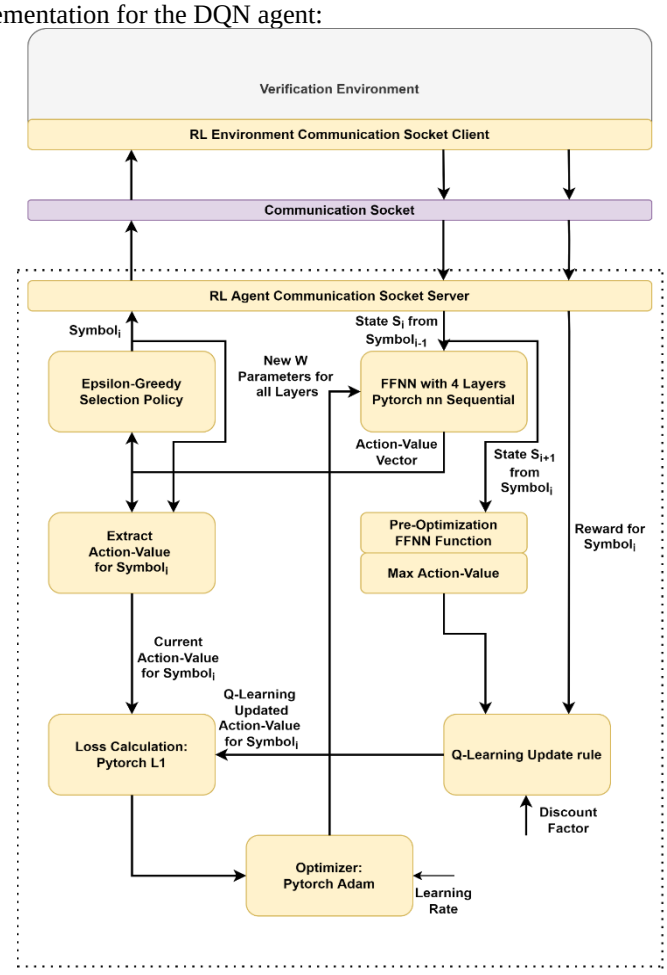

Figure 6: The DQN agent

LZW compression encoder processes, every clock cycle, a 4-bit input symbol, belonging to an input sequence. See ***Figure 3*** for a waveform description of this process.

The action provided by the RL agent to the RL environment is eventually translated into what input symbol shall be driven to the DUT in the next clock cycle.

The RL environment state is itself a representation of the last 17 input symbols as received by the DUT. A series of 17 input symbols is the largest write value and the largest match value, the LZW compression encoder CAM can process. Both the write and the match occur at `CAM[15]`, see ***Figure 4***. A sequence of input symbols is directly responsible for the way the CAM gets populated. We define further the reward function, which in principle, returns a higher value, the more input symbols are written or matched in the CAM. By using both the state and the reward, the DQN agent shall generate symbols to achieve a maximum reward, which in terms of verification means, shall converge towards functional coverage closure.

Figure 7 shows the representation of the RL environment state. It is a 69-bit binary vector: every input symbol is 4-bit and the MSB, the valid bit, signals a post-reset state.

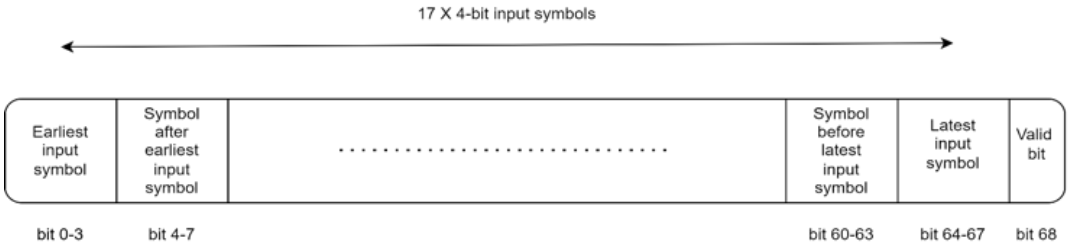

Figure 7: RL environment state

The RL agent is in its core a 4-layer FFNN that processes the 69-bit RL environment state vector as defined above and produces a 16-element output vector. Each element Q(S,a), of this output vector is an action-value scalar. Q(*S*,*a*) is defined as the expected future reward in state *S*, if we execute action *a*. Every action is equivalent to driving a specific input symbol. We showcase the FFNN using a string diagram [12] in ***Figure 8***:

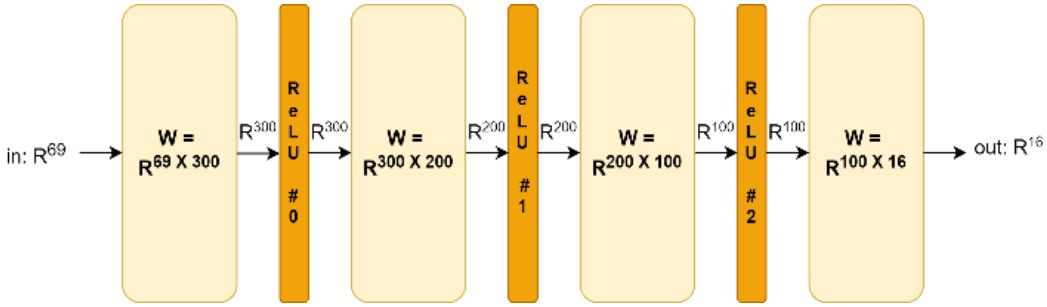

Figure 8: The FFNN for the DQN

The reward itself, in the context of our CAM write functional coverage problem is defined as follows:
if driving an input symbol causes no CAM activity, then the returned reward is null, otherwise, there are two cases:

1- CAM write -> the reward is the number of input symbols written at the CAM address.
2- CAM match -> the reward is the number of valid input symbols at the CAM address + 1

We award an additional point for a CAM match on the account of such an event potentially enables a subsequent larger CAM write event.

We use an ε-greedy policy, denoted as π in section ***II***, on the FFNN 16-element action-value output vector. The policy selects with a probability ε, $Symbol_i$, the input symbol randomly or with a probability (1 – ε), the symbol equivalent to the maximum FFNN vector output. This symbol is then driven to the RL environment. This results in a new state S_{i+1} and a reward R provided by the RL environment.

Before using the new state S_{i+1} to eventually generate a new input symbol, we execute an optimization pass on our FFNN in the view to minimize the loss between $Symbol_i$'s original expected reward in state S, and a re-evaluation of $Symbol_i$'s expected reward in state S, which uses the actual received reward R.

For this purpose, we use the Q-Learning update rule:

$$\mathbf{Q(S_i,A_i) = (1-\alpha)\ Q(S_i,A_i) + \alpha(R + \gamma max\ Q(S_{i+1},a))}$$

Formula 2: Q-Learning update rule

where α is the learning rate, γ is the discount factor, and max $Q(S_{i+1},a)$ is the maximum Q(S,a) output value as given by the FFNN before updating its parameters on the loss, see [6, 7, 8]. In the next section, we detail the actual values used for the Q-Learning update rule during simulation. Using the Q-Learning update rule, the action-value function Q implemented in the FFNN is optimized on each iteration.

The policy π used in the DQN agent is ε-greedy with a decreasing value throughout the episodes/epochs. An example is demonstrated in the next section. The learning process in DQN is fully independent of the policy. Such algorithms are called off-policy algorithms, as the policy choice has no bearing on the prediction accuracy for the action-value function Q(S,A). The RL agent PyTorch implementation is resident on GitHub and available upon direct request. See [13].

VI. THE DQN SIMULATION RESULTS

First, we recall here that a standard constrained random simulation, with a uniform distribution used for the input symbol generation, brought us to a maximum of 28 hits out of the 136 CAM write functional coverage bins, with a CAM overall occupancy of 44 out of 152 CAM locations. In a standard process, the next steps would be manual tuning of constraints and writing series of directed tests to address the functional coverage holes. We now report the DQN use performance on the CAM write functional coverage.

During, digital simulations, the post-reset RL environment state is constant and is also the same as between two contiguous input sequences. Hence an epoch always starts from the same RL environment state. This is also the first value driven to our FFNN. Moreover, every input symbol generates an RL environment state transition as per our definition, creating constant feedback for the DQN learning process.

We run an episode/epoch, equivalent to an input sequence, until the LZW compression encoder CAM is full. This is because subsequent input symbols, in the best case would return matches but also many null rewards and this could deteriorate the learning process achieved during the CAM population.

We first start with a simulation that runs 500 episodes/epochs with a linearly decreasing ε-greedy value ranging from full exploration: 1.0, where all input symbols are selected randomly, to full exploitation 0.0 where all input symbols are selected based on their expected reward as shown by the FFNN output vector. As for the Q-Learning update rule, we set the learning rate to one hundredth: 1.00e-2, and the discount factor to 0.75.

We first bring below in ***Figure 9***, the FFNN loss over the generated symbols. We can observe that it takes around 500 symbols for the FFNN to converge on a low and constant expected reward estimate. We recall here that our goal is to close the CAM write functional coverage, hence we want the DQN agent to generate sequences of input symbols from exploration to exploitation until we reach the following termination condition: all 136 CAM write functional coverage bins have been hit at least once.

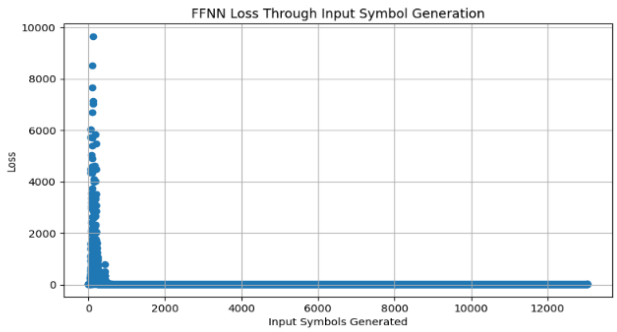

Figure 9: FFNN L1 loss: |X-Y|

Figure 10 is the simulation report after 500 episodes/epochs. We observe that we have managed to hit 133 out of the 136 CAM write functional coverage bins. This is an excellent result, compared to a standard simulation with a uniform distribution on the input symbols which achieved a maximum of 28.

```
CAM WR FC bins hit (max 136): 133 -> 97.79%

Total CAM elements hit (max 152): 152

CAM addr: 15

State/LZW sequence: [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

Reward: 16.0

Episode 499 is complete.

Current loss 35.31644058227539

Experience: 121 of Episode 499 is complete.

All episodes (epochs) completed, exiting ...
```

Figure 10: 500 episodes/epochs final simulation report

We now refer to the last RL environment state, appearing at the State/LZW sequence line in

Figure 10. We remark that this sequence of input symbols is appropriate to populate the higher CAM addresses. On the account that full coverage is almost achieved, it is possible that the DQN agent started generating short repeating sequences in the first episodes/epochs, and then longer and longer ones, as the simulation went through the episodes/epochs. We investigate this point further with the explanation on ***Figure 11***.

The three last missing bins, in this simulation, are the following ones:

1- `CAM[13]` -> write 10 symbols (max 15)
2- `CAM[15]` -> write 12 symbols (max 17)
3- `CAM[15]` -> write 14 symbols (max 17)

Figure 11 below, reports how many CAM locations have been written to (the maximum is 152), throughout the input symbols generation. The black horizontal line in ***Figure 11*** (and also

Figure 12), represents the maximum score achieved by a random uniform distribution of input symbols (44).

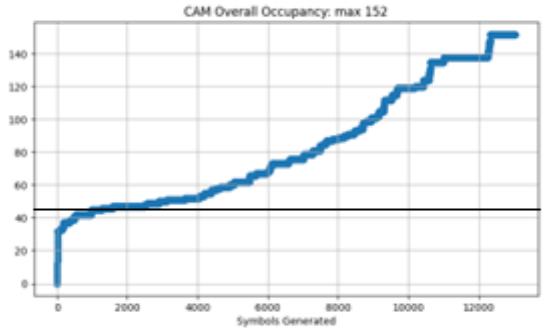

Figure 11: 500 episodes/epochs CAM overall occupancy

In total, we have 152 CAM locations, see ***Figure 4***. Because an input sequence written to the CAM is of a minimum of two input symbols, it gives us 136 CAM write functional coverage bins. Back to ***Figure 11***, we can see, at the beginning of the process, that around 32 locations get filled very quickly. This is because, the input symbol generation process is fully random at the beginning with full exploration and generates mostly non-repeating sequences of two input symbols. Then slowly the learning process kicks in and we can see a quasi-linear population of the CAM locations, meaning that the DQN agent generates longer and longer repeating input sequences, until it manages to occupy the whole CAM with a long enough sequence of repeating input symbols, see the sequences of 3s reported in

Figure 10. Note though that towards the end of the simulation, when the occupancy reaches 140, we can observe a jump. This points to the fact that some functional coverage bins have been skipped, as indeed reported in

Figure 10.

To remedy the three remaining uncovered CAM write functional coverage bins, and instead of developing directed tests, as we would in a standard verification process, we rerun a DQN simulation with 750 episodes/epochs. This should allow for a smoother transition from exploration to exploitation as the decreasing rate for ε is lower than for the first simulation.

Figure 12 shows the CAM overall occupancy throughout the test for the 750 epochs/episodes simulation.

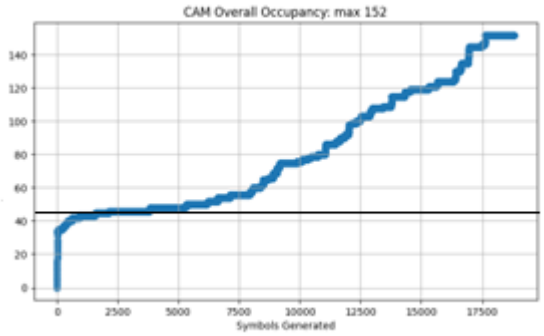

Figure 12: 750 episodes/epochs CAM overall occupancy

The 750 episodes/epochs DQN simulation hits 127 functional coverage bins but most importantly, hit our coverage holes. By merging the functional coverage results of both simulations, we have achieved full CAM write functional coverage results for the LZW compression encoder. For the sake of comparison, ***Figure 13*** shows the CAM overall occupancy for a standard verification process with a uniform input symbol distribution.

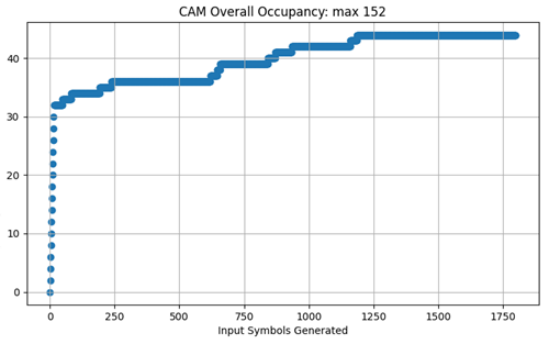

Figure 13: CAM overall occupancy with random uniform input distribution

We see the same steep curve corresponding to random sequences of two input symbols throughout the CAM address range. Then most of the time there are no improvement (flat line) unless a sequence of three or four possible is randomly repeated and causes a little step. Using many input sequences (above 750) to populate the CAM, we have achieved a maximum of 44 compared to full CAM population and full CAM write functional coverage for the DQN approach.

VII. SUMMARY AND CONCLUSION

RL has been previously used for a range of applications [5] and in this paper, we introduced the use of an adapted DQN agent in the context of a modern RTL verification process. During a standard simulation based constrained random coverage driven functional verification, the LZW compression encoder used as the DUT in our experiments would require substantial manual interventions to create the stimulus necessary to close the CAM write functional coverage. It took our DQN agent two series of episodes/epochs, 500 and then 750, to reach all the CAM write functional coverage bins. This result was achieved during standard regression processes in terms of simulation time and computing resources.

We believe this positive result paves the way of a broader use of both the verification environment architecture used in this paper and the DQN agent, including in the context of industry semiconductor projects.

Good design candidates for this paper's approach are RTL designs that show reluctance in achieving good functional coverage results using a standard verification process and require a substantial amount of manual intervention to tune randomization constraints and develop directed tests.

REFERENCES

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[2] Eder, K., Flach, P., & Hsueh, H. W. (2006, August). Towards automating simulation-based design verification using ILP. In *International Conference on Inductive Logic Programming* (pp. 154-168). Springer, Berlin, Heidelberg.

[3] Fajcik, M., Smrz, P., & Zachariasova, M. (2017, December). Automation of processor verification using recurrent neural networks. In *2017 18th International Workshop on Microprocessor and SOC Test and Verification (MTV)* (pp. 15-20). IEEE.

[4] Ioannides, C., & Eder, K. I. (2012). Coverage-directed test generation automated by machine learning--a review. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *17*(1), 1-21.

[5] Li, Y. (2019). Reinforcement learning applications. *arXiv preprint arXiv:1908.06973*.

[6] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[7] Graesser, L., & Keng, W. L. (2019). *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional.

[8] Zai, A., & Brown, B. (2020). *Deep reinforcement learning in action*. Manning Publications.

[9] Welch, T. A. (1984). A technique for high-performance data compression. *Computer*, *17*(06), 8-19.

[10] Spear, C. (2008). SystemVerilog for verification: a guide to learning the testbench language features. Springer Science & Business Media.

[11] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... & Lerer, A. (2017). Automatic differentiation in pytorch.

[12] Xu, T., & Maruyama, Y. (2021, October). Neural String Diagrams: A Universal Modelling Language for Categorical Deep Learning. In *International Conference on Artificial General Intelligence* (pp. 306-315). Springer, Cham.

[13] Ohana, E. (2022). LZW D&V, https://github.com/eric-ohana-phd/LZW