An Open Source Solution for Digital Hardware Verification using Reinforcement Learning

A Thesis

submitted by

Aebel Joe Shibu

in partial fulfilment of the requirements
for the award of the degree of

B.Tech and **M.Tech**

Department of Electrical Engineering
Indian Institute of Technology Madras

June 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **An Open Source Solution for Digital Hardware Verification using Reinforcement Learning**, submitted by **Aebel Joe Shibu**, to the Indian Institute of Technology, Madras, for the award of the degree of **B.Tech** and **M.Tech**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other institute or university for the award of any degree or diploma.

Dr. Pratyush Kumar
DDP Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Dr. Janakiraman Viraraghavan
DDP Co-guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 18th June 2021

ACKNOWLEDGEMENTS

I sincerely thank my project guide, Dr. Pratyush Kumar, for his valuable insights and constant guidance throughout my Dual Degree Project timeline. His friendly support contributed significantly towards the successful completion of this project. I also express my gratitude towards my teammate Sadhana S for the technical support on the hardware design aspects and the countless useful discussions that we had. I am also indebted to my parents and brother for their love and support, which has helped me through uncertainties time after time.

ABSTRACT

KEYWORDS: Digital Hardware Verification, Reinforcement Learning

Digital hardware design verification has traditionally been done by simulating various input signals and comparing how the design behaves for those inputs with respect to a reference model. The inputs are chosen randomly, ensuring that they lie within some valid constraints as per the hardware design specifications. This means that sufficiently exploring different parts of the design are left to chance and might take very long simulation times depending on how infrequent certain events occur in the design. The objective of this project is to provide a open-source software framework capable of seamlessly incorporating some of the existing reinforcement learning (RL) approaches into the hardware verification pipeline for improving the process using AI. The framework allows the development of testbenches without significant additional overhead in terms of engineering effort when compared with traditional hardware design verification environment development, and serves as a starting point for further research and development aimed at making the process more intelligent using AI.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS i

ABSTRACT ii

LIST OF TABLES v

LIST OF FIGURES vii

ABBREVIATIONS viii

NOTATION ix

1 INTRODUCTION 1

2 BACKGROUND 3

2.1 Hardware Verification 3

2.2 Reinforcement Learning 5

2.3 Why RL is a good candidate for hardware verification 7

3 RELATED WORK 8

4 FRAMEWORK 12

4.1 Overview 12

4.2 Verilog layer 13

4.3 RL layer 14

4.4 Cocotb layer 16

4.5 Usage 16

5 EXPERIMENTS 21

5.1 RLE compressor 22

5.2 COO compressor 26

5.3 COO decompressor 27

5.4 RLE decompressor 33

6 FUTURE WORK **37**

LIST OF TABLES

4.1 Mapping of the framework functions to their respective triggers . . . 17

5.1 Functional events tracked in RLE compressor 23

5.2 Functional events tracked in COO compressor 27

5.3 Functional events tracked in COO decompressor 30

5.4 Functional events tracked in RLE decompressor 33

LIST OF FIGURES

2.1 State diagram of a 1-bit branch predictor 4

2.2 Block diagram showing a typical digital hardware verification pipeline 5

2.3 The RL feedback loop 6

2.4 A typical sequential circuit 7

4.1 An illustration of state aggregation. 13

4.2 A top level view of the framework 13

4.3 Illustration of the definition of an episode in OpenAI Gym environments 15

4.4 Block diagram of the complete framework with the component layers demarcated using magenta (cocotb layer), blue (Verilog layer) and orange (RL layer) 17

4.5 Software architecture of the framework 18

5.1 Comparison of functional coverage achieved with the RLE compressor after 1000 iterations. The x-axis corresponds to e_0, e_1, e_2 and e_3, the four functional events tracked as part of coverage. 24

5.2 Histograms of action component 1 chosen with and without RL feedback 25

5.3 Histograms of action component 2 chosen with and without RL feedback 25

5.4 Histograms of action component 3 chosen with and without RL feedback 25

5.5 Comparison of plots showing the evolution of reward as the steps progress for the RLE compressor 26

5.6 Comparison of functional coverage achieved with the COO compressor after 1000 iterations. The x-axis corresponds to $e_0, e_1,$ and e_2, the three functional events tracked as part of coverage. 28

5.7 Histograms of action component 1 chosen with and without RL feedback 29

5.8 Histograms of action component 2 chosen with and without RL feedback 29

5.9 Comparison of plots showing the evolution of reward as the steps progress for the COO compressor 29

5.10 Comparison of functional coverage achieved with the COO decompressor after 1000 iterations. The x-axis corresponds to $e_0, e_1,$ and e_2, the three functional events tracked as part of coverage. 31

5.11 Histograms of action component 1 chosen with and without RL feedback 32

5.12 Histograms of action component 2 chosen with and without RL feedback 32

5.13 Comparison of plots showing the evolution of reward as the steps progress for the COO decompressor 32

5.14 Comparison of functional coverage achieved with the RLE decompressor after 1000 iterations. The x-axis corresponds to $e_0, e_1,$ ..., e_6, the seven functional events tracked as part of coverage. 35

5.15 Histograms of action component 1 chosen with and without RL feedback 36

5.16 Comparison of plots showing the evolution of reward as the steps progress for the RLE decompressor 36

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
RL	Reinforcement Learning
MDP	Markov Decision Process
FSM	Finite State Machine
DUT	Device Under Test
UVM	Universal Verification Methodology
RLE	Run Length Encoding
SAC	Soft Actor Critic

NOTATION

R	Reward
S	State space
A	Action space
s_t	state at the end of timestep t
a_t	action for timestep t
R_t	reward at the end of timestep t
e_i	i-th functional event being tracked
m_i	weight of event e_i in the reward computation
n_i	number of clock cycles in which event e_i was hit

CHAPTER 1

INTRODUCTION

The ability to process and store huge amounts of data has been one of the key enablers of AI's recent successes. Even with the vast amounts of useful data that is available from various spheres, the limitations in how efficiently it is being used lately have been because of the rapid improvements in hardware technology coming to a halt in the post-Moore era. With the increasing demand for more powerful hardware, including but not limited to domain-specific processors, compressors and storage elements, it is necessary to make the development pipeline of hardware as robust and efficient as possible.

Hardware verification is the procedure of testing if a given hardware design correctly implements the specification. It is recognized as the largest task in silicon development and has the biggest impact on the key business drivers of quality, schedule, and cost. As hardware designs get more complicated, the engineering effort involved for testing and verification of it becomes one of the key aspects where improvements are possible. A significant barrier in enabling efficient techniques to come into this sphere is the need for human intervention in the process. The advent of powerful AI methods which can now be deployed using nominal resources offers the prospect of replacing some of this human intervention to save time and effort in the verification procedure.

In addition to demonstrating the successful use of AI in hardware verification, there arises a need to establish a framework that can serve well as a starting point for further research and development on the topic since the various open-source tools that are required for the individual components of such a framework already exists. Using AI successfully in the realm of hardware design verification could also speed up the adoption of such techniques for software design verification in the future.

The key contribution of this project is the development of a framework for verification of hardware leveraging the existing open-source tools available in the field of Reinforcement Learning(RL) and hardware verification. The framework is used to build the verification environments for some hardware designs, examining its usability in terms of the additional engineering effort involved due to the inclusion of RL into the

development pipeline as well as the improvements in the verification results when the RL feedback is present. This thesis also identifies why it is a good time to tackle this problem now and provides a possible roadmap for how this framework could evolve in the future.

The rest of the thesis is organized as follows. Chapter 2 goes over the preliminaries of hardware verification and RL. Chapter 3 surveys some of the existing literature on RL and its applications in the systems field. In addition, some of the open-source tools available for RL are also examined. Chapter 4 explains the framework developed in terms of its individual components as well as how those components interact with each other. Chapter 5 describes some of the experiments performed with this framework and discusses the results from them. Chapter 6 concludes with the possible next steps in the context of the framework and its usage.

CHAPTER 2

BACKGROUND

2.1 Hardware Verification

Digital hardware verification currently involves the designer specifying a set of constraints for the input stimulus to the device under test (DUT) to be valid. The verification tests comprise of selecting the input signal values randomly from within these constraints and simulating the DUT to test the correctness of the design. Although the sheer randomness in this type of constrained random verification approach is beneficial for discovering bugs in the design, a more principled approach for hardware verification currently in use relies on evaluating the quality of the verification in terms of functional coverage.

The functionality that is to be verified is mapped to certain events in the hardware design as part of the verification plan and these events are converted to code to define what are called functional coverage points. The events could be certain signals/registers taking certain values, certain buffers getting full, etc. completely dependent on the details of the hardware design. The occurrence of these events during the verification tests signifies that the required functionality associated with these events has been exercised. Functional coverage is thus a description of the events that need to happen in the hardware for some functionality to be exercised. It is different from a typical test description in that, a test description describes how to exercise a design in order to test some functionality while the functional coverage point exists just to ensure that the functionality was exercised rather than describe the logic for how to test it. Thus evaluating the outcomes of a verification run using functional coverage is a way to ensure that the design does indeed meet the corresponding functionality of the design specification correctly.

The functional coverage achieved in a verification run is typically collected by sampling events and combinations of values that have occurred in the DUT, pertaining to different parts of the design specification. For example, consider a DUT that corresponds to a 1-bit branch predictor. The state diagram is fairly trivial and is shown in

Figure 2.1. It has two states and can be represented using a single bit. Let this bit be stored in a register *REG* of the DUT and *REG* taking the value of 0 correspond to the **Predict "do not take branch"** state whereas REG taking the value of 1 correspond to the **Predict "take branch"** state. Then, $REG == 1$ serves as the functional coverage point for when the DUT has entered the **Predict "take branch" state**. Thus, if during the verification run, it is observed that the condition $REG == 1$ is sampled a sufficient number of times, the functional correctness of the **Predict "take branch" state** is more guaranteed.

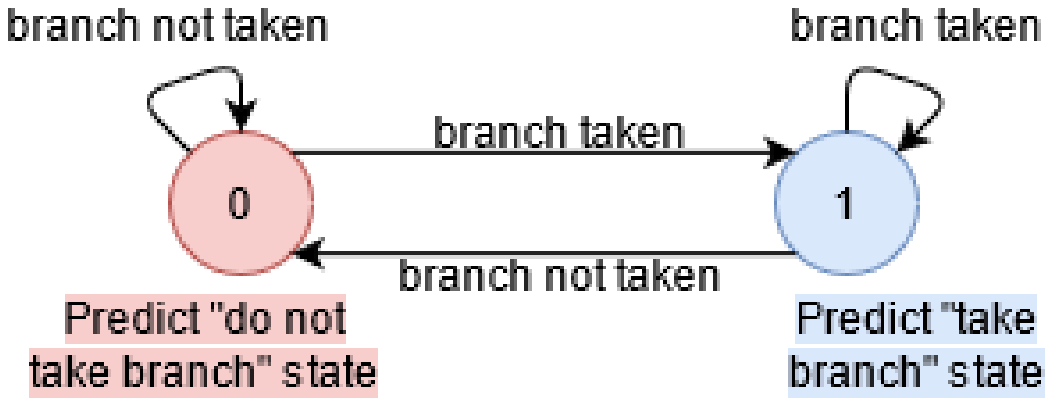


Figure 2.1: State diagram of a 1-bit branch predictor

A typical verification pipeline is shown in Figure 2.2. The input generator provides inputs to the DUT and the reference model. The outputs from both are compared against each other to verify correctness while the functional coverage gets tracked.

Using functional coverage provides the verification engineer with a measure of the functional completeness of the testing, and tells them when the goals set out in the verification plan have been met, and the simulation can be stopped. Instead of writing tests to exercise specific features of the design, these features are fully enumerated in the functional coverage model, and the tests are written only to steer the constrained random stimulus generation towards filling the functional coverage points. The iterative process involving the formulation of these tests is one of the key aspects of the verification pipeline where human intervention is needed. As part of this work, some of this feedback using human intelligence has been automated using Reinforcement Learning(RL) thus making the process more intelligent by itself.

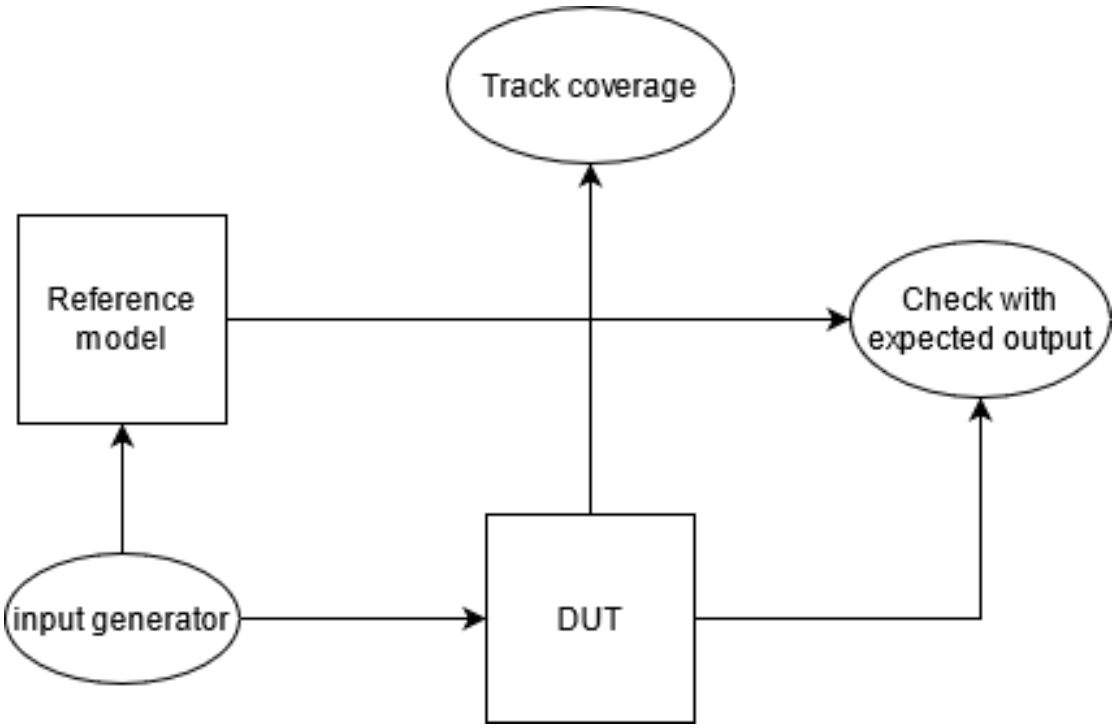


Figure 2.2: Block diagram showing a typical digital hardware verification pipeline

2.2 Reinforcement Learning

Reinforcement Learning (RL) refers to an area of machine learning where the problem is modeled in terms of an environment wherein an agent is expected to make decisions that should ultimately lead to the solution of the problem. The environment and the agent are typically referred to as the RL environment and the RL agent respectively, and the latter is where the intelligence for solving the problem is made to manifest. This is achieved by allowing the RL agent to learn about the consequences of the decisions that it takes by feeding it with appropriate rewards.

RL formalizes the problem statement as a Markov Decision Process (MDP) to define the interaction between the RL agent and the RL environment. The MDP framework is simple in terms of the components while allowing the representation of a wide variety of problems. The feedback loop given in Figure 2.3 represents how the RL agent learns. In each timestep t, the action a_t chosen by the RL agent results in consequences in the environment. The environment is represented in such a way that the effects in the environment in the timestep t has no dependence on the past and is completely specified (either stochastically or deterministically) by the action taken a_t and the state at the start of the timestep s_{t-1}. At the end of each timestep, the agent receives the updated state

s_t from the environment. In addition, since the goals of the problem being solved are modeled in terms of a reward signal, the RL agent also receives a reward R_t at the end of each timestep indicating whether the consequences of the action a_t were favorable or not.

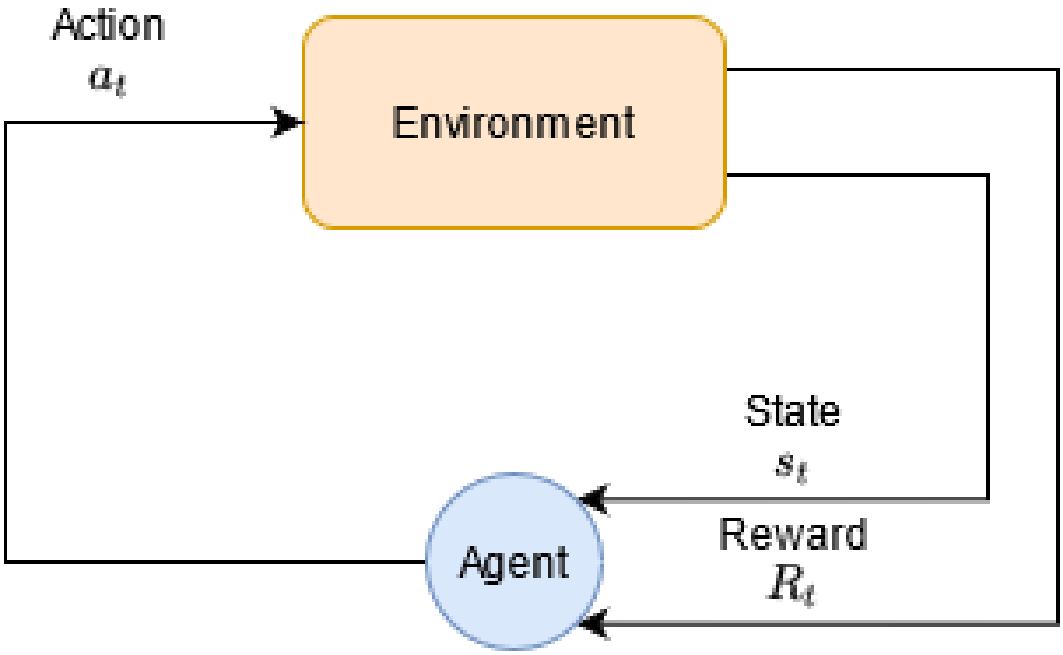


Figure 2.3: The RL feedback loop

Typically, RL attempts to solve the problem by learning through trial and error. Since the reward signal is formulated such that a positive reward is given to the RL agent when it chooses an action that results in effects that bring the environment closer to the goal state, the agent learns to favor that action since it resulted in favorable consequences and in turn led to a positive reward. Therefore, how the reward function is defined has a significant impact on the ability of the RL agent to solve the problem and needs to be done with care. In addition, since this process of learning is reliant on the agent being able to explore actions and discover ones that are rewarding, it results in the classic exploration vs exploitation tradeoff in RL problems.

Similarly, what each state and action of the MDP represents in terms of the problem domain is also important. The MDP definition can essentially be seen as a way to distill the important domain knowledge about the problem in such a way that it can be represented in a manner solvable using RL algorithms. The MDP definition as well as the exploration vs exploitation tradeoff are important in the specific context of digital hardware verification.

2.3 Why RL is a good candidate for hardware verification

The MDP framework used by RL is a simple way of representing the essential features of the artificial intelligence problem [Sutton and Barto (2018)]. The underlying similarities between an RL environment and the finite state machine of a hardware design are what make RL a strong candidate for artificial intelligence to be added into the feedback loop of hardware verification. The hardware implementation of a finite state machine (FSM) consists essentially of just two types of components, namely, the combinational elements and the memory elements as shown in Figure 2.4.

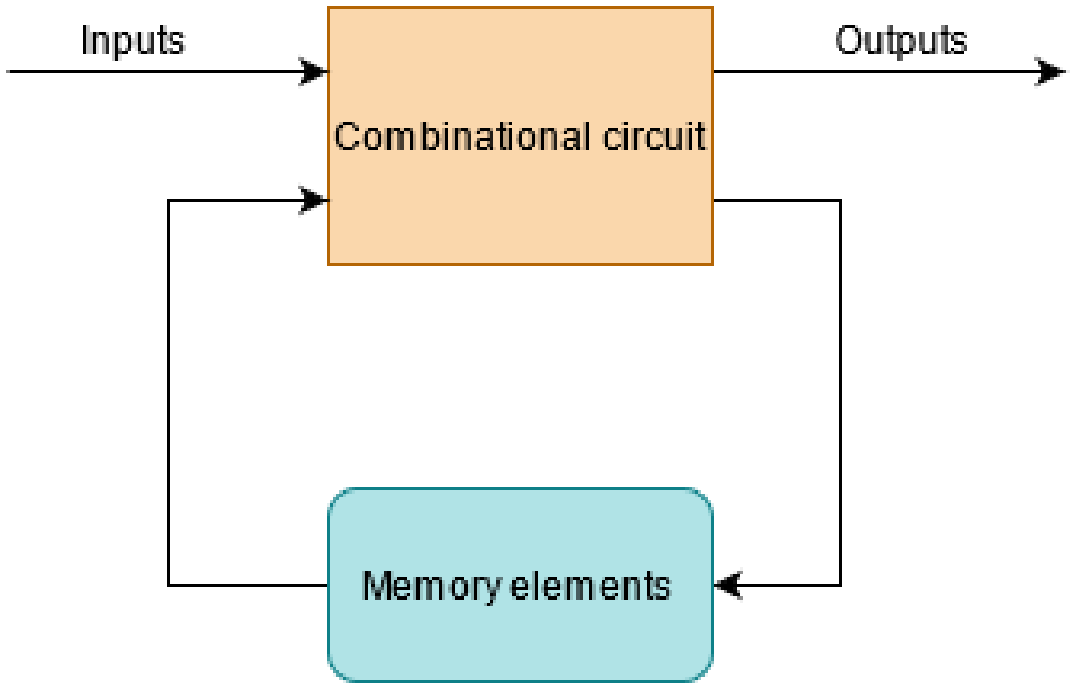


Figure 2.4: A typical sequential circuit

The state of the finite state machine is completely specified by the contents of the memory elements. This structure of what comprises a state in the hardware naturally fits into the RL framework as the RL environment is formalized as an MDP which inherently involves defining states, abstracting the details of what happens in the environment. The DUT on which the verification is performed can thus be considered as the RL environment and the RL agent replaces the human intelligence which steered the verification procedure towards higher coverage in traditional verification. From this point on, in order to avoid ambiguity, the state of the FSM is referred to as the hardware state, and the state of the MDP is referred to as the Markov state wherever applicable.

CHAPTER 3

RELATED WORK

Although the essential components of many of the RL algorithms currently in use have been in existence since the 90s, RL has achieved dramatic breakthroughs only after the advent of deep learning. Some of the recent achievements in the field involve massive success in Atari games [Mnih *et al.* (2013)], Go [Silver *et al.* (2016)], Chess [Silver *et al.* (2017)] and StarCraftII [Vinyals *et al.* (2019)]. The approximation power of neural networks along with the ample availability of computing resources is what allowed to extend the success of RL from tabular low-dimensional problems to problems that are much larger in scale.

One of the oldest RL algorithms, Q-learning [Watkins and Dayan (1992)] was made usable for high-dimensional problems in the form of Deep Q-learning [Mnih *et al.* (2015)] and is a common algorithm in use today. The value function is approximated using a neural network which is referred to as a Deep Q-network (DQN). DQNs were the first among RL algorithms able to work directly from raw visual inputs [Arulkumaran *et al.* (2017)] and are now used for RL environments having a discrete action space.

Another class of RL algorithms that tries to learn the optimal policy of the RL agent directly exists. These policy search methods could either be based on backpropagation or gradient-free methods. Some examples of these methods include Trust Region Policy Optimization (TRPO) [Schulman *et al.* (2017*a*)] and Proximal Policy Optimization (PPO) [Schulman *et al.* (2017*b*)]. In contrast to DQNs, these methods are usable in RL environments with continuous action spaces as well.

Actor-Critic algorithms are another kind of popular RL algorithms that combine some of the advantages of policy search methods with traditional learned value functions. These methods involve the training of one neural network for the policy of the RL agent and another neural network for the value function and another neural network for the policy. These two networks are called the Actor network and the Critic network respectively and together they constitute the Actor Critic algorithm. An example of

this class of algorithms is Deep Deterministic Policy Gradient (DDPG) [Lillicrap *et al.* (2015)]. DDPG is closely related to deep Q-learning and can be considered as deep Q-learning for continuous action spaces rather than discrete action spaces.

Soft Actor Critic (SAC) [Haarnoja *et al.* (2018)] is an actor-critic method the training of which involves maximizing the expected return while regularizing entropy of the policy. This algorithm is used as part of the RL feedback in the framework developed in this project as it achieves state-of-the-art performance in some of the RL benchmarks and seemed more robust to randomness.

RL has seen a notable amount of success in the systems field. DeepRM [Mao *et al.* (2016)] and DeepRM2 [Ye *et al.* (2018)] are two of the recent success stories in the resource management domain. Policy gradient methods were used to solve the problem of handling tasks with different resource demands. Liu *et al.* (2017) proposes a hierarchical framework using RL for resource allocation as well as power management in cloud computing systems. The hierarchical nature of the framework was adopted to reduce the complexity of the state/action space and to enable distributed operation of power management [Li (2018)].

Several applications of RL can be seen in the field of network routing protocols as discussed by Mammeri (2019). A notable algorithm called Q-routing [Boyan and Littman (1993)] which is based on Q-learning serve as the starting point for many of the existing RL based routing algorithms. Chen *et al.* (2020) proposes RLRouting which uses network throughput and delay to compute the reward for the RL agent. The agent learns a policy that predicts the future behavior of the network and suggests better routing paths between switches.

In the context of hardware systems, RL has been successfully used for building a solution to the problem of chip placement, one of the important stages of the hardware chip design process. Mirhoseini *et al.* (2020) formulates the reward signal using the wirelength required as a measure of the placement quality as the RL agent attempts to map the nodes of a chip netlist onto a chip canvas. Another application of RL in the systems field, related to security can be seen in [Böttinger *et al.* (2018)] where deep RL is used to aid fuzzing to find security vulnerabilities in input-processing code.

Specific to hardware verification, with simulation-based verification as the predom-

inant method for hardware verification, many sophisticated approaches using Bayesian networks, genetic algorithms, Markov models, and inductive logic have been tried to improve its effectiveness, as reviewed by Ioannides and Eder (2012). One such recent work involves [Wang *et al.* (2018)] in which artificial neural networks(ANN) are used in the verification pipeline to extract critical features of the test transactions and also learn the priority of coverage groups based on previous test iterations. Using this information which the neural network learns over multiple previous iterations, the input stimulus to the DUT is modified in the subsequent test iterations. Another recent work by Singh *et al.* (2019) involves the use of RL to generate all possible sequences of vectors needed to approach a target state as well as the corresponding path to the target state which contains a potential design error. This approach uses prior information in the form of the state diagram of the FSM to devise the RL feedback. Hughes *et al.* (2019) recommends the use of RL for design verification, where the verification problem is modeled as a single step MDP and serves as a key starting point for this project.

The use of AI in providing feedback to coverage-based hardware verification pipelines introduces the need for verification engineering personnel to be knowledgeable in the related techniques. Therefore, an important criterion for selecting such a method for practical use is how technically demanding it would be to construct and fine-tune the verification environment [Ioannides and Eder (2012)]. In addition, ease of problem representation, the extent of prior knowledge about the design required, and how well the framework can accommodate different kinds of designs are some other important criteria to be considered. Taking into account all these factors, the framework built as part of this work is believed to be a good choice.

Although test benches for verification of hardware have been traditionally written using hardware description languages(HDLs) like VHDL or SystemVerilog, cocotb has been used for this project. Cocotb is a free and open-source package using which the test benches for verification can be written in a higher-level general-purpose programming language like Python. It encourages the same philosophy of design re-use and randomized testing as Universal Verification Methodology(UVM) and was specifically designed to lower the overhead of creating a test [Higgs and Hodgeson (2013)]. The advantage of being able to use Python for the tests is twofold. Firstly, test benches are easier to write and set up since Python is a programming language that is much more accessible and widely used compared to conventional HDLs. Secondly, and more relevant

to this work, the testbench logic being completely written in Python provides access to a plethora of open-source software packages which are compatible with Python. This is paramount in letting us efficiently leverage the existing work in the field of AI since a major chunk of the research and development done in this regard has been using Python.

One such open-source tool used as part of this work in order to build a robust framework with RL is OpenAI Gym. One of the key problems in the field of RL that OpenAI Gym tried to solve was the lack of standardization of RL environments used in published research [Brockman *et al.* (2016)]. The RL environments provided as part of Gym were made to adhere to a specific structure that was general enough to be able to represent a wide variety of RL environments using the library. In addition, tools were provided for new RL environments to be defined in the same structure. Since Gym does not make any assumptions about the structure of the RL agent, it is compatible with the various numerical computation and RL algorithm packages available.

Complementing the RL environment implemented with the help of OpenAI Gym, Stable Baselines3, a set of reliable implementations of reinforcement learning algorithms in PyTorch [Raffin *et al.* (2019)], has been used for the RL agent as part of this work. In essence, the usage of cocotb has made the implementation of the verification testbench to be in Python which results in access to RL tools like OpenAI Gym and Stable Baselines3 and easy interfacing of these tools with the verification pipeline using the sophisticated control available in Python. There are also projects like RL Baselines3 Zoo [Raffin (2020)] which consist of supporting resources for aiding the successful use of Stable Baselines3 by providing scripts for training, evaluating agents, tuning hyperparameters and plotting results. In addition it contains tuned hyperparameter values for common RL agents and environments.

The final building block used in the framework is the open-source tool Verilator [Snyder (2001)] which is compatible with cocotb and is used to simulate the hardware design written in Verilog for the verification procedure.

CHAPTER 4

FRAMEWORK

4.1 Overview

The hardware verification problem has been modeled as an episodic RL task where each run of the verification procedure corresponds to one RL episode. Each episode could comprise multiple timesteps (used interchangeably with step) where each step involves the RL agent receiving the state and reward signal from the environment, followed by generating the next action based on some policy. As part of the MDP definition, the input stimulus to the DUT has to be parameterized suitably using the action space. Thus each action chosen by the agent should result in a sequence of valid input signals driving the DUT for that step of the RL episode. Subsequently, the reward signal can be defined as a function of the number of times each of the functional events that need to be covered in the verification procedure are hit.

Although the most natural way to define the Markov state space would be to have it similar to the state space of the FSM of the DUT, this is impractical since the state space of the FSM could be huge with potentially thousands of registers. Instead, some of the key registers associated with the DUT could be identified and used to define the Markov state space. Such a representation is still meaningful and can be considered as a way of state aggregation wherein a large number of the states of the FSM are combined to form aggregated states that map to the Markov states in the MDP. An illustration of this concept is shown in Figure 4.1. Since this sort of state aggregation results in the loss of details about how the component states of the aggregated state interact, it has to be done without losing sight of what the key registers that can be used to represent the DUT are. The choice of what these registers are is essentially a way of providing prior domain knowledge to the RL agent by better representing the problem.

The architecture of the framework consists of three layers that interact with one another as shown in Fig. 4.2. The Verilog layer consists of the hardware design written using Verilog and the cocotb layer consists of the testbench written using cocotb.

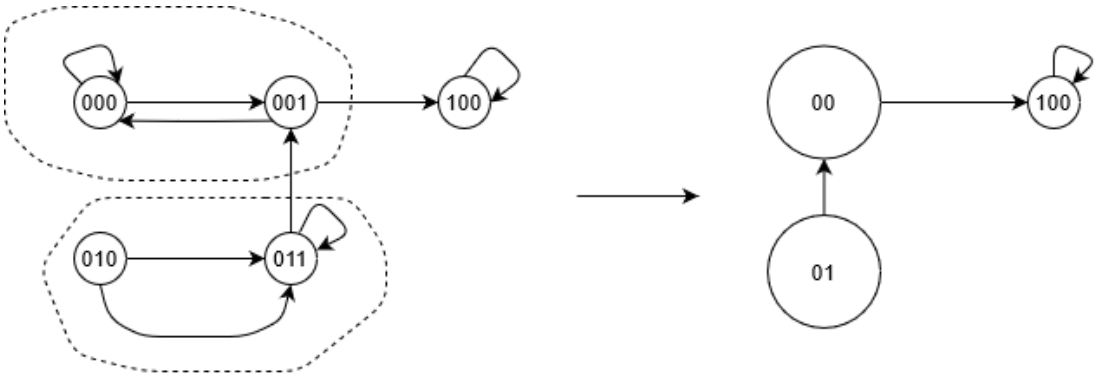


Figure 4.1: An illustration of state aggregation.

These two layers interact with each other, simulating the DUT according to the verification logic. A conventional verification environment for hardware, written using cocotb, consists only of these two layers. The RL layer has been added on top of this to interact directly with the cocotb layer and indirectly with the Verilog layer (that is, DUT), thus completing the feedback loop for adding intelligence to the process. The details associated with the three layers are discussed in the following sections.

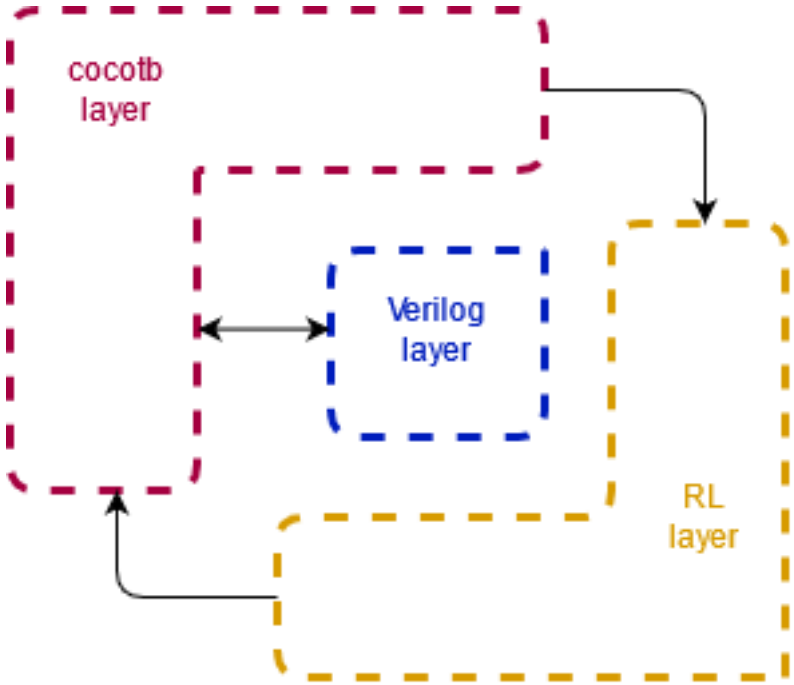


Figure 4.2: A top level view of the framework

4.2 Verilog layer

Everything written using Verilog constitutes this layer. Although in traditional verification environments, both the hardware design and the test bench logic can be written

using Verilog/SystemVerilog, in cocotb-based verification, only the hardware design is written using Verilog. The testbench logic is instead using cocotb/Python. Therefore in our framework, the Verilog layer contains only the DUT design. The DUT is instantiated as the top level in the simulator and receives stimulus from the cocotb layer to drive the input signals for verification. Further, it interacts with the cocotb layer so that the verification run can be monitored for the coverage and computing the necessary data for the RL feedback.

4.3 RL layer

The RL agent makes up the core of the RL layer. The agent was implemented using Stable Baselines3 but can be replaced with any RL algorithm library as long as it is compatible with OpenAI Gym environments as the verification environment was modeled in the Gym format. The RL layer logic is executed in a process of its own which runs in parallel with the process that handles the cocotb and Verilog layers. This process can be thought of as the agent process, while the process which contains the logic of the cocotb layer and Verilog layer is the environment process.

The execution flow of each RL episode in a Gym environment can is in phases as shown in Figure 4.3. Every episode starts with a call to the reset function of the Gym environment. The reset function is used to reset the RL environment to the start state before the beginning of the next episode. Followed by this, the step function which executes the logic for each timestep gets called. One episode of RL could comprise of just a single step or could have multiple successive steps as shown in Figure 4.3. The number of steps in each RL episode, as well as, what each step corresponds to in terms of the DUT verification procedure is a property of the environment and hence decided by the user who devises the verification logic for that DUT.

The agent process and the environment process communicate with each other with the help of a Pipe object which is native to the multiprocessing module of Python. Through this pipe, the RL agent sends the action for the next step, as dictated by the policy it follows, to the environment process. Similarly, at the end of each step, the RL agent receives the information related to the state and reward, from the environment process, through this pipe.

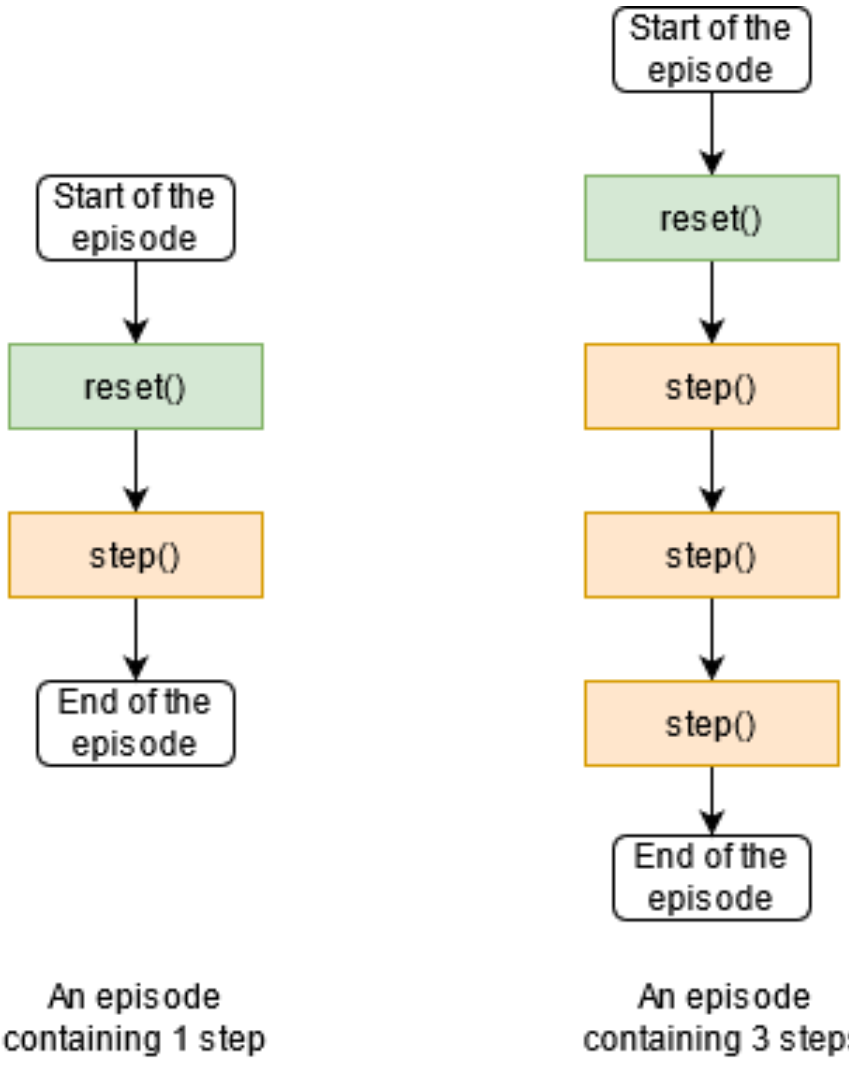


Figure 4.3: Illustration of the definition of an episode in OpenAI Gym environments

4.4 Cocotb layer

The cocotb layer contains the testbench for verifying the DUT. It interacts with the DUT during the simulation and acts as the bridge between the Verilog and RL layers. The logic of this layer executes as part of the environment process that runs in parallel with the agent process discussed in Section 4.3. The action chosen by the RL agent at the start of each timestep is received at this layer through the pipe and is processed to generate the exact input signal sequence that must be driven to the DUT as part of that step. While the DUT gets simulated with that input sequence, the elements of interest within the DUT like registers, buffers, etc., are monitored using cocotb coroutines, similar to how it is done in a traditional cocotb-based hardware verification setup. Once the verification logic for that step completes, the information required for the Markov state and reward computation is communicated back to the agent process via the pipe.

The complete block diagram of what happens in the framework in one timestep, along with which layer each block belongs to, is shown in Figure 4.4.

Although the OpenAI Gym environment-related function definitions reside in the RL layer, the verification logic associated with the environment is in the cocotb layer. The verification logic in the cocotb layer is triggered when the RL agent calls the reset() and step() functions of the Gym environment. Similar to how a conventional Gym environment is implemented by defining the state and action spaces and overriding the reset() and step() functions provided in the Gym module, for implementing the hardware verification testbench logic, abstract function definitions that can be overridden by the user, are provided in the cocotb layer. These functions get called in the simulation loop in a specific sequence based on triggers from the RL agent and RL environment. The cocotb layer architecture is defined with abstract functions like this so that the user can write verification logic of any complexity using cocotb without having to deal with the details of interfacing between the RL and cocotb layers. The key abstract functions and their respective triggers are shown in Table 4.1.

4.5 Usage

The overall software architecture of the framework is presented in Figure 4.5.

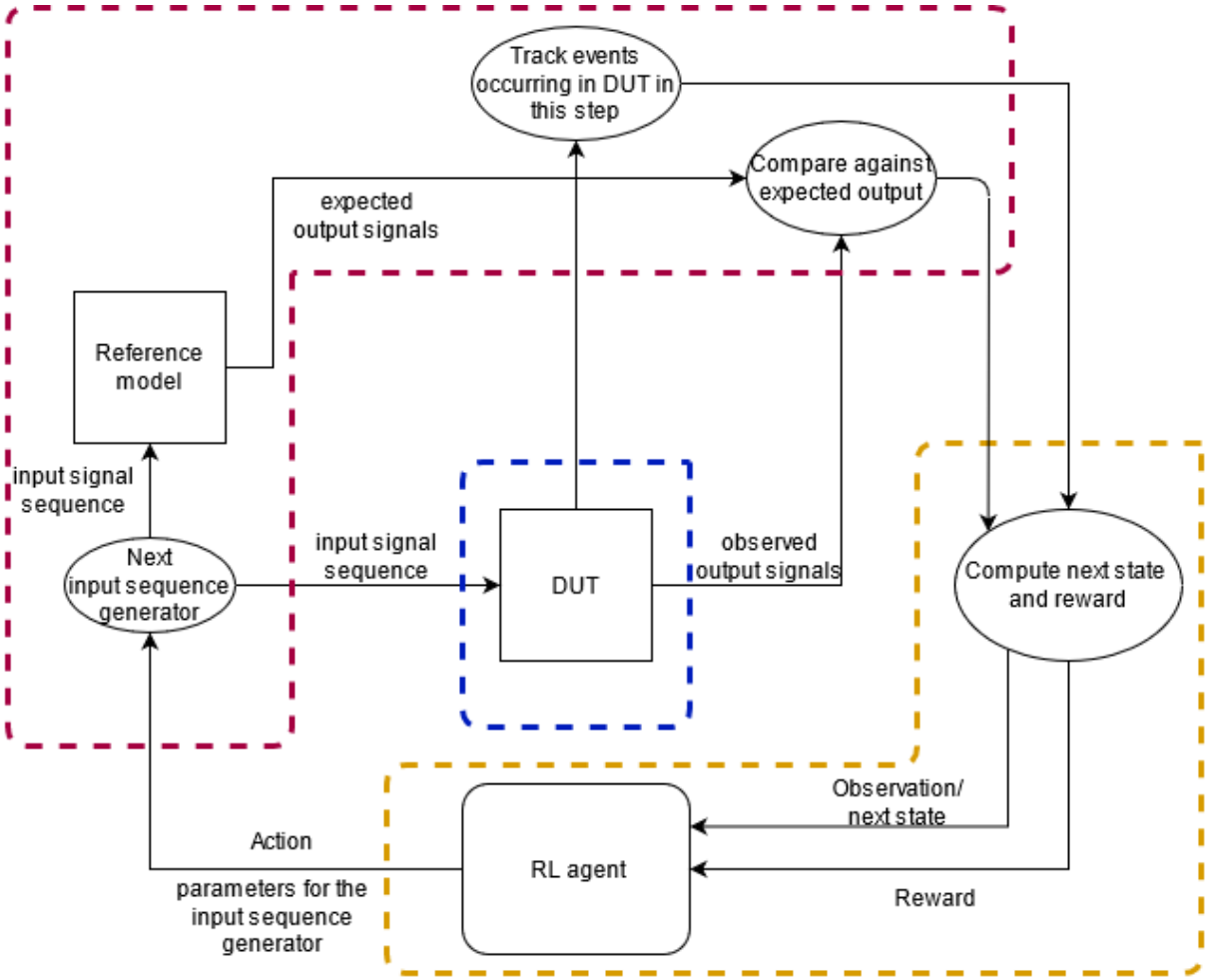


Figure 4.4: Block diagram of the complete framework with the component layers demarcated using magenta (cocotb layer), blue (Verilog layer) and orange (RL layer)

Table 4.1: Mapping of the framework functions to their respective triggers

Cocotb layer function	**Trigger**
setup_rl_episode()	*reset()* of Gym environment
rl_step()	*step()* of Gym environment
terminate_rl_episode()	End of 1 episode

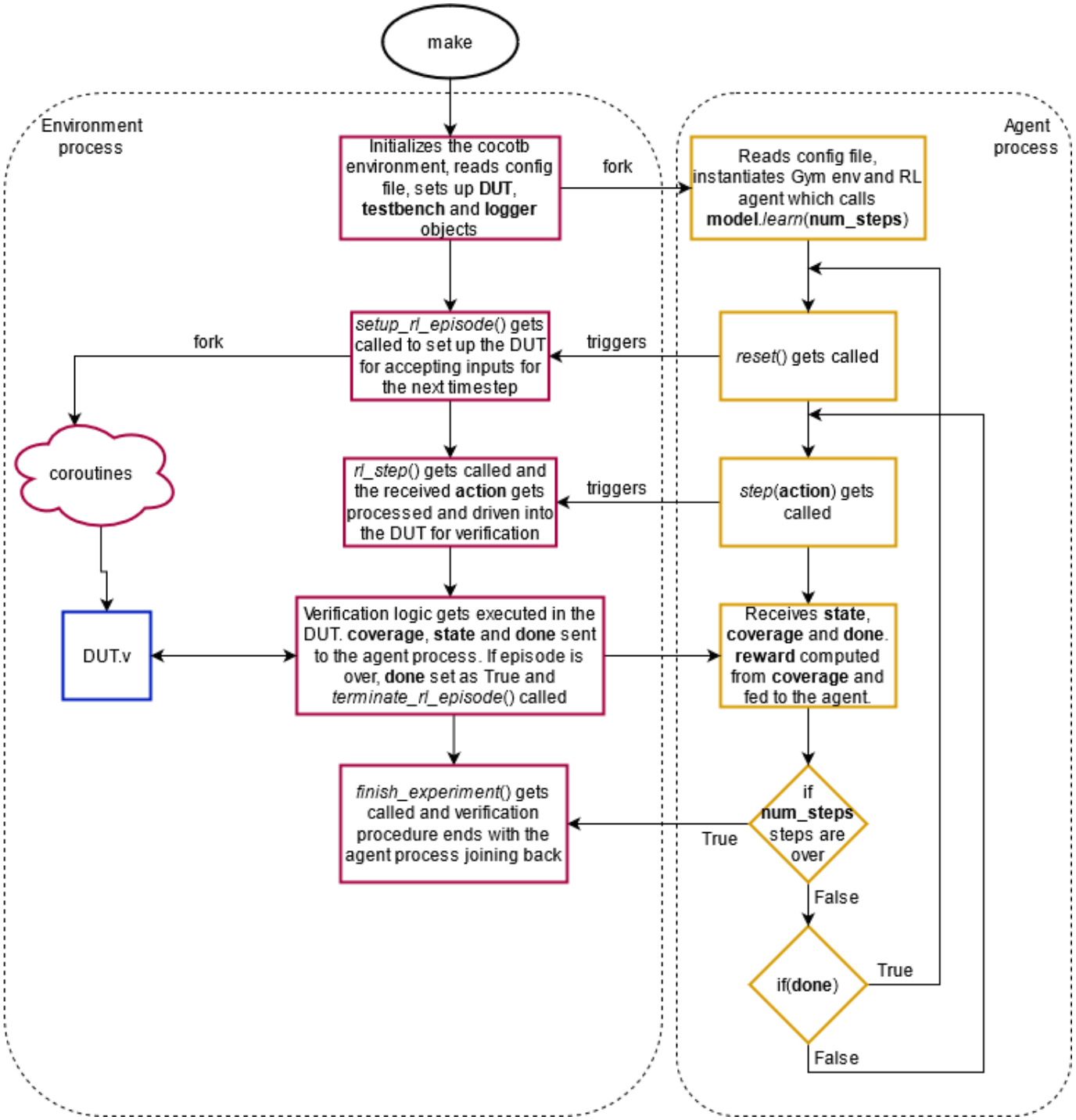


Figure 4.5: Software architecture of the framework

Building a verification testbench using this framework requires formalizing the problem as an MDP and implementing the verification logic for the DUT as discussed in the previous sections. The MDP formalization starts with defining what each timestep(step) of an RL episode involves in terms of the verification logic that must be executed on the DUT.

In each such step, the RL agent chooses an action from the action space of the MDP based on some policy. The framework offers an action space that is based on probability knobs that parameterize the input stimulus to the DUT. Each individual knob results in a real number lying in the range [0, 1] and can collectively be considered as a set of independent probability values which can be used to stochastically specify the input signal sequence for that step. The user can choose the number of such knobs required in the action space and how the input sequence should be generated from these knobs. In addition to this, the user also has the option of specifying finite sets of discrete values, which need to be included as part of the action space. Such discrete sets are supported as part of the action space by mapping equal ranges of the continuous interval [0, 1], in which the knob value lies, to the values of the discrete set.

By default, the framework assumes a single step, single Markov state environment. However, the user can provide the implementation of the function which calculates the Markov state based on the internals of the DUT if the verification requires a multi-state MDP.

Identifying the functional events of interest in the DUT is the next step. These events are what is being targeted for increased functional coverage and how much each of them is covered is used to compute the reward that is given to the RL agent. The events can be tracked using coroutines in the cocotb layer of the framework.

The user can also specify the reward function that needs to be associated with the functional coverage observed in each step of the RL episode. Let e_i denote the $i + 1$-th functional event of interest that needs to be covered. For each such event that occurs during a step, an integer specified by the user is multiplied with the number of times that event occurred in the step in order to compute the contribution of that particular

event to the reward. That is, reward

$$R = \sum_{i=0}^{i=N-1} n_i \times m_i \tag{4.1}$$

where N denotes the number of events being monitored in the step, n_i denotes the number of times e_i occurred in the step and m_i denotes the user-specified integer multiplier that decides the contribution of e_i to the reward.

This manner of defining the reward function is also handy for reward shaping. Consider an event e_5 that occurs only when the events e_3 and e_4 occur together. In such a case, assigning a reward only for e_5 by setting just m_5 as positive (say, 1) might not lead to e_5 getting covered sufficiently if e_3 and e_4 themselves are rare. In such a case, a reward function where m_3 and m_4 is set as 1 and m_5 is set as 2 leads to a better-shaped reward function that has a higher chance of hitting e_5 since the events e_3 and e_4 which lead to e_5 is encouraged too.

The framework also supports a mode where conventional random verification without the RL feedback loop can be performed in order to obtain a baseline for comparison with the methods involving RL.

CHAPTER 5

EXPERIMENTS

The objectives of the following experiments involved testing the usability of the framework when applied to different hardware designs as well as studying the improvements when using RL. The hardware designs were written in Bluespec SystemVerilog and the testbenches were built using the framework developed.

The verification experiment was first done without using RL for 1000 iterations to obtain a baseline based on traditional random verification. Subsequently, the verification experiment was performed after enabling the RL feedback for the same number of iterations. To demonstrate the advantage of using RL, the experiments attempted to increase the functional coverage of certain events of interest which occurred relatively less frequently in the DUT during simulation under a conventional random verification run. The N functional events tracked as part of coverage, in each of the following experiments, are denoted by $e_0, e_1, e_2, ..., e_{N-1}$. Thus when using RL, the objective is to increase the number of times a certain e_i is hit. This is a useful goal as the states of the DUT which are hard to hit and thus constitute corner cases in conventional verification can be made easier to hit by increasing the coverage of the events that in turn lead to those corner cases.

The algorithm used by the RL agent during the experiments was Soft Actor Critic. SAC optimizes a stochastic policy in an off-policy way and is trained to maximize a tradeoff between expected reward and entropy, a measure of randomness in the policy. The verification task was modeled as an episodic task comprising a single step. Each RL step involved simulating the DUT and reference model with some inputs. The action space involved different parameters that control these inputs that were supplied to the DUT while the Markov state space consisted only of a single unchanging state throughout. The RL agent explored the action space by perturbing the last chosen action with normal noise sampled from $\mathcal{N}(0, 0.1)$.

5.1 RLE compressor

The RLE compressor design takes in a sequence of natural numbers and compresses it using run length encoding. Thus, it attempts to reduce the length of the sequence by representing the consecutive zeros in the input sequence using their count. The design involved two design configuration parameters count_width and word_width which defined the number of bits used to represent each zero count and each non-zero element respectively. The internal working and details of the component registers of the compressor are omitted.

Since the execution logic of the DUT is dependent on the number of zeros in the sequence that is to be compressed, the ability to generate this sequence was given to the RL agent by making the probability of each element of the sequence being zero as a parameter in the action space of the agent. In addition to this, since the design configuration parameter count_width, as well as the length of the sequence that needs to be compressed, can affect the coverage, they are added to the action space as well. The action space thus becomes the cross product $[0, 1] \times \{1, 2, ..., 8\} \times \{100, 200, ..., 1000\}$. Each action from this space can be represented using three components, the first of which will be used as the generator probability knob for the sequence while the second will be used as the value for the parameter count_width and the third will be used as the length of the sequence. For example, if the action that gets chosen by the agent is (0.4, 6, 300), then a sequence of 300 natural numbers will be generated, where each number will be 0 with a probability of 0.4 and the count_width parameter would be set as 6. The word_width parameter was set constant as 4 throughout this experiment.

During the verification process, there were 4 events of interest tracked as part of the functional coverage. The exact conditions associated with these events in the hardware are not important for the results, but are provided in the second column of Table 5.1 for the sake of completeness.

The results presented below involve comparisons against a baseline involving conventional random verification with the same input space and parameterization as the RL experiment. e_3 was the functional event which was targeted to test if the coverage could be increased using RL. Therefore, the reward multiplier m_3 associated with it was set as 1 while the other multipliers were set to 0 as shown in Table 5.1. Thus a positive

Table 5.1: Functional events tracked in RLE compressor

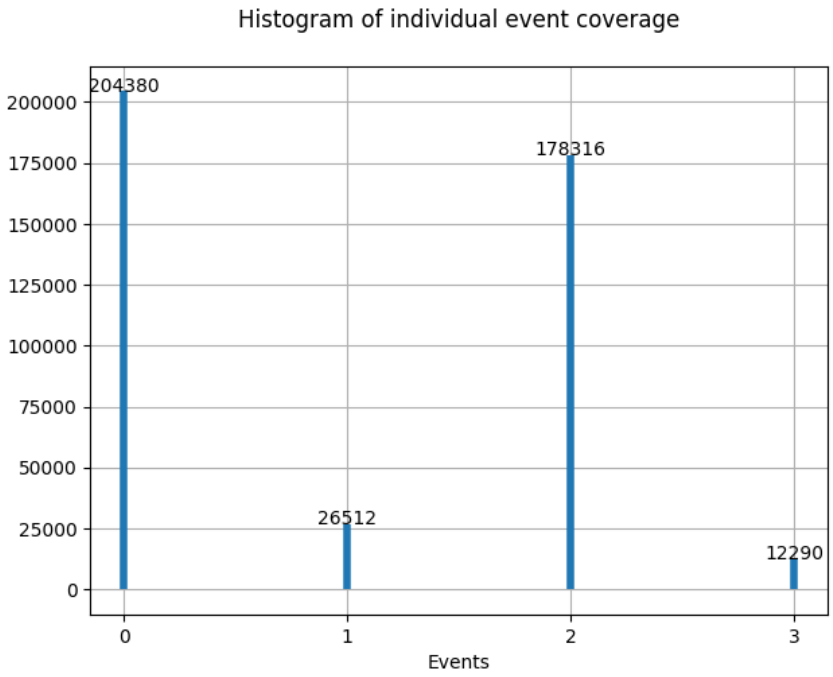
Functional event (e_i)	**Condition**	**Reward multiplier**(m_i)
e_0	rg_word_counter == 16	0
e_1	rg_zero_counter == 64	0
e_2	rg_counter == ($2^{\text{count\_width}} - 2$)	0
e_3	rg_next_count != 0	1

reward was given to the RL agent for each clock cycle in which the event e_3 was hit.

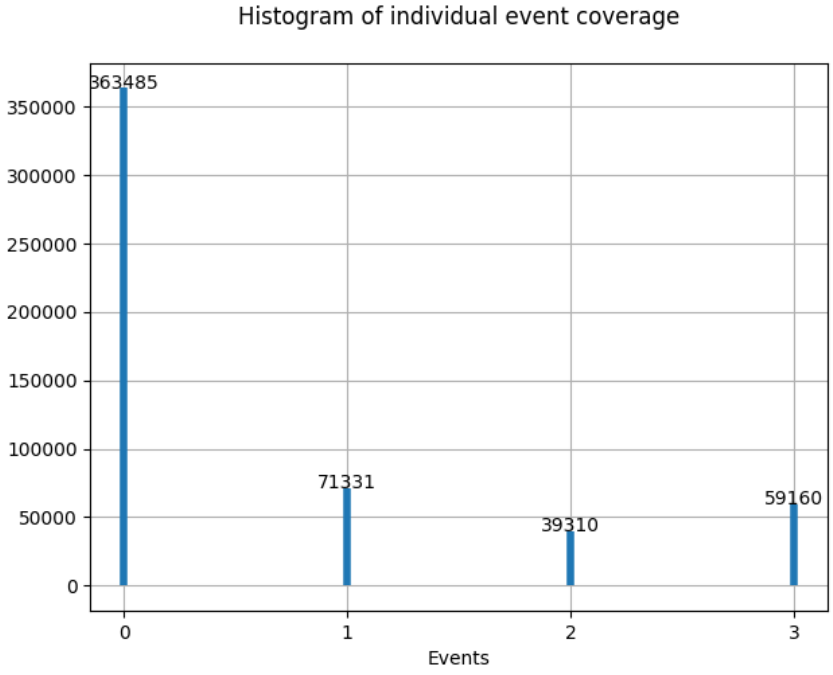
Figure 5.1 shows the comparison between functional coverage involving the 4 events tracked in both the cases. Event e_3 occurred 12,290 times in the random verification baseline as shown in Figure 5.1a whereas when RL feedback was used, its coverage increased to 59,160 times as shown in Figure 5.1b. The event e_3 happens only when a special case in the design logic gets exercised. Therefore the increase in its coverage gives more confidence to the correctness of that logic.

The increase in functional coverage observed was due to the RL agent favoring certain actions from the action space based on the reward that was received. Figures 5.2, 5.3 and 5.4 show the comparison between the choices made by the RL agent for each of the three components of action. In the histograms of the action choices in the baseline, the distribution is uniform as the actions are sampled randomly from the action space whereas in the histograms of the action choices when RL feedback was provided, there is a preference to certain actions which gets learnt over time as the agent receives more training experience samples during the experiment.

Although the baseline does not involve the training of a RL agent, the reward computation is still done based on equation 4.1, for comparison. The rewards thus calculated in each step has been plotted in Figure 5.5. In Figure 5.5a, the resulting reward in each step has no notable trend since the action choices are completely random in the baseline, whereas in Figure 5.5b, the reward increases as more steps are completed and the experiment progresses. This is because the RL agent learns from the reward signal and starts favoring the actions which lead to better rewards as discussed above.

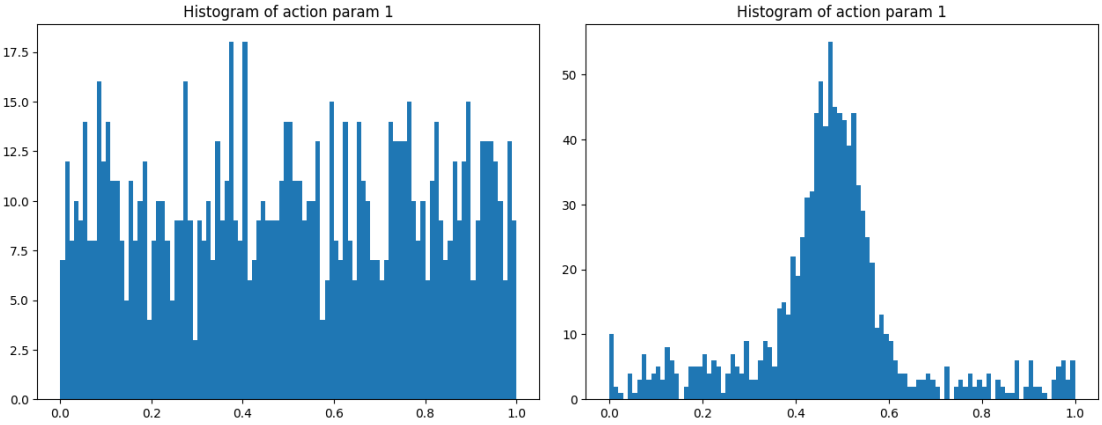


(a) Baseline without RL feedback



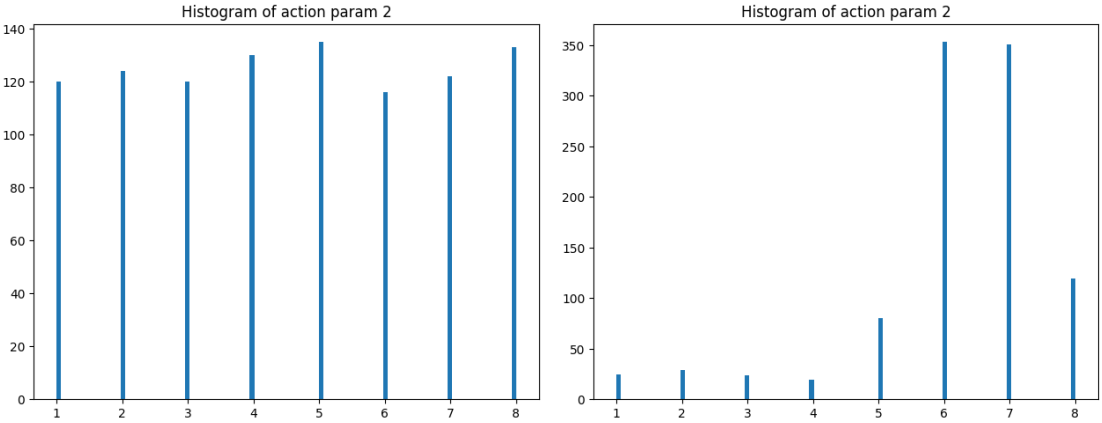
(b) With RL feedback

Figure 5.1: Comparison of functional coverage achieved with the RLE compressor after 1000 iterations. The x-axis corresponds to e_0, e_1, e_2 and e_3, the four functional events tracked as part of coverage.



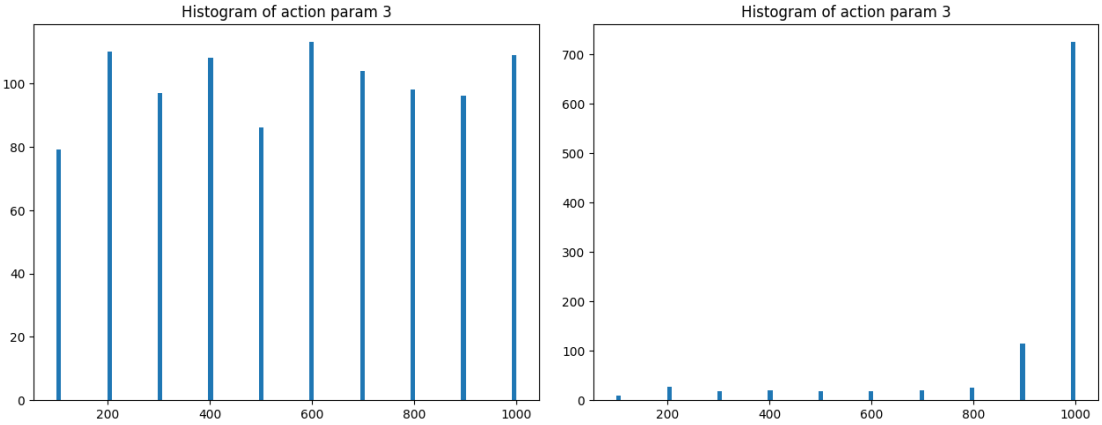
(a) Baseline without RL feedback (b) With RL feedback

Figure 5.2: Histograms of action component 1 chosen with and without RL feedback



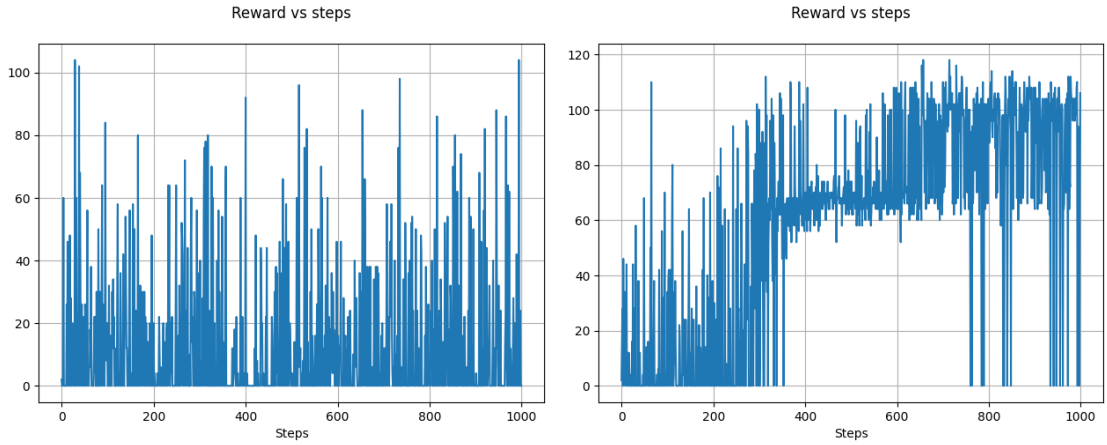
(a) Baseline without RL feedback (b) With RL feedback

Figure 5.3: Histograms of action component 2 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.4: Histograms of action component 3 chosen with and without RL feedback



(a) Baseline without RL feedback

(b) With RL feedback

Figure 5.5: Comparison of plots showing the evolution of reward as the steps progress for the RLE compressor

5.2 COO compressor

Given an input sequence, the COO compressor compresses it by denoting only the non-zero elements and their corresponding indices as part of the compressed sequence. COO is short for “co-ordinate list". The design involved two design configuration parameters word_width and index_width which defined the number of bits used to represent each non-zero word and the number of bits used to represent its corresponding index in the sequence. Both of these parameter values were chosen by the RL agent for the verification. The details of the component registers and their internal working is omitted.

The action space was the cross product $\{1, 2, ..., 8\} \times \{1, 2, ..., 8\}$ as both index_width and word_width. Thus each action from this space had two components, the first being the index_width value and the second being the word_width value. For example, if the action that gets chosen by the agent is (3, 7), then the index_width and word_width values were set as 3 and 7 respectively for the decompression operation the hardware performed. The input sequence which the DUT was supplied was 100 elements long.

There were 3 events of interest tracked as part of the functional coverage. The exact conditions associated with these events in the hardware are not important for the results, but are provided in the second column of Table 5.2 for the sake of completeness.

The results presented below involve comparisons against a baseline involving conventional random verification similar to the previous experiment. e_2 was the functional event which was targeted to test if the coverage could be increased using RL. Therefore,

Table 5.2: Functional events tracked in COO compressor

Functional event (e_i)	**Condition**	**Reward multiplier(m_i)**
e_0	rg_block_counter == 16	0
e_1	rg_block_length % 4 != 0	1
e_2	rg_next_count != 0	0

the reward multiplier m_2 associated with it was set as 1 while the other multipliers were set to 0 as shown in Table 5.2. Thus a positive reward was given to the RL agent for each clock cycle in which the event e_2 was hit.

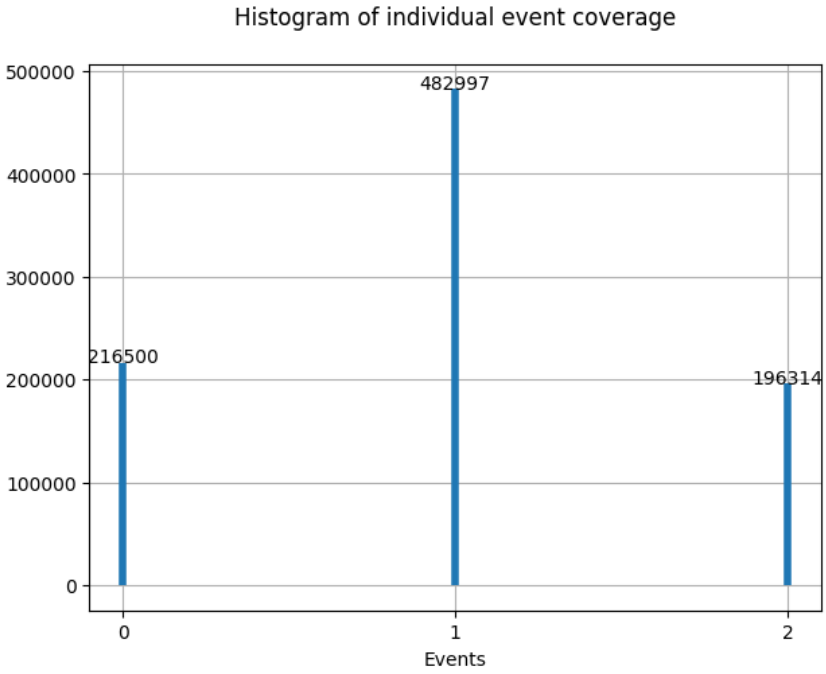
Figure 5.6 shows the comparison between functional coverage involving the 3 events tracked in both the cases. Event e_2 occurred 196,314 times in the random verification baseline as shown in Figure 5.6a whereas when RL feedback was used, its coverage increased to 364,874 times as shown in Figure 5.6b.

The comparison between the histograms of the action choices made as well as the reward plots are given below like in Section 5.1. The RL agent favoring certain actions in the action space is observable in Figures 5.7 and 5.8. The higher value of computed rewards is observable in Figure 5.9.

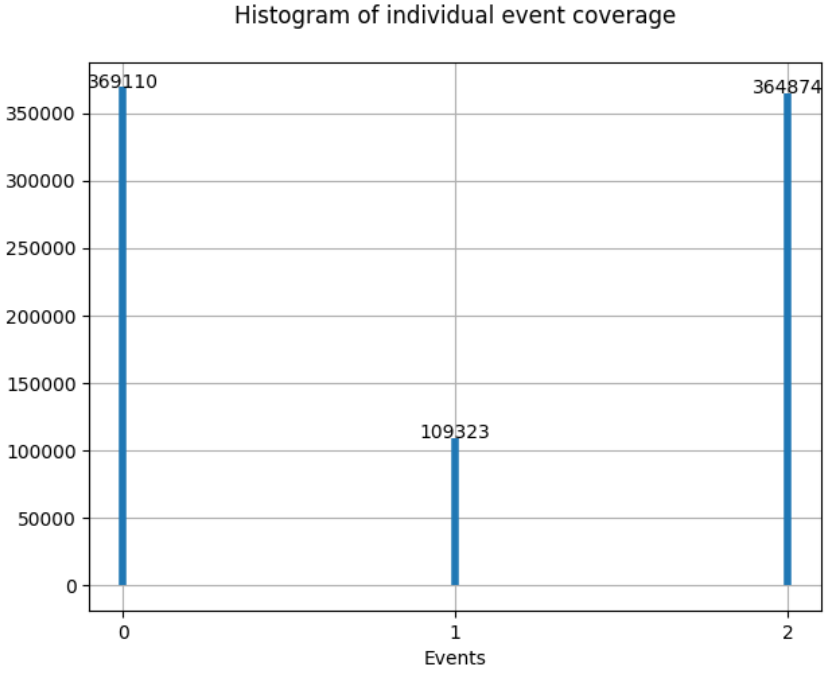
5.3 COO decompressor

Given a compressed sequence which denotes the elements of the original sequence using just the indices and values of the non-zero elements in it, the COO decompressor produces the original sequence. That is, the functionality of this design is essentially the inverse of that of the COO compressor discussed in Section 5.2. The design involved two design configuration parameters word_width and index_width which defined the number of bits used to represent each non-zero word and the number of bits used to represent its corresponding index in the sequence. Both of these parameter values were chosen by the RL agent for the verification. The details of the component registers and their internal working is omitted.

The action space was the cross product $\{1, 2, ..., 8\} \times \{1, 2, ..., 8\}$ as both index_width and word_width. Thus each action from this space had two components, the first being the index_width value and the second being the word_width value. For example, if the

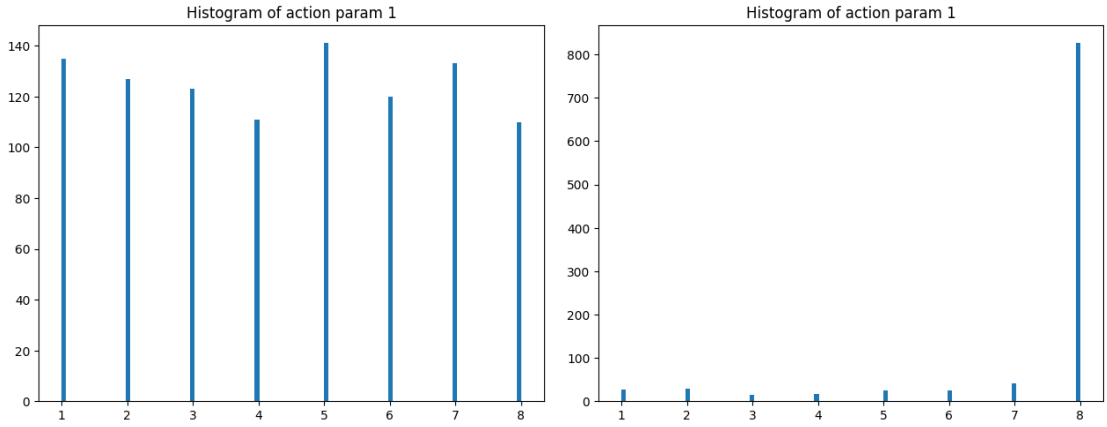


(a) Baseline without RL feedback



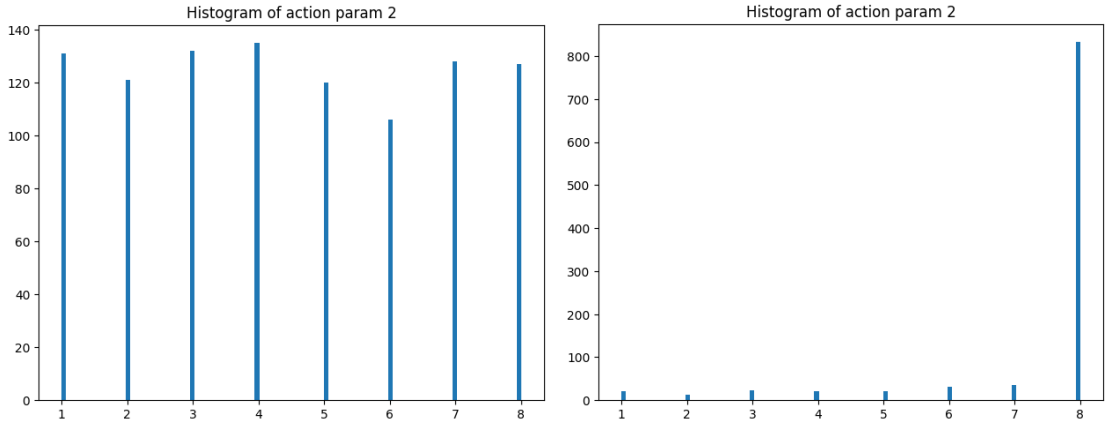
(b) With RL feedback

Figure 5.6: Comparison of functional coverage achieved with the COO compressor after 1000 iterations. The x-axis corresponds to $e_0, e_1,$ and e_2, the three functional events tracked as part of coverage.



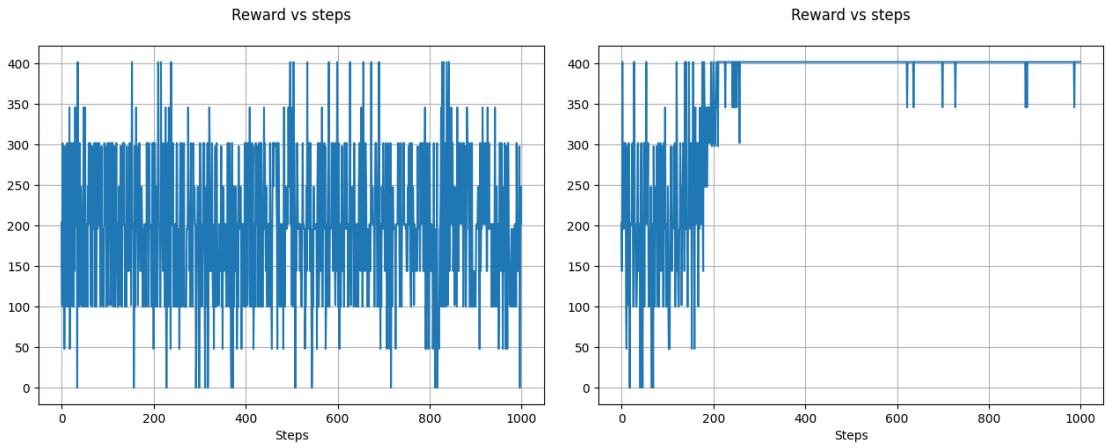
(a) Baseline without RL feedback (b) With RL feedback

Figure 5.7: Histograms of action component 1 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.8: Histograms of action component 2 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.9: Comparison of plots showing the evolution of reward as the steps progress for the COO compressor

action that gets chosen by the agent is (4, 6), then the index_width and word_width values were set as 4 and 6 respectively for the decompression operation the hardware performed. The compressed sequence which the DUT was supplied was 20 elements long.

There were 3 events of interest tracked as part of the functional coverage. The exact conditions associated with these events in the hardware are not important for the results, but are provided in the second column of Table 5.3 for the sake of completeness.

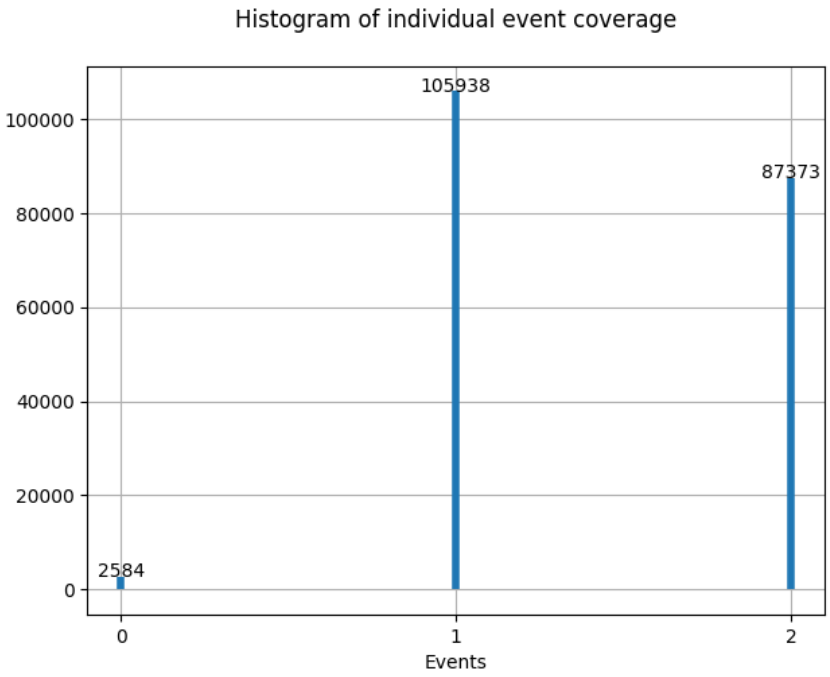
Table 5.3: Functional events tracked in COO decompressor

Functional event (e_i)	**Condition**	**Reward multiplier(m_i)**
e_0	rg_block_counter == 16	0
e_1	(rg_word_width + rg_index_width) %4 != 0	1
e_2	rg_next_count != 0	0

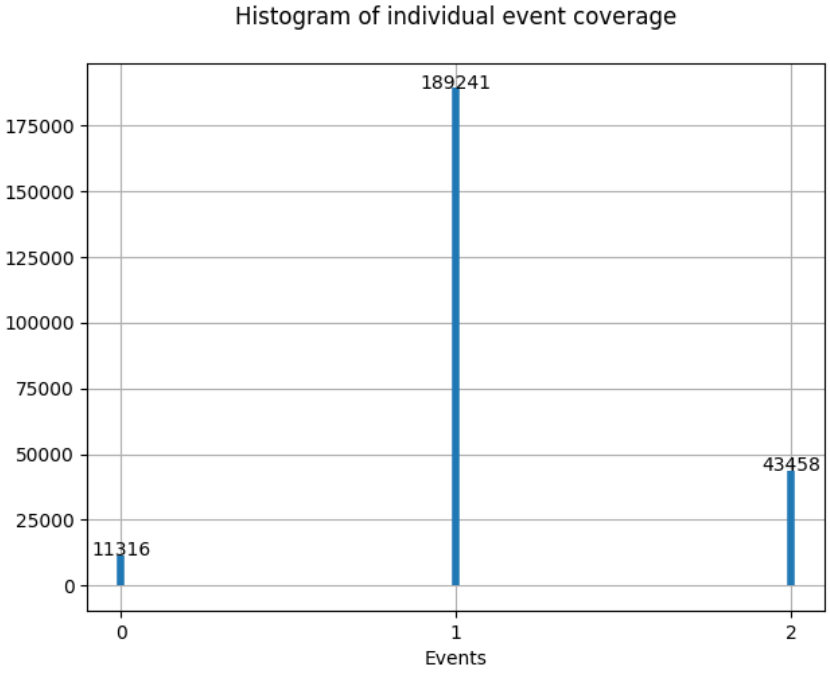
The results presented below involve comparisons against a baseline involving conventional random verification similar to the previous experiments. e_1 was the functional event which was targeted to test if the coverage could be increased using RL. Therefore, the reward multiplier m_1 associated with it was set as 1 while the other multipliers were set to 0 as shown in Table 5.3. Thus a positive reward was given to the RL agent for each clock cycle in which the event e_1 was hit.

Figure 5.10 shows the comparison between functional coverage involving the 3 events tracked in both the cases. Event e_1 occurred 105,938 times in the random verification baseline as shown in Figure 5.10a whereas when RL feedback was used, its coverage increased to 189,241 times as shown in Figure 5.10b.

The comparison between the histograms of the action choices made as well as the reward plots are given below like in the previous sections. The RL agent favoring certain actions in the action space is observable in Figures 5.11 and 5.12. The higher value of computed rewards is observable in Figure 5.13.

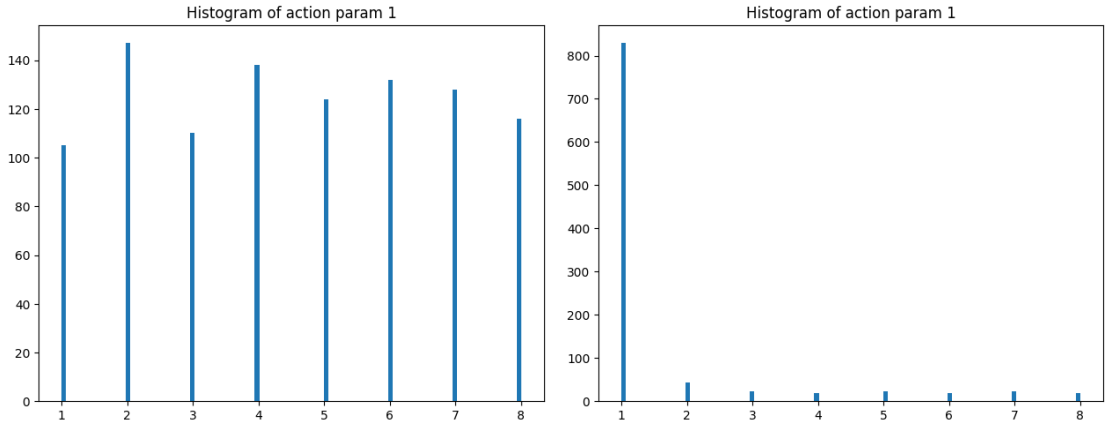


(a) Baseline without RL feedback



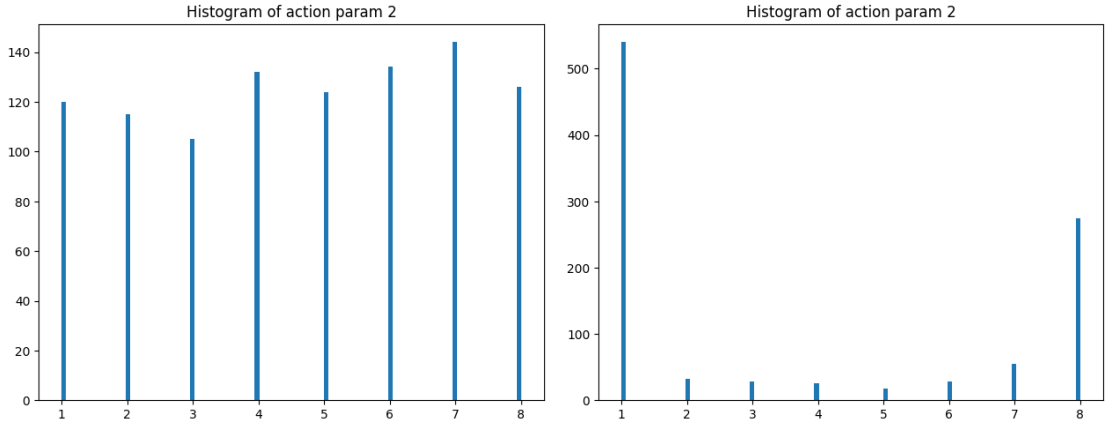
(b) With RL feedback

Figure 5.10: Comparison of functional coverage achieved with the COO decompressor after 1000 iterations. The x-axis corresponds to e_0, e_1, and e_2, the three functional events tracked as part of coverage.



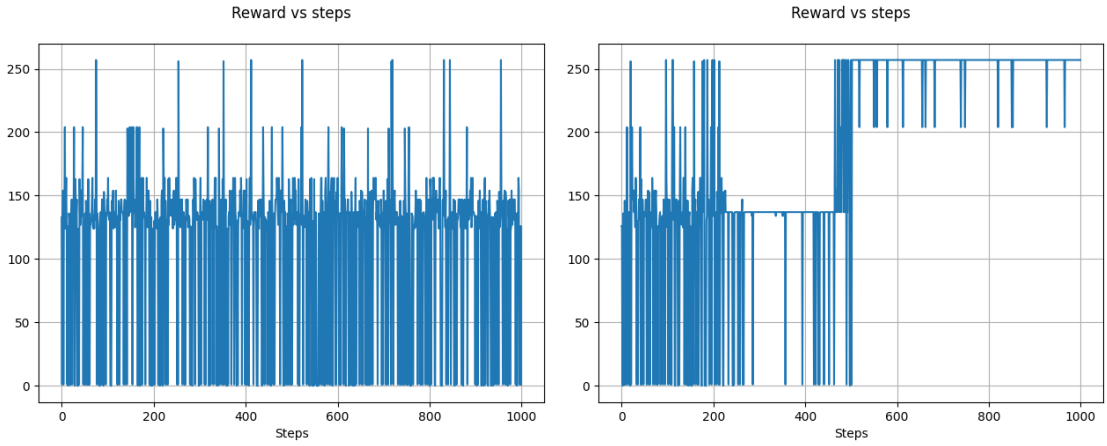
(a) Baseline without RL feedback (b) With RL feedback

Figure 5.11: Histograms of action component 1 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.12: Histograms of action component 2 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.13: Comparison of plots showing the evolution of reward as the steps progress for the COO decompressor

5.4 RLE decompressor

Given an input sequence which is compressed using run length encoding, the RLE decompressor produces the original sequence. The functionality of this design is thus the inverse of that of the RLE compressor from Section 5.1. The design involved two design configuration parameters word_width and count_width which defined the number of bits used to represent each zero count and each non-zero element respectively. The internal working and details of the component registers of the compressor are omitted.

The verification logic started with generating a sequence of length 400 and compressing it using run length encoding with the values of word_width and count_width as 4 and 6 respectively. The probability of each element of this sequence being 0 is

The action space of the RL agent was the interval $[0, 1]$ and the value chosen, say p where $p \in [0, 1]$ was used to generate a sequence where each element of the sequence was 0 with p probability. The length of this sequence was set as 400. The sequence was then compressing using run length encoding with the values of word_width and count_width as 4 and 6 respectively. It was this compressed sequence that was supplied as input to the RLE decompressor for its verification test.

There were 7 events of interest tracked as part of the functional coverage. The exact conditions associated with these events in the hardware are not important for the results, but are provided in the second column of Table 5.4 for the sake of completeness.

Table 5.4: Functional events tracked in RLE decompressor

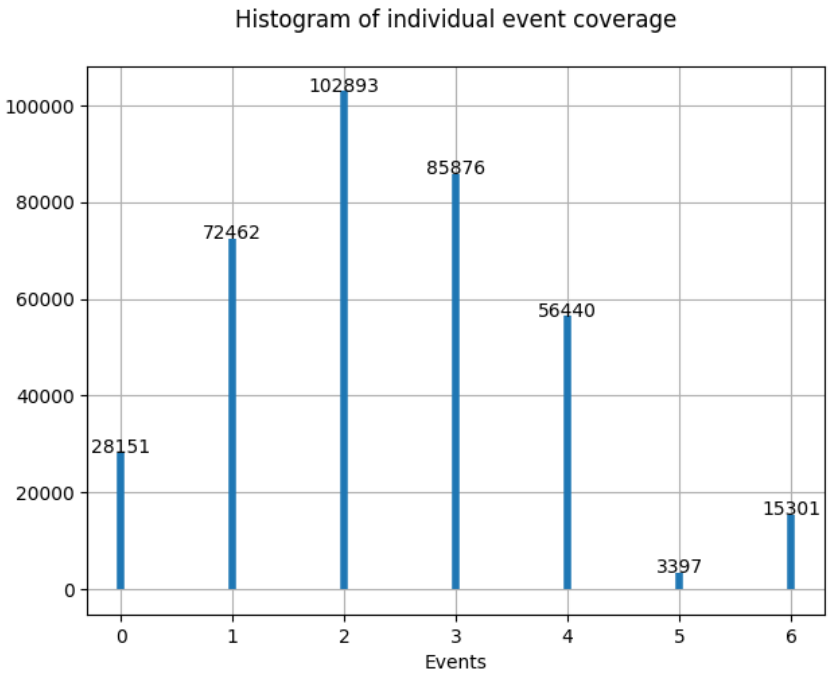
Functional event (e_i)	**Condition**	**Reward multiplier**(m_i)
e_0	rg_count_valid == 0	0
e_1	rg_word_valid == 0	0
e_2	rg_counter_valid == 0	0
e_3	rg_zero_counter == 64	0
e_4	rg_word_counter == 16	0
e_5	rg_counter == 2**count_width - 2	1
e_6	rg_next_count != 0	0

The results presented below involve comparisons against a baseline involving conventional random verification similar to the previous experiment. e_5 was the functional event which was targeted to test if the coverage could be increased using RL. Therefore,

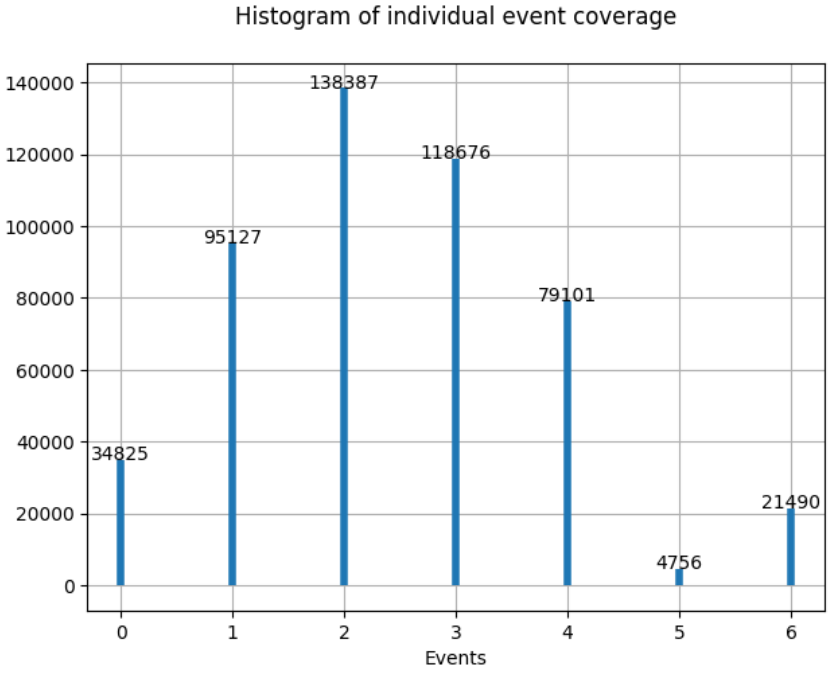
the reward multiplier e_5 associated with it was set as 1 while the other multipliers were set to 0 as shown in Table 5.4. Thus a positive reward was given to the RL agent for each clock cycle in which the event e_5 was hit.

Figure 5.14 shows the comparison between functional coverage involving the 3 events tracked in both the cases. Event e_5 occurred 3,397 times in the random verification baseline as shown in Figure 5.14a whereas when RL feedback was used, its coverage increased to 4,756 times as shown in Figure 5.14b.

The comparison between the histograms of the action choices made as well as the reward plots are given below like in Section 5.1. The RL agent favoring certain actions in the action space is observable in Figure 5.15. The higher value of computed rewards is observable in Figure 5.16.

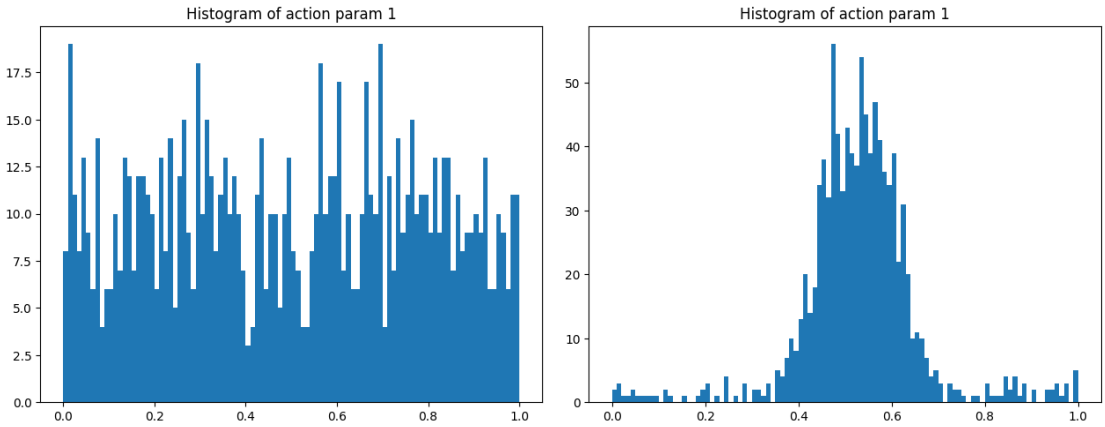


(a) Baseline without RL feedback



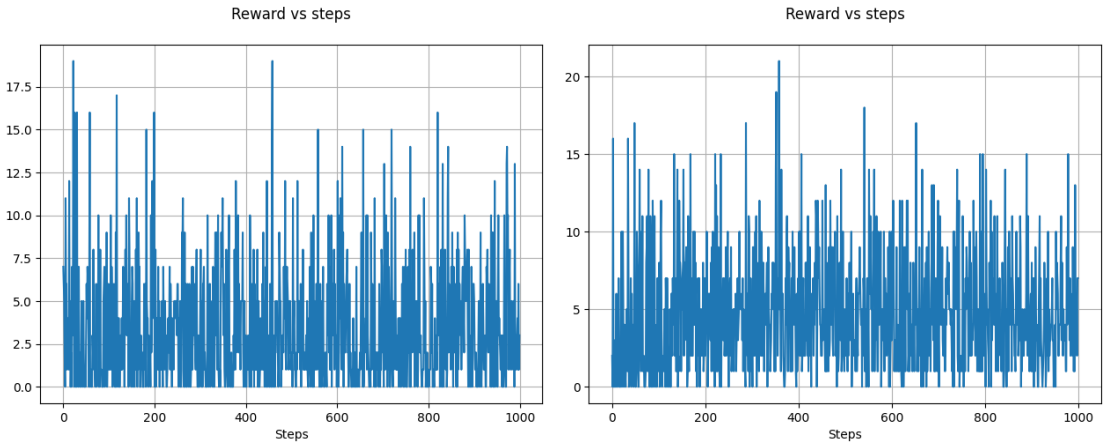
(b) With RL feedback

Figure 5.14: Comparison of functional coverage achieved with the RLE decompressor after 1000 iterations. The x-axis corresponds to e_0, e_1, ..., e_6, the seven functional events tracked as part of coverage.



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.15: Histograms of action component 1 chosen with and without RL feedback



(a) Baseline without RL feedback (b) With RL feedback

Figure 5.16: Comparison of plots showing the evolution of reward as the steps progress for the RLE decompressor

CHAPTER 6

FUTURE WORK

Although the inclusion of RL into the hardware verification procedure is believed to be a valuable step in the right direction, there are still challenges that need to be addressed. The exploration vs exploitation tradeoff plays a significant role in the context of applying RL to hardware verification. With setups involving just a single Markov state as part of the state space (like in the experiments presented in Chapter 5), hitting coverage holes are difficult. The rewards assigned to reaching these coverage holes could be high but since by definition coverage holes are difficult to reach by exploration, these rewards will be exceedingly sparse leading to no significant learning for the RL agent. The most natural way of tackling this challenge involves expanding the Markov state space and using more sophisticated RL algorithms that are better suited for sparse reward environments.

Even for moderately complex designs, if an MDP with multiple states is required, it involves carefully defining what each Markov state represents in terms of the DUT elements. Intrinsic reward signals like Curiosity [Pathak *et al.* (2017)] and count-based exploration techniques involving hashing [Tang *et al.* (2017)] are possibilities for aiding the agent to explore when the MDP state space grows more, whereas the discrete action space growing too large can be handled by the method suggested by Dulac-Arnold *et al.* (2016).

Another idea that is worth pursuing to target coverage holes during verification involves Hindsight Experience Replay (HER) [Andrychowicz *et al.* (2018)]. The coverage holes serve as the goal states that need to be reached by the agent using hindsight in such a setup. The recent work by Lee and Moon (2021) suggests an approach where HER is used with SAC, the RL algorithm used in this project.

The open-source framework developed as part of this work was found to be viable for building RL-based verification setups for various digital hardware designs without significant overhead in terms of the engineering effort required. The interfacing between the cocotb and the RL components of the framework is seamless, allowing the

user to implement any verification logic that is possible in traditional verification setups. It also serves as a starting point for integrating more sophisticated RL ideas mentioned above into the hardware verification procedure and experimenting with them.

REFERENCES

1. **Andrychowicz, M.**, **F. Wolski**, **A. Ray**, **J. Schneider**, **R. Fong**, **P. Welinder**, **B. McGrew**, **J. Tobin**, **P. Abbeel**, and **W. Zaremba** (2018). Hindsight experience replay.

2. **Arulkumaran, K.**, **M. P. Deisenroth**, **M. Brundage**, and **A. A. Bharath** (2017). A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.

3. **Boyan, J. A.** and **M. L. Littman**, Packet routing in dynamically changing networks: A reinforcement learning approach. *In Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

4. **Brockman, G.**, **V. Cheung**, **L. Pettersson**, **J. Schneider**, **J. Schulman**, **J. Tang**, and **W. Zaremba** (2016). Openai gym.

5. **Böttinger, K.**, **P. Godefroid**, and **R. Singh** (2018). Deep reinforcement fuzzing.

6. **Chen, Y.-R.**, **A. Rezapour**, **W.-G. Tzeng**, and **S.-C. Tsai** (2020). Rl-routing: An sdn routing algorithm based on deep reinforcement learning. *IEEE Transactions on Network Science and Engineering*, **7**(4), 3185–3199.

7. **Dulac-Arnold, G.**, **R. Evans**, **H. van Hasselt**, **P. Sunehag**, **T. Lillicrap**, **J. Hunt**, **T. Mann**, **T. Weber**, **T. Degris**, and **B. Coppin** (2016). Deep reinforcement learning in large discrete action spaces.

8. **Haarnoja, T.**, **A. Zhou**, **P. Abbeel**, and **S. Levine** (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.

9. **Higgs, C.** and **S. Hodgeson** (2013). cocotb. URL `https://github.com/cocotb/cocotb`.

10. **Hughes, W.**, **S. Srinivasan**, **R. Suvarna**, and **M. Kulkarni** (2019). Optimizing design verification using machine learning: Doing better than random.

11. **Ioannides, C.** and **K. I. Eder** (2012). Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, **17**(1). ISSN 1084-4309. URL `https://doi.org/10.1145/2071356.2071363`.

12. **Lee, M. H.** and **J. Moon** (2021). Deep reinforcement learning-based uav navigation and control: A soft actor-critic with hindsight experience replay approach.

13. **Li, Y.** (2018). Deep reinforcement learning: An overview.

14. **Lillicrap, T. P.**, **J. J. Hunt**, **A. Pritzel**, **N. Heess**, **T. Erez**, **Y. Tassa**, **D. Silver**, and **D. Wierstra** (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

15. **Liu, N.**, **Z. Li**, **Z. Xu**, **J. Xu**, **S. Lin**, **Q. Qiu**, **J. Tang**, and **Y. Wang** (2017). A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning.

16. **Mammeri, Z.** (2019). Reinforcement learning based routing in networks: Review and classification of approaches. *IEEE Access*, **7**, 55916–55950.

17. **Mao, H.**, **M. Alizadeh**, **I. Menache**, and **S. Kandula**, Resource management with deep reinforcement learning. *In Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16. Association for Computing Machinery, New York, NY, USA, 2016. ISBN 9781450346610. URL https://doi.org/10.1145/3005745.3005750.

18. **Mirhoseini, A.**, **A. Goldie**, **M. Yazgan**, **J. Jiang**, **E. Songhori**, **S. Wang**, **Y.-J. Lee**, **E. Johnson**, **O. Pathak**, **S. Bae**, **A. Nazi**, **J. Pak**, **A. Tong**, **K. Srinivasa**, **W. Hang**, **E. Tuncer**, **A. Babu**, **Q. V. Le**, **J. Laudon**, **R. Ho**, **R. Carpenter**, and **J. Dean** (2020). Chip placement with deep reinforcement learning.

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

20. **Mnih, V.**, **K. Kavukcuoglu**, **D. Silver**, **A. A. Rusu**, **J. Veness**, **M. G. Bellemare**, **A. Graves**, **M. Riedmiller**, **A. K. Fidjeland**, **G. Ostrovski**, *et al.* (2015). Human-level control through deep reinforcement learning. *nature*, **518**(7540), 529–533.

21. **Pathak, D.**, **P. Agrawal**, **A. A. Efros**, and **T. Darrell** (2017). Curiosity-driven exploration by self-supervised prediction.

22. **Raffin, A.** (2020). Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo.

23. **Raffin, A.**, **A. Hill**, **M. Ernestus**, **A. Gleave**, **A. Kanervisto**, and **N. Dormann** (2019). Stable baselines3. https://github.com/DLR-RM/stable-baselines3.

24. **Schulman, J.**, **S. Levine**, **P. Moritz**, **M. I. Jordan**, and **P. Abbeel** (2017*a*). Trust region policy optimization.

25. **Schulman, J.**, **F. Wolski**, **P. Dhariwal**, **A. Radford**, and **O. Klimov** (2017*b*). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

26. **Silver, D.**, **A. Huang**, **C. J. Maddison**, **A. Guez**, **L. Sifre**, **G. Van Den Driessche**, **J. Schrittwieser**, **I. Antonoglou**, **V. Panneershelvam**, **M. Lanctot**, *et al.* (2016). Mastering the game of go with deep neural networks and tree search. *nature*, **529**(7587), 484–489.

27. **Silver, D.**, **T. Hubert**, **J. Schrittwieser**, **I. Antonoglou**, **M. Lai**, **A. Guez**, **M. Lanctot**, **L. Sifre**, **D. Kumaran**, **T. Graepel**, **T. Lillicrap**, **K. Simonyan**, and **D. Hassabis** (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

28. **Singh, K.**, **R. Gupta**, **V. Gupta**, **A. Fayyazi**, **M. Pedram**, and **S. Nazarian**, A hybrid framework for functional verification using reinforcement learning and deep learning. *In Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362528. URL https://doi.org/10.1145/3299874.3318039.

29. **Snyder, W.** (2001). Verilator. URL https://veripool.org/verilator/.

30. **Sutton, R. S.** and **A. G. Barto**, *Reinforcement learning: An introduction*. MIT press, 2018.

31. **Tang, H.**, **R. Houthooft**, **D. Foote**, **A. Stooke**, **X. Chen**, **Y. Duan**, **J. Schulman**, **F. D. Turck**, and **P. Abbeel** (2017). #exploration: A study of count-based exploration for deep reinforcement learning.

32. **Vinyals, O.**, **I. Babuschkin**, **W. M. Czarnecki**, **M. Mathieu**, **A. Dudzik**, **J. Chung**, **D. H. Choi**, **R. Powell**, **T. Ewalds**, **P. Georgiev**, *et al.* (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, **575**(7782), 350–354.

33. **Wang, F.**, **H. Zhu**, **P. Popli**, **Y. Xiao**, **P. Bodgan**, and **S. Nazarian**, Accelerating coverage directed test generation for functional verification: A neural network-based framework. *In Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450357241. URL `https://doi.org/10.1145/3194554.3194561`.

34. **Watkins, C. J.** and **P. Dayan** (1992). Q-learning. *Machine learning*, **8**(3-4), 279–292.

35. **Ye, Y.**, **X. Ren**, **J. Wang**, **L. Xu**, **W. Guo**, **W. Huang**, and **W. Tian** (2018). A new approach for resource scheduling with deep reinforcement learning.