EasyChair Preprint

№ XXX

Efficient Stimuli Generation using Reinforcement Learning in Design Verification

Deepak Narayan Gadde, Thomas Nalapat, Aman Kumar,
Djones Lettnin, Wolfgang Kunz and Sebastian Simon

May 23, 2024

Efficient Stimuli Generation using Reinforcement Learning in Design Verification

Deepak Narayan Gadde[1], Thomas Nalapat[1], Aman Kumar[1],
Djones Lettnin[2], Wolfgang Kunz[3], Sebastian Simon[1]
[1]Infineon Technologies Dresden GmbH & Co. KG, Germany
[2]Infineon Technologies AG, Germany
[3]Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Germany

***Abstract*—The increasing design complexity of System-on-Chips (SoCs) has led to significant verification challenges, particularly in meeting coverage targets within a timely manner. At present, coverage closure is heavily dependent on constrained random and coverage driven verification methodologies where the randomized stimuli are bounded to verify certain scenarios and to reach coverage goals. This process is said to be exhaustive and to consume a lot of project time. In this paper, a novel methodology is proposed to generate efficient stimuli with the help of Reinforcement Learning (RL) to reach the maximum code coverage of the Design Under Verification (DUV). Additionally, an automated framework is created using metamodeling to generate a SystemVerilog testbench and an RL environment for any given design. The proposed approach is applied to various designs and the produced results proves that the RL agent provides effective stimuli to achieve code coverage faster in comparison with baseline random simulations. Furthermore, various RL agents and reward schemes are analyzed in our work.**

***Index Terms*—Reinforcement Learning, Design Verification, Coverage, Metamodeling**

I. Introduction

Due to the latest advancements in semiconductor technology, it is feasible to include various functionalities and features in a single SoC. In the SoC development, design verification continues to be one of the most expensive and time-consuming stages. A recent study done by the Wilson Research Group states that verification consumes around $60\,\%$ of the overall project time [1]. Hence, it is evident that the rising complexity of hardware designs necessitates the development of new approaches and methodologies which can provide verification engineers with the ability to fulfill their objectives faster and with minimal resources.

The simulation-based design verification is a well-established and powerful technique which utilizes Constrained Random Verification (CRV) and coverage-driven methodologies [2]. Although it enhances the verification process, reaching coverage targets remains a greater challenge and is required for verification closure. This is primarily due to the demand for manual involvement of verification engineers adjusting constraints inside the testbench based on previous random simulations to drive the stimuli to reach the ultimate coverage objective.

This work has been developed in the project VE-VIDES (project label 16ME0243K) which is partly funded within the Research Programme ICT 2020 by the German Federal Ministry of Education and Research (BMBF)

Numerous initiatives have been made to optimize design verification through the use of Machine Learning (ML) techniques. These are reviewed in [3], [4], and [5]. Most of these studies focused on functional coverage improvement, simulation speedup, and reducing the test count. This paper presents a novel approach to addressing the coverage closure problem in constrained random simulations by formulating it as an RL task. Furthermore, it addresses how RL could produce a better stimulus in comparison with a typical random stimulus during the simulation to achieve target code coverage of a given DUV. This study aims to promote the adoption of automation methodologies that utilize metamodeling and RL by showcasing the performance and comparing various RL models and reward schemes employed in the approach.

The contributions of this work are as follows:

- An novel method to reach code coverage goals faster with the help of RL stimuli (Sec. IV)
- A metamodeling framework to build a generic SystemVerilog testbench for the given DUV (Sec. IV-B)
- A configurable RL environment to utilize various RL policies and to create RL actions for the given DUV (Sec. IV-D)
- The integration of the RL environment into the simulation environment (Fig. 3)

II. Background Knowledge

This section introduces the relevant techniques used in our work.

A. Reinforcement Learning

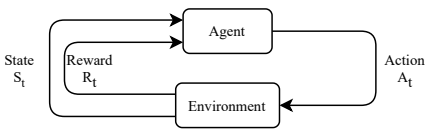

Fig. 1. RL algorithm

RL is a special type of ML algorithm and is shown in Fig. 1. Here, an agent interacts with an environment by performing certain actions A_t. For each action, the state S_t of the agent gets altered and the agent gets a reward or penalty R_t for action A_t as feedback [6]. The agent keeps performing these three tasks, which help it learn about and explore the environment around it.

The agent develops insights on which actions result in rewards and which ones lead to negative feedback or penalties. It has several core components which are defined in the following Table I along with their representation in our given coverage closure problem.

TABLE I
RL core components and their analogy in our work

RL core components	Definition	Analogy in our work
Agent	The learner or decision maker	An intelligent unit which generates a stimulus
Environment	The world with which the agent interacts	A simulator which performs RTL simulations
Action	All possible choices the agent can take	New stimuli to drive DUV
Reward	Immediate return from the environment based on certain actions	+1 or -1 or 0 depends on performance of action on environment
State	The current situation returned by the environment	Current coverage score
Target	The goal of the agent	Reach 100% coverage
Termination	The state to end the RL process	When 100% coverage is reached or after certain number of RL actions
Policy	The strategy that the agent employs to determine its actions at any state	This work explores various policies e.g., actor-critic

B. Metamodeling

Code generation is crucial for boosting chip design productivity. However, Python-based code generators can lead to inconsistencies in data structures. Metamodeling [7] helps ensure interoperability of data among various generators and promotes reusable code. It is essential for addressing inconsistencies across diverse systems with multiple viewpoints that require consistent automatic code generation from common sources. Fig. 2 shows the basic flow in metamodeling.

In metamodeling, each model (design) has a corresponding *metamodel* that defines its structure, constraints, and properties. These metamodels are usually represented as structured data sources like *XML* files. A metamodeling environment facilitates interaction with these models by providing access, creation, and transformation capabilities based on the metamodel's description. The environment typically includes an automatically generated API, which allows users to interact with model instances and their properties. This API is utilized by a Mako-based [8] template and a template engine for generating the desired code.

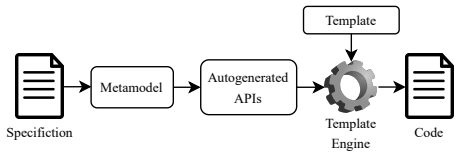

Fig. 2. Metamodeling flow

III. Related Work

Over the past two decades, there has been a significant amount of research on ML techniques to enhance the design verification process: [3], [4], and [5]. RL, a subset of ML, has also been investigated for solving various problems within the domain of verification. The first work to propose RL in verification was [9], where it was employed in combination with Supervised Learning (SL) to achieve higher functional coverage. The recent studies [10], [11], and [12] utilized various SL algorithms such as Deep Neural Network (DNN), Artificial Neural Network (ANN), and Decision Tree (DT) to significantly enhance simulation speed in comparison to random simulation approaches. Meanwhile, RL is applied in the works [13], [14], and [15] for the same purpose. Additionally, [16], [17], [18], and [5] utilize RL to reach target functional coverage more rapidly.

TABLE II
Comparision of related works

Work	Year	ML Approach	ML Model	Testbench Language	Application
[9]	2019	SL, RL	DNN, Q-Learning	SV	FCI, SS
[10]	2020	SL	ANN	Python Cocotb	SS
[19]	2020	RL	Q-Learning	-	FCI
[16]	2020	RL	Tree search, NN	-	FCI
[17]	2021	RL	Custom	-	FCI, SS
[18]	2021	RL	Soft Actor-Critic	Python Cocotb	F
[5]	2022	RL	SGTD	-	RTS
[11]	2022	SL	NN	-	FCI, SS
[12]	2022	SL	ANN, DNN, DT	SV-UVM	SS
[13]	2023	RL	DNN	SV-UVM	SS
[20]	2023	UL	Isolation forest	-	SS,TS
[14]	2023	RL	Actor-Critic	SV-UVM	FCI
[15]	2023	RL	DQN	SV	FCI
Our work	2024	RL	PPO, A2C, DQN	SV	CCI

Notes: Unsupervised Learning (UL), Simulation Speedup (SS), Functional Coverage Improvement/Closure (FCI), Code Coverage Improvement/Closure (CCI), Reaching Target State (RTS), Test Selection (TS), SystemVerilog (SV), Universal Verification Methodology (UVM)

Our work is compared with other relevant research based on various aspects to illustrate its significance as depicted in Table II. Most of the works which employed an RL approach require manual effort to configure the RL environment and to write a design-specific testbench. These works utilized specific RL algorithms in their methods and did not examine how other RL models would perform for the same application. The approach presented in this paper is design-agnostic and configurable in terms of learning policy, reward scheme, and target coverage type. Additionally, we produced the results by employing three RL algorithms in our approach, namely Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO), and Deep Q Network (DQN) from the Stable-Baselines3 library [21], to show their performance to reach maximum code coverage of the DUV. These algorithms were chosen based on their support for continuous and discrete actions. A2C and PPO support both types of action space, whereas DQN supports only the discrete type.

IV. Proposed Approach

Fig. 3 depicts the proposed framework, which begins by parsing the design Intellectual Property (IP) to extract all

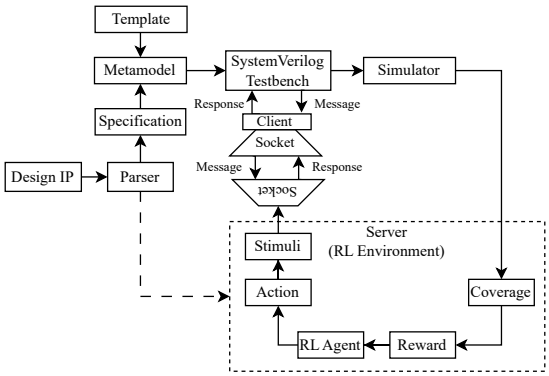

Fig. 3. RL-guided design verification

information about the primary ports. This data is used by the metamodel to generate a SystemVerilog testbench that interfaces with an RL environment implemented in Python. The simulation begins with the RL environment sending initial zero values for all ports to the testbench. Subsequently, the DUV is simulated using the values from the RL environment. Coverage for this timestep is collected using the simulator, which is then relayed back to the RL environment to calculate the reward. Based on the reward and coverage value, the RL agent selects a new action i.e., a new stimulus which will in turn be used to stimulate the DUV for another step. This process is repeated until the coverage reaches $100\,\%$ or until after a predetermined number of steps.

The following subsections provide a detailed explanation of these processes:

A. Parsing and specification formation

The Register Transfer Level (RTL) code of the given design IP, usally in VHDL, Verilog, or SystemVerilog, is parsed to extract the primary port parameters such as name, type, size, and direction of the ports. This information is used to create a standardized specification in a Extensible Markup Language (XML) format.

B. Testbench creation

The metamodel utilizes this specification and a predefined template to create a testbench in SystemVerilog. The DUV must be driven by the stimuli received from the RL agent. Direct communication with the SystemVerilog testbench is not feasible as our RL environment is implemented in Python.

This communication can be achieved by implementing a client-server application [22]. The Direct Programming Interface (DPI) [2] can be utilized to call functions written in C from the SystemVerilog testbench. Therefore, a client responsible for creating a client socket, managing handshakes, and sending the response from the server, which in this context is the RL environment, is implemented in C.

Based on a predefined template, the metamodel creates a testbench that requests stimuli from the server and drives the inputs of the DUV with the received data.

C. Simulation and coverage collection

The simulator is configured to dump coverage data after each clock cycle. The specific type of coverage (block, Finite State Machine (FSM), toggle, or expression) is determined by the settings in the configuration file. The RL environment reads the saved coverage value after each clock cycle.

D. RL environment

The RL environment based on OpenAI's Gym [23] is configured by providing the below details in the configuration file.

- *Top Module* - The design IP to be verified
- *Coverage Type* - The type of coverage (block, FSM, toggle or expression) based on which the reward is calcluated
- *Learning Policy* - RL algorithm (PPO, A2C or DQN) that guides the learning process of the RL agent
- *Ports* - The ports of the IP that contribute to the increase in coverage
- *Reward Scheme* - Optimistic or penalty based reward scheme

1) Actions: The action space is defined by the designated ports. The RL environment reads the size of these ports from the parser's output and configures the action space to encompass all possible combinations of values that the ports can assume. For instance, for an Arithmetic Logic Unit (ALU) with a 3-bit opcode, the action space would range from 0 to $2^3 - 1$, and action 6 would drive the *opcode* port of the ALU with its binary value *110*.

2) Reward: Following each clock cycle of the simulation, the RL environment reads the coverage data dumped by the simulator. If the coverage at the current step exceeds the coverage from the previous step, the reward is assigned a value of 1. If the coverage decreases, the reward is determined according to the predefined reward scheme. Algorithm 1 illustrates the reward calculation process for both optimistic and penalty-based reward schemes.

Algorithm 1: Reward Scheme used in our framework

```
if current_coverage > previous_coverage then
    reward ← 1
else if reward_scheme == penalty then
    reward ← -1
else if reward_scheme == optimistic then
    reward ← 0
end if
```

3) RL agent: The RL agent selects an action according to the specified policy. It then performs the chosen action by packaging the stimulus and port name into a message packet and sending this packet to the client (SystemVerilog testbench) upon receiving a request. Subsequently, the agent receives a reward corresponding to this action. This reward assesses the effectiveness of the action, and the agent updates its policy and chooses the next action based on this received reward. The chosen action is performed in the next step. This process continues until coverage reaches $100\,\%$ or until after a predetermined number of steps.

TABLE III
Results produced on various design IPs to reach maximum code coverage

Design IP	Max. Code Coverage (%)	#Random Stimuli	#RL Stimuli					
			PPO (Optimistic)	PPO (Penalty)	A2C (Optimistic)	A2C (Penalty)	DQN (Optimistic)	DQN (Penalty)
JTAG TAP [24]	94.39	1699	1373	545	627	1225	493	975
ALU	90.91	22	12	8	16	22	27	20
CORDIC [25]	99.72	124	95	55	105	63	60	96
RISC-V [25]	84.58	330	226	291	506	535	734	271
FIR [24]	100	11	6	10	22	11	10	6
FIFO [24]	100	23	22	23	22	22	22	31

V. Results

In this section, we present the results obtained by applying stimuli generated by the RL agent to various designs. Subsequently, we compare the number of steps required to achieve maximum code coverage using stimuli generated randomly and those generated by RL agents based on different policies.

Table III presents a comparison of the number of stimuli generated randomly versus those produced by RL agents with various policies and reward schemes to reach maximum code coverage across all designs. It is evident that RL agents require fewer stimuli to attain the maximum possible code coverage when compared with random stimuli. For the majority of the designs, RL agents based on PPO yield superior results with the penalty scheme. However, the PPO agent with the optimistic scheme reached the same final coverage of RISC-V and FIR design IP with fewer simulations.

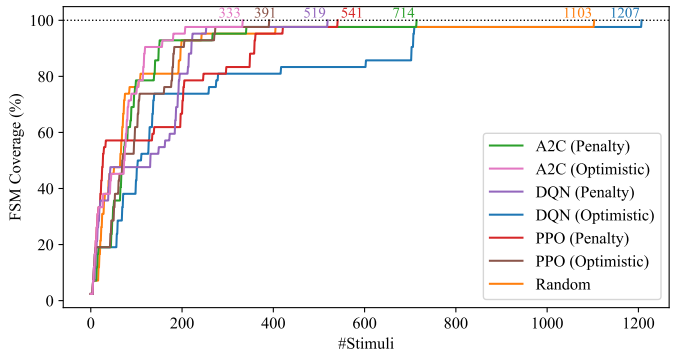

Fig. 4. Progression of the FSM Coverage with respect to stimuli generated by different RL agents and random simulation

An additional experiment was done on JTAG TAP IP to reach maximum FSM coverage to see how RL agents perform if we try to target a specific coverage type. Fig. 4 illustrates the change in FSM coverage for RL stimuli generated by different agents as well as random stimuli. Our results show that RL agents were able to achieve the maximum code coverage with a lower number of stimuli. The A2C agent with the optimistic reward scheme outperformed other agents by reaching 100 % FSM coverage around 3 times faster than random stimulation.

VI. Conclusion

In this work, we introduce a design-agnostic framework leveraging RL for design verification purposes. By automating the setup of RL environments and generating customized testbenches according to the design being verified, our framework streamlines the verification process. Results obtained from verifying six distinct designs confirm that RL-guided verification requires a reduced number of stimuli relative to conventional random simulations to attain the threshold coverage. Moreover, our findings indicate that the PPO-based RL agent often exhibits superior performance compared to DQN and A2C agents.

References

[1] H. Foster, "Wilson Research Group and Siemens EDA, 2022 Functional Verification Study," Siemens EDA, Tech. Rep., 2022.

[2] A. B. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. Springer, 2017.

[3] D. Yu *et al.*, "A Survey of Machine Learning Applications in Functional Verification," DVCon US, 2023.

[4] K. A. Ismail *et al.*, "Survey on Machine Learning Algorithms Enhancing the Functional Verification Process," *Electronics*, 2021.

[5] A. Dinu *et al.*, "Reinforcement Learning Made Affordable for Hardware Verification Engineers," *Micromachines*, 2022.

[6] R. S. Sutton *et al.*, *Reinforcement Learning: An Introduction*. 2018.

[7] J. Schreiner *et al.*, "A new approach for generating view generators," DVCon US, 2017.

[8] *Mako Templates for Python*. [Online]. Available: https : / / www . makotemplates.org.

[9] W. Hughes *et al.*, "Optimizing Design Verification using Machine Learning: Doing better than Random," arXiv, 2019.

[10] B. S. Varambally *et al.*, "Optimising Design Verification Using Machine Learning: An Open Source Solution," arXiv, 2020.

[11] X. Zheng *et al.*, "Using Neural Networks for Novelty-based Test Selection to Accelerate Functional Coverage Closure," 2023.

[12] M. A. E. Ghany *et al.*, "Speed Up Functional Coverage Closure of CORDIC Designs Using Machine Learning Models," *ICM*, 2021.

[13] N. Bhuvaneswary *et al.*, "Hybrid Optimized Verification Methodology using Deep Reinforcement Neural Network," *J. Intell. Fuzzy Syst.*, 2023.

[14] S. Tweehuysen *et al.*, "Stimuli Generation for IC Design Verification using Reinforcement Learning with an Actor-Critic Model," in *IEEE ETS*, 2023.

[15] E. Ohana, "Closing Functional Coverage With Deep Reinforcement Learning: A Compression Encoder Example," DVCon US, 2023.

[16] P. Xu *et al.*, "A Reinforcement Learning Approach to Design Verification Strategies of Engineered Systems," IEEE SMC, 2020.

[17] H. Choi *et al.*, "Application of Deep Reinforcement Learning to Dynamic Verification of DRAM Designs," in *DAC*, 2021.

[18] A. J. Shibu *et al.*, "VeRLPy: Python Library for Verification of Digital Designs with Reinforcement Learning," arXiv, 2021.

[19] N. Pfeifer *et al.*, "A Reinforcement Learning Approach to Directed Test Generation for Shared Memory Verification," in *DATE*, 2020.

[20] R. Liang *et al.*, "Late Breaking Results: Test Selection For RTL Coverage By Unsupervised Learning From Fast Functional Simulation," in *DAC*, 2023.

[21] A. Raffin *et al.*, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, 2021.

[22] C. Bob, *How to connect SystemVerilog with Python*, Mar. 2019.

[23] G. Brockman *et al.*, *OpenAI Gym*, 2016.

[24] A. V. Rashinkar, "A Comparison of Higher-Level Hardware Description Languages for RTL-Design," M.S. thesis, TU Dresden, 2023.

[25] *OpenCores*. [Online]. Available: https://opencores.org.