AI Driven Advanced Debugging in SoC Design Verification

Shreya Jayateerth Joshi, Poonam M Shettar, Sanjoy Saha, Alok Kumar, S Shrinidhi Rao,Garima Srivastava
Samsung Semiconductors India R&D, Bengaluru, India

***Abstract*- The recent advancements in System-on-chips (SoC) have led to the integration of multiple application-specific processors to support the increasing processing demand. Such complex designs make the Design Verification (DV) stage challenging. Since the traditional debug mechanisms fail to keep up, we propose Artificial Intelligence (AI) based verification techniques to assist a verification engineer in testbench (TB) bring-up and debugging. The proposed approach promises to improve verification efficiency by using Deep learning models and Python script alongside SystemVerilog (SV) assertions with Universal Verification Methodology (UVM) reporting mechanisms in the TB. We leverage the capability of Deep learning algorithms to identify required TB or design updates in the post-simulation stage, as well as support testbench bring-up in the pre-simulation stage. We have employed Large Language Models (LLM) to provide interactive query-based DV assistance thereby, making our work more user friendly. With this approach, we observed an average of 56% reduction in the overall verification effort.**

I. INTRODUCTION

The demand for high processing capabilities in a chip continues to challenge the semiconductor industry. As predicted by Moore's Law, the miniaturization of transistors has enabled the production of highly complex designs. However, the manufacturing cost has significantly increased, placing immense pressure on DV. In a complex SoC, the verification effort has increased substantially due to the large number of signals to be analyzed and longer simulation time. Hence, the process of TB bring-up and debugging are a bottleneck in the chip life cycle. The traditional debugging flow is highly reliant on manual log inspection, intense waveform analysis, and repeated simulation runs. Such approaches are time-consuming and they fail to leverage the large amount of reusable data generated across multiple verification projects. The proposed methodology presents a hybrid AI-driven DV assistance, which leverages Deep learning models, SV assertions/checkers, rule-based analysis and LLM-based reasoning to enhance the efficiency and accuracy of SoC DV. We propose a multi-stage framework to assist verification engineer in TB bring-up (pre-simulation stage) and in simulation log and wave dump analysis (post-simulation stage). The key features of our approach are as follows:

1) The TB configurations and initialization sequences are checked using rule-based checks, fuzzy and semantic register mapping, and anomaly detection. It also includes reserved region validation of the memory map and security configuration checks for detection of illegal writes. The early detection of TB issues before simulation helps in reducing TB bring-up time, simulation iterations and debugging effort.

2) The Python rule-based analysis forms the backbone of simulation log analysis in the post-simulation stage. It picks UVM messages displayed by the SV assertions/checkers and guides in extracting signals to be fed to the ML models for further analysis.

3) The comparative study of ML models employed in the post-simulation stage concluded on the best-suited model for bus protocol violation in terms of accuracy and precision. This analysis was performed on Advanced High-performance Bus (AMBA AHB v3.0), Advanced Peripheral Bus (AMBA APB v3.0), Advanced eXtensible Interface (AMBA AXI v3.0) [11] interface signal dumps.

4) The anomalies flagged by ML model are picked up by the Python rule-based analysis to check possible issues and suggest TB/design updates, thereby reducing the dependency on the verification engineer.

5) The findings and conclusions from ML models and rule-based checks in pre- and post-simulation stages are recorded in debug logs, which are referred by the interactive debugger.

6) The pre-trained Bidirectional Encoder Representations from Transformers (BERT) [13] and OpenAI [30] models, past project databases, simulation log and debug logs are used to match known error patterns, suggest fixes, and provide interactive query-based debugging assistance to a verification engineer.

II. LITERATURE SURVEY

Andy Truong Et al., [1] present a comparative study of clustering and classification algorithms applied to simulation log files in Constrained Random Verification (CRV) TB. The works in [2]-[4] portray ML models employed in stimulus generation to enhance Design Verification. ML models can also help in improving coverage metrics in CRV TB [5]-[8]. E. El Mandouh Et al., [9] utilizes K-means clustering to identify trace segments in the trace dump, which are either unique for closer debugging or redundant. Such approaches only assist and require

verification engineers' intervention since they may or may not lead to design bugs. Abhiram S. Et al., [10] present a comparative study of ML models in the verification of APB protocol. In this work, the ML models are trained to only identify the type of transactions (Read/Write) and highlight error responses.

In design verification scenarios where it is difficult to randomize data or events, generating randomized test cases and running simulations can be computationally intensive. The randomized approach can result in longer simulation time in case of highly complex designs. We would always need directed tests to ensure thorough verification. Hence, a TB usually includes both directed tests and randomization. Our approach is applicable to hybrid and multifaceted TB in which a TB could be based on SV, SV+C etc. and tests can be both directed and randomized. Also, a mere simulation log analysis can result in inconclusive debugging since simulation log provides partial insights of design and testing environment. Hence, we have included simulation wave dump analysis after fetching relevant information from simulation log analysis to present a complete debug conclusion. The flow of our framework is in line with the human approach of verification. It aims at assisting a verification engineer at every stage of verification in a user-friendly manner to quicken DV.

III. METHODOLOGY

A. Pre-Simulation Framework: Testbench bring-up assist

The framework as seen in Fig. 1. is executed after initial TB bring-up where TB is in its nascent stage. Hence, an early detection of TB issues before simulation reduces the number of simulation iterations. This framework consists of ML models and Python routines to validate TB register initialization sequences of the project under development. The TB data is fetched and reformatted (CSV format) by python routine, which is used in both pre- and post-simulation stages. By leveraging deterministic rule-based checks, fuzzy, semantic similarity analysis, and ML models, the framework identifies missing, inconsistent, or incorrect register settings before simulation. This framework consists of below stages:

1) *Pre-processing and Rule based checking*

As a part of pre-processing, a statistical baseline model B is constructed from a set of previously verified projects $\{P_1,P_{2..}P_n\}$, where each project dataset is represented as $P_i=\{(r_j,v_j)\}$. Here, r_j is the register key and v_j is the corresponding initialization value. For every register key, the baseline (JSON format) maintains $B(r)=\{p_r,v_{r,mode},V_{r,set}\}$ where p_r is presence ratio denoting frequency of occurrence across projects, $v_{r,mode}$ is Mode Value denoting most frequently used initialization value and $V_{r,set}$ is Value Set denoting set of all observed values.

After pre-processing, deterministic rule-based checks are performed to capture missing/additional register configurations and register value deviations between baseline JSON and TB CSV file into an output CSV file.

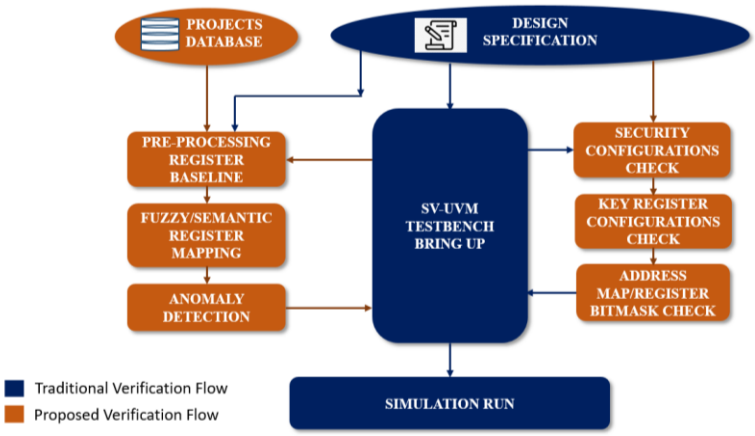

Fig. 1. Traditional vs Proposed Verification flow in the Pre-simulation stage

2) *Fuzzy and Semantic Register Mapping*

The register naming conventions vary across projects due to typographical differences or IP revisions. To ensure reliable cross-project mapping, the below mechanisms are performed on the output CSV file from previous step:

- *Fuzzy String Matching:* By leveraging RapidFuzz [26] similarity metric, register names are compared and register pairs exceeding a similarity ratio of ≥ 85% are flagged as probable matches. These matches highlight probable syntactic register name variations (e.g., EN_CTRL vs ENABLE_CTRL) thereby filtering preliminary string deviations. Sematic matching is done next for advanced filtering of string deviations.

- *Semantic Embedding Matching:* A lightweight pre-trained transformer encoder [13] (BERT) is employed to generate vector embeddings for register identifiers. Cosine similarity [34] between these embeddings is used to identify semantically related terms. The register identifier matches with similarity ≥ 0.75 are treated as potential equivalents.

3) *Machine learning based anomaly detection*

To identify previously unseen TB configuration inconsistencies, an Isolation Forest (IF) based anomaly detection [12] ML model is employed. Each project configuration is vectorized into a fixed-length feature vector $X_i = [v_{i1}, v_{i2},.. v_{in}]$, where each dimension corresponds to a baseline register. The ML model is trained on baseline datasets to model the normal distribution of register patterns.

4) *Reserved Region and Security Configuration Validation:*

This framework stage verifies illegal writes to reserved bits of Special Function Registers (SFRs). The register specification defines the access permissions for each bit. For every 32-bit SFR, a bit-mask is generated and stored in a CSV file along with its corresponding register name or address and offset. The register values extracted from TB are then compared and checked to ensure that reserved regions are not written. The security configurations of the registers are validated. Any mismatch is flagged as a potential source of bus errors or privilege faults. All the warnings and errors flagged by this framework are recorded in the debug log for future reference as required.

B. Post-Simulation log and wave dump analysis

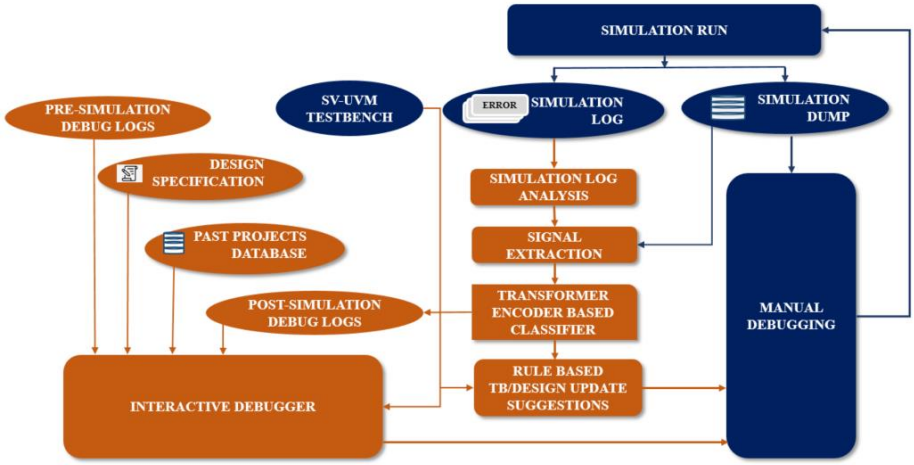

Fig. 2. Traditional vs Proposed Verification flow in the Post-simulation stage

1) *SV Assertions/Checkers with UVM reporting*

Specialized checkers/assertions can be written to validate the design signals' behaviour and flag mismatches in the form of UVM messages (UVM_INFO, UVM_ERROR, and UVM_FATAL) in the simulation log. These messages along with the other UVM messages from the TB guide the Python script in fine-tuning signal extraction.

2) *Deep Learning Models*

Instead of relying on basic VIP (Verification IP) monitors or manual waveform inspection, our proposed framework employs deep learning models to analyze on-chip bus signals' simulation wave dumps (AHB, APB, and AXI) and dig deep into the abnormalities in transaction flow. This approach accelerates root-cause detection issues like transaction stall, protocol violations, thereby reducing debugging time.

a) *Dataset Generation and pre-processing for model training:* A rule-driven dataset generator produces both compliant and non-compliant (AHB, APB, and AXI) transactions. The faulty behavior is introduced by systematically injecting specific protocol violations into the dataset. Each transaction segment, along with its corresponding error annotation is saved in an individual JSON file for model training.

- *APB Dataset*: Each transaction consists of a setup phase followed by one or more access phase. Faults are introduced by tampering control signals, like asserting PENABLE early, omitting PREADY, or de-asserting PSEL during an active access phase.
- *AHB Dataset*: It uses rule-based checks to identify illegal HTRANS state transitions, address misalignment, invalid HPROT settings, and HRESP. Temporal features like address deltas between successive transactions are captured to facilitate sequence-based learning.

- *AXI Dataset*: Independent datasets are generated for read/write transactions. Faults are embedded by corrupting handshake sequences, control responses between related channels.

b) *Comparative study of ML models*

We considered the following ML models in this comparative study:

- *Recurrent Neural Network (RNN):* Recurrent layer with 64 hidden units is used to process sequential input, producing a 64-dimensional representation at each step to retain and learn patterns within the sequence [25].
- *Long Short-Term Memory (LSTM*): We used two stacked LSTM layers, designed to learn both short and long-term dependencies in the transaction sequences [24].
- *Gated Recurrent Unit (GRU):* Two recurrent layers with 64 hidden units efficiently capture dynamic signal behavior and mid-range dependencies without excessive computational cost [31].
- *Temporal Convolutional Network (TCN):* Three 1D convolution blocks (64 filters, 3 kernels , dilations = 1, 2, 4) capture short and long-term dependencies to detect timing-based violations and irregular protocol sequences [23].
- *Hybrid CNN-GRU Model*: Two 1-D convolutional layers (64 filters, 3 kernels) extract local features, and then two GRU layers (64 hidden units each) learn global sequence relationships [22].
- *Transformer Encoder*: It uses self-attention to capture global dependencies in signal sequences; composed of an input-projection layer with positional encoding followed by 2 encoder blocks, each containing 4-head multi-head attention, a 128-dimensional feed-forward sublayer, and layer-normalization [21].

All models were trained on fixed-length transaction windows, where each time step represents a set of AMBA protocol signal values across one clock cycle. The final hidden representations from each network were passed through a dense layer with Rectified Linear Unit (ReLU) activation [27], followed by a softmax [28] layer for multi-class classification of transaction errors. A categorical cross-entropy loss function [20] and Adam optimizer [19] were used for all experiments, with training conducted using a batch size of 32 with dropout rate of 20%. The consistent setup across architectures ensures fair performance comparison while highlighting the trade-offs between recurrent, convolutional, and attention-based approaches for protocol error classification. All the predictions from the ML models are recorded in the debug log for future reference as required.

3) *Rule- based Analysis: Python scripting*

a) In the post-simulation stage, UVM messages from the simulation log are analyzed to facilitate AXI, AHB and APB bus signals extraction from simulation wave dump. The design signals required to debug are extracted in CSV format and fed to the ML model. The predictions from the ML model on the bus transaction could further lead to the extraction of security configuration signals or other processor signals from the simulation wave dump. Thereby, the python routine performs selective data extraction from wave dump to minimize unnecessary information processing and captures data critical to current debugging flow.

b) Using the ML models' predictions, the Python script performs root cause analysis to check if the simulation failure is due to the TB update or Design update. In case of some protocol violations, the script maps the failing transaction to the line of code in the TB to be updated, using the TB (CSV file). It also suggests if the failure is due to any missing security configurations in the transaction path or any missing port definitions in the design. All the conclusions from the rule-based analysis are recorded in the debug log for future reference as required.

4) *LLM-Based Log Analysis and Interactive Debugger*

This framework consists of a generative AI layer to enable intelligent, context-aware debugging assistance. While regex-based parsing [29] is employed to systematically extract UVM messages from simulation logs, traditional approaches end at this extraction stage. The proposed system extends by leveraging LLMs and embedding-based semantic retrieval to interpret extracted log messages, classify them, and correlate with historical debug data to recommend possible corrective actions. This framework integrates seamlessly with the verification environment to process raw simulation logs, learn from recurring error patterns, and provide interactive, human-like assistance to verification engineers. The debug logs from pre- and post-simulation stages are referred by the interactive debugger as required.

a) *Log Parsing and Pre-processing*: The UVM messages from checkers are extracted and analyzed by regex-based script that scans for keywords such as *ERROR*, *FATAL*, *ASSERT*, and *FAIL*. Each detected log entry is timestamped, categorized based on severity and stored in CSV file. That file is fed to subsequent embedding-based

semantic comparison and LLM-driven reasoning, enabling the system to interpret log patterns with historical failure data.

b) *Embedding-Based Error Similarity*: After UVM messages are extracted from simulation logs, each unique message is converted into a semantic embedding vector using a pre-trained BERT-based encoder. Historical errors, along with their root causes, fixes, and notes, are stored in JSON database and indexed using Facebook AI Similarity Search (FAISS) [18] for real-time similarity search. When a new error is encountered, its embedding is compared against the FAISS index using cosine similarity. If a match above a defined threshold is found, the corresponding record containing the fix is retrieved; otherwise, the system prompts the user to add the new entry, automatically updating the database and index. This mechanism provides a continuously evolving debug knowledge base across projects.

c) *Interactive Q&A:* An OpenAI-based language model [30] provides contextual reasoning and question answering by referencing the debug log. The verification engineer can enquire about design signal's value, test progress, suggestions on fixing simulation failure and explanation on pre- and post-simulation suggestions.

5) *Toolchains and Libraries Used*

The framework uses libraries for data processing, model training, and interactive debugging. Pandas [17] is used in both pre- and post simulation framework for retrieving design specifications, pre-processing TB etc. The regular expressions (re) library facilitates efficient extraction of error and event messages from simulation logs. Scikit-learn [16] supports dataset preparation, pre-processing, and Isolation Forest ML model. PyTorch [15] is used in LLM inference and training of deep learning models (RNN, LSTM, GRU, TCN, CNN-GRU, and Transformer Encoder). FAISS library is used in LLM-Based Log Analysis. Matplotlib [14] and Seaborn library [32] are employed to visualize anomaly trends and model performance comparisons. Tkinter library [33] is used in building GUI for the proposed framework.

6) *Integration Challenges*

We overcame the below challenges faced during the implementation of the proposed framework:

a) Dataset creation for AXI/AHB/APB protocol with valid and invalid transactions for model training.

b) Consistent conversion and integration of different formats (text, JSON, CSV).

c) Synchronization between the trained ML models' input and the simulation signal dumps.

d) Mapping the anomalies predicted by ML-models and rule-based violations to the exact transaction and register configuration in the TB.

e) Maintaining consistent decision flow in a hybrid framework containing the deterministic scripts and probabilistic AI components (ML and LLM modules).

IV. RESULTS

The pre-simulation framework results are shown in figs. 3 and 4. Fig. 3 shows a bit-level deviation heat map that compares the new project's register-field patterns with the baseline configuration from prior verified projects. Each row corresponds to a register-field pair, and each column represents a bit position within its initialization value. Red cells indicate bits that differ from the baseline configuration and blue cells represent matching bits. Fig. 4 presents a 2D projection of register initialization patterns obtained using PCA on Isolation Forest derived embeddings. Each point represents a verified project's configuration and star indicates the new project.

The evaluation in post simulation framework is based on standard classification metrics like accuracy, precision, and F1-score. The variation of loss throughout the training process can be seen in figs. 5, 6, 7 and 8. A comparative summary of the model performances is provided in Table I .The Transformer encoder achieved the highest overall performance, demonstrating its ability to capture complex temporal and contextual dependencies. The LSTM and GRU models also performed well in learning long-range dependencies. RNN showed lower accuracy due to its limited ability to retain long-term information. TCN model achieved stable performance with faster convergence compared to recurrent models. The hybrid CNN-GRU model demonstrated a balance between spatial feature learning and temporal dependency modeling. Figs. 9, 10 and 11 shows the GUI snapshots of pre-simulation dashboard with debug log, post-simulation dashboard with debug log and LLM-based interactive debugger.

Considering one verification engineer and, the verification effort was evaluated in TB bring-up, test generation, simulation runtime and debug effort. The proposed framework building requires a one-time effort of 12 weeks and 2 weeks' time to scale it to a new project. Fig. 12 shows the verification effort analysis for one feature's DV with 15 weeks verification effort in traditional verification approach and 5-6 weeks in proposed verification approach. When deployed in other projects, we observed an average reduction of 56% in verification effort.

V. CONCLUSION AND FUTURE SCOPE

The intention of this work was to reduce the verification effort and support the verification engineer in both pre- and post simulation stages. We achieved this by seeing an average of 56% reduction in the overall verification effort on projects for which we deployed this methodology. We also achieved a reduction in TB bring-up effort using automated configuration checks and debugging effort using automated simulation log and wave dump analysis. The entire framework was made user-friendly by incorporating GUI and interactive debugger.

When this is deployed in future projects, we expect further reduction in debugging time due to improved debug knowledge base of interactive debugger. We expect reduction in TB bring-up time due to improved baseline data from past projects in the pre-simulation stage. We are working on improving design specification analysis to highlight feature changes across projects for better support in the pre-simulation stage. We aim to run the post-simulation framework during simulation to reduce verification time. We are working to improvise the suggestions by interactive debugger using web-based links, simulation snapshots etc. to make the framework more beneficial.

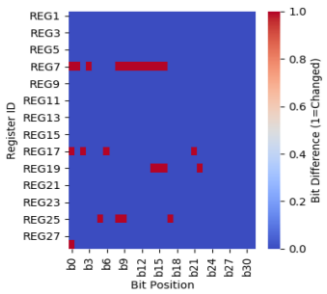

Fig. 3. Register Bit deviations

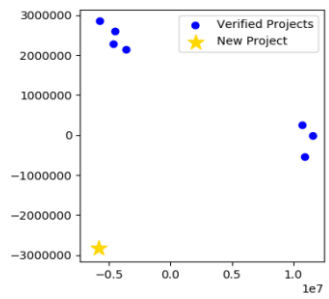

Fig. 4. Anamoly Map (Isolation Forest + PCA)

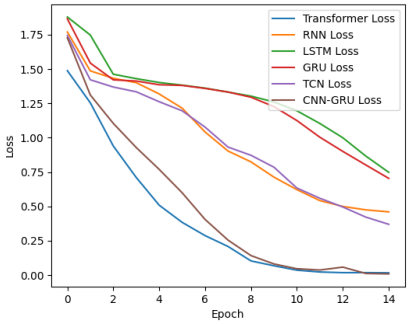

Fig. 5. Loss across epochs APB

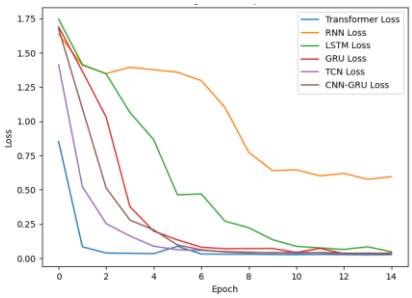

Fig. 6. Loss across epochs AHB

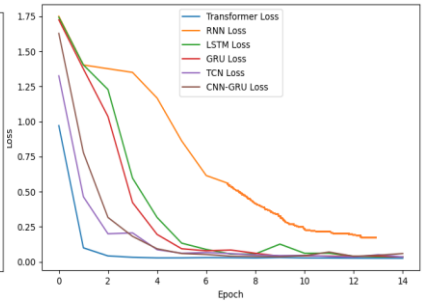

Fig. 7. Loss across epochs AXI-write channel

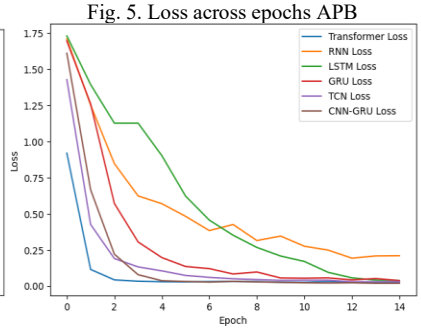

Fig. 8. Loss across epochs AXI-read channel

TABLE I
DEEP LEARNING MODELS PERFORMANCE METRICS

Protocol	Algorithm	RNN	LSTM	GRU	TCN	CNN-GRU	Transformer-Encoder
APB	**Accuracy (%)**	95.50	97.65	96.38	92.69	95.68	98.50
	Precision (%)	95.82	97.76	96.23	84.08	97.34	98.67
	F1-Score	93.74	97.77	96.05	84.85	92.60	98.48
AHB	**Accuracy (%)**	95.17	96.21	96.30	96.67	97.45	97.89
	Precision (%)	91.27	91.29	91.45	92.45	92.62	92.71
	F1-Score	93.18	93.20	93.24	94.56	94.73	94.84
AXI-Read Channel	**Accuracy (%)**	92.00	97.10	97.50	97.98	98.15	98.32
	Precision (%)	91.13	97.23	97.65	97.99	98.32	98.84
	F1-Score	91.95	96.54	96.63	98.01	98.43	98.76
AXI-Write Channel	**Accuracy (%)**	91.32	96.17	96.16	98.05	99.17	99.50
	Precision (%)	91.11	97.05	97.12	97.54	99.15	99.56
	F1-Score	91.56	95.89	95.91	98.16	99.16	99.51

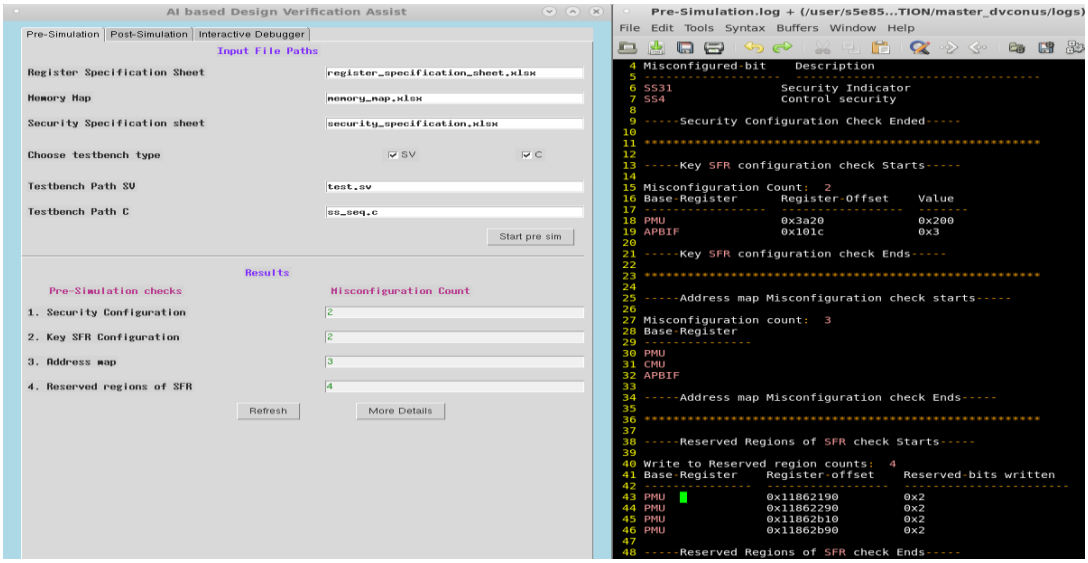

Fig. 9. Pre Simulation Debugger GUI

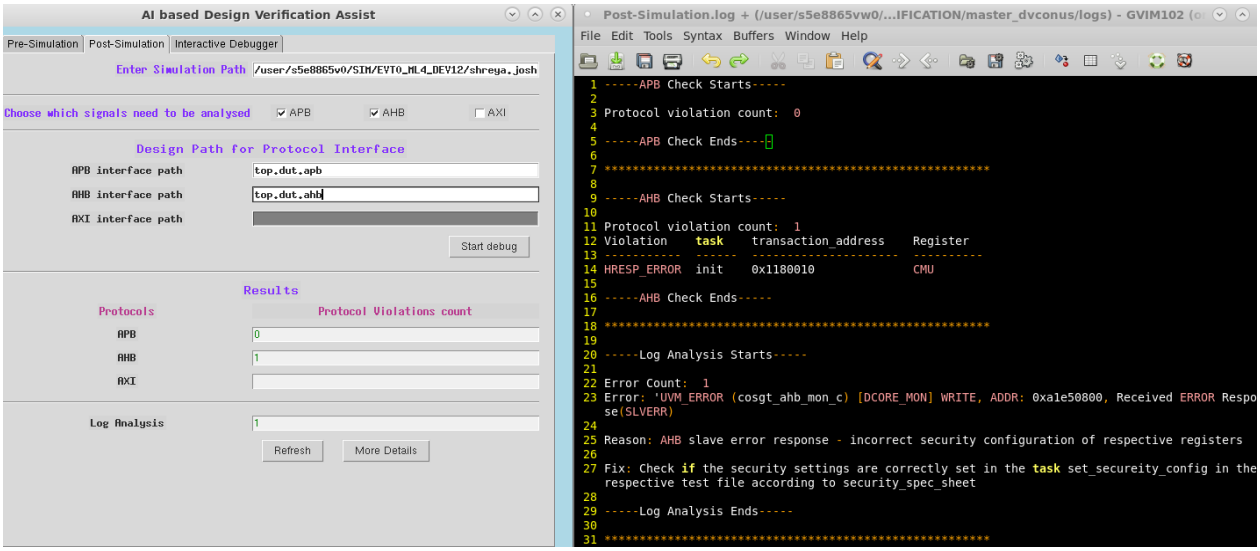

Fig. 10. Post Simulation Debugger GUI

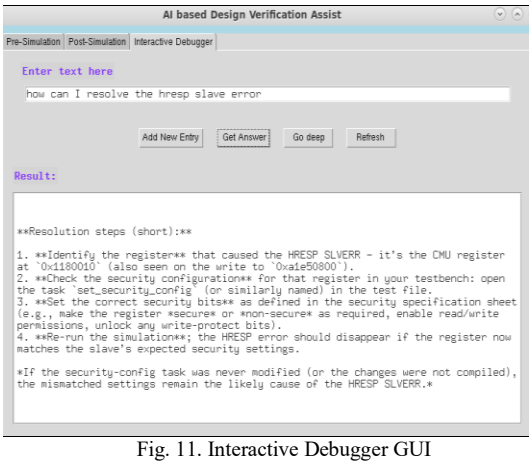

Fig. 11. Interactive Debugger GUI

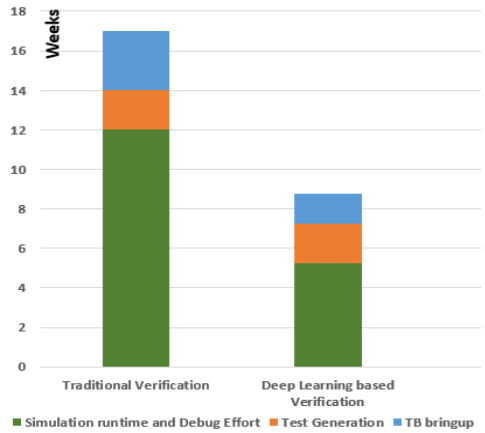

Fig. 12. Verification Effort Analysis

REFERENCES

[1] Andy Truong, Daniel Hellström, Harry Duque, Lars Viklund, “Clustering and Classification of UVM Test Failures Using Machine Learning Techniques”, DV Con Europe, 2018.

[2] Khaled A. Ismail, Mohamed A. Abd El Ghany, “Survey on Machine Learning Algorithms Enhancing the Functional Verification Process” Electronics 2021, 10(21), 2688; https://doi.org/10.3390/electronics10212688

[3] K. A. Ismail and M. A. Abd El Ghany, "High Performance Machine Learning Models for Functional Verification of Hardware Designs," 2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES), Giza, Egypt, 2021, pp. 15-18, doi: 10.1109/NILES53778.2021.9600502.

[4] R. K. M. Vangara, B. Kakani and S. Vuddanti, "An Analytical Study on Machine Learning Approaches for Simulation-Based Verification," 2021 IEEE International Conference on Intelligent Systems, Smart and Green Technologies (ICISSGT), Visakhapatnam, India, 2021, pp. 197-201, doi: 10.1109/ICISSGT52025.2021.00049.

[5] S. Jayakumar, P. V. Pillai and S. K. Mandal, "Accelerated Design Verification Coverage Closure Using Machine Learning," 2025 38th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID), Bangalore, India, 2025, pp. 332-337, doi: 10.1109/VLSID64188.2025.00070.

[6] B. Kumar, G. Parthasarathy, S. Nanda and S. Rajakumar, "Optimizing Constrained Random Verification with ML and Bayesian Estimation," 2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD), Snowbird, UT, USA, 2023, pp. 1-6, doi:10.1109/MLCAD58807.2023.10299818.

[7] R. K. M. Vangara, B. Kakani and S. Vuddanti, "An Analytical Study on Machine Learning Approaches for Simulation-Based Verification" 2021 IEEE International Conference on Intelligent Systems, Smart and Green Technologies (ICISSGT), Visakhapatnam, India, 2021, pp. 197-201, doi: 10.1109/ICISSGT52025.2021.00049.

[8] P. Gaur, S. S. Rout and S. Deb, "Efficient Hardware Verification Using Machine Learning Approach," 2019 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS), Rourkela, India, 2019, pp. 168-171, doi: 10.1109/iSES47678.2019.00045.

[9] E. El Mandouh and A. G. Wassal, "Accelerating the debugging of FV traces using K-means clustering techniques," 2016 11th International Design & Test Symposium (IDT), Hammamet, Tunisia, 2016, pp. 278-283, doi: 10.1109/IDT.2016.7843055.

[10] Abhiram Srisai & Mohammed Wasim,” Investigating Machine Learning for verification of AMBA APB protocol”, 2022, [Online] https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9078450&fileOId=9078709.

[11] AMBA architecture [website:https://developer.arm.com/documentation/ihi0022/latest]

[12] Liu, Fei Tony & Ting, Kai & Zhou, Zhi-Hua. (2009). Isolation Forest. 413 - 422. 10.1109/ICDM.2008.17.

[13] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers). 2019.

[14] Barrett, Paul & Hunter, J. & Miller, J.T. & Hsu, J.-C & Greenfield, P.. (2005). matplotlib -- A Portable Python Plotting Package.

[15] Paszke, Adam et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” ArXiv abs/1912.01703 (2019)

[16] Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 12.(2012)

[17] Mckinney, Wes. (2011). pandas: a Foundational Python Library for Data Analysis and Statistics. Python High Performance Science Computer.

[18] Douze, Matthijs et al. “The Faiss library.” ArXiv abs/2401.08281 (2024)

[19] Kingma, Diederik P. and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” CoRR abs/1412.6980 (2014)

[20] Mao, Anqi et al. “Cross-Entropy Loss Functions: Theoretical Analysis and Applications.” ArXiv abs/2304.07288 (2023)

[21] Vaswani, Ashish et al. “Attention is All you Need.” Neural Information Processing Systems (2017).

[22] R. Jaiswal and B. Singh, "A Hybrid Convolutional Recurrent (CNN-GRU) Model for Stock Price Prediction," 2022

[23] S. Chauhan, Sehaj, S. Kumar, Siddharth, D. Panwar and S. Chopra, "Temporal Convolutional Network and its Application in Various Sectors," 2023

[24] Staudemeyer, Ralf C. and Eric Rothstein Morris. “Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks.” ArXiv abs/1909.09586 (2019)

[25] Schmidt, Robin M.. “Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.” ArXiv abs/1912.05911 (2019)

[26] RapidFuzz Python Library (website: https://share.google/bi5714MAdkHI8fmRF)

[27] ReLU Activation Function (website:https://share.google/8mAzpedgHrMfm8EPh)

[28] Softmax activation function (website: https://share.google/0yjyaa2nqSO58Rmz4)

[29] Python RegEx (website: https://share.google/lJ8t1jOQXH3gLMapu)

[30] openai-gpt-oss-120b (website:https://platform.openai.com/docs/models/gpt-oss-120b)

[31] GRU Networks (website: https://www.geeksforgeeks.org/machine-learning/gated-recurrent-unit-networks/)

[32] Seaborn library (website: https://seaborn.pydata.org/)

[33] Tkinter Library (website: https://www.geeksforgeeks.org/python/python-gui-tkinter/)

[34] Cosine Similarity (website: https://www.geeksforgeeks.org/dbms/cosine-similarity/)