Towards Self-Adaptive SoC Design Verification: KG-Enhanced Generative AI, RL and Backpropagation Debugging

Insu Jang [1] Seonghee Yim [2] Hanna Jang [3] Youngsik Kim[4]

[1]Samsung Electronics, Hwaseong, Korea.
[1] is1226.jang; [2] sh.yim; [3] hn01.jang; [4] ys31.kim@samsung.com

***Abstract*- This paper proposes a verification-efficiency–maximizing plan that leverages knowledge-graph (KG)-based reinforcement learning for the critical silicon bugs of SOCs during the planning and front-end design stages. Design changes of blocks, IPs, or modules are modeled as triples in a KG, and a reinforcement-learning agent learns to prioritize verification tasks by exploiting the functional relationships captured in the KG. The KG-based approach demonstrates superior performance compared with conventional verification that relies solely on the hierarchical design-instance relationships.**

I. INTRODUCTION

Modern System-on-Chip (SoC) designs integrate hundreds of functional blocks (BLKs) and intellectual property (IP) cores, creating verification challenges that scale exponentially with complexity. While simulation-based verification remains essential, its effectiveness diminishes when debugging intricate, multi-layered failures across RTL, gate-level, and system-level abstractions. Traditional methods often rely on manual trace analysis—a process both time-consuming and prone to human error—especially when diagnosing issues in power management, clock domain crossings, or protocol compliance.

To overcome these limitations, we propose an integrated debugging framework that combines **knowledge graph(KG)**-based reasoning and **backpropagation**-based failure analysis, and provide examples from the actual development flow of SOC. We will try to prove the results in detail. The task information used for this purpose may be a general case or a case of specific conditions, but it consists of cases that may occur in actual development.

Our approach uniquely bridges:

- Structural knowledge representation (via vectorized DB of SoC block-IP graphs)
- Dynamic fault propagation modeling (via backpropagation from test failures to root causes)

This synergy enables:

✓ Automated fault localization by tracing verification failures backward through design hierarchies and dependencies.

✓ Cross-layer correlation between simulation results, specification changes (Design H/W Description Spec, Verification Spec), and historical debug patterns.

✓ Proactive error prevention by identifying high-risk IP interactions during early verification.

II. HELPFUL HINTS

The existing verification methodology relies heavily on constraint-random testing and coverage-driven approaches. Machine-learning techniques—particularly reinforcement learning (RL)—have demonstrated potential, but they suffer from sparse reward signals. Knowledge graphs (KGs) have been employed in EDA for design documentation, yet they are not widely used for verification. Backpropagation debugging has been applied to software debugging, but it remains a novel concept in the hardware-verification context. This paper presents concrete data and system configurations for three key utilization areas, illustrated as follows.

- **Knowledge Graph Construction :**
  . From design and verification documents, we define entities such as BLKs, IPs, functions, and test cases that correspond to the hierarchical components of SoC. These entities are then mapped to capture their relationships.
  . In the initial KG build-up, RL-generated results will be leveraged to assign priorities to important paths and high-risk interfaces.

. Version information of blocks that change across design stages is extracted. The version data is linked to the previously defined entities to support release-level tracking.

- **Reinforcement Learning Optimization**

. Before expanding to the full SoC, verification begins with a focused IP block. IPs are classified into two types for targeted analysis.

IP_Type1: Strong internal relationship with its block, weak relationship with external blocks(e.g., Audio block/abox IP)

IP_Type2: Belongs to a specific block but exhibits both internal and external relationships(e.g., USB block/ bus IP)

. The framework balances exploratory testing (new test cases) against exploitive testing (known critical paths).

. Use transfer learning for similar IP blocks to accelerate training

- **Verification Efficiency Boosters**

. In addition to an ETL pipeline, a workflow that updates the graph database(Graph DB) and its edges will be built—either as a systematic pipeline or as an autonomous agent—to reflect design-change data.

. In order to effectively implement incremental verification for repetitive improvement, a backpropagation debugging flow is created to trace test-result dependencies. This flow is learned with RL, enabling the system to update relationship information dynamically.

. A concrete scoring-based reward function is proposed to concentrate RL rewards on historically problematic regions.

III. Proposed Framework & Experimental Results

The proposed framework integrates Generative AI for test case synthesis, Knowledge Graphs for semantic verification, and Backpropagation Debugging for gradient-based error tracing. This closed-loop system adapts verification strategies to design changes through reinforcement learning policies informed by both KG reasoning and debugging signals.

A. Backpropagation-Enhanced KG Debugging

To capture the graph structure of nodes that modify information and their relationships in a real-world project, we define the IP type based on the version-wise change magnitude of IPs associated with each block. For each IP type, we track the variability of test results for the functions that are related to the changed IP information. The resulting impact data are used to update the dependency information between blocks during functional verification. These updated relationship data are then feedback as the verification base KG graph. We introduce a bidirectional analysis flow:

1) **Forward Verification:**
   Execute traditional test benches to detect failures or apply the Graph DB relationship of the currently applied edge's attribute-based information.

2) Backward Propagation:
   Step 1: Anchor failures to specific IPs/blocks using KG embeddings (e.g., JTAG access violations in OIS_MCU_TOP during sleep states).
   Step 2: Propagate errors backward through dependency edges (e.g., clock/reset networks, data paths) using gradient-like sensitivity analysis.
   Step 3: Rank root causes by combining KG connectivity weights and historical debug statistics.

B. Reinforcement Learning Implementation Guidance

- State representation: Encode both structural (KG topology) and temporal (simulation cycle) features, Use graph neural networks for embedding complex relationships
- Reward shaping: Balance between bug detection (immediate reward) and coverage (long-term reward), Incorporate domain knowledge via penalty terms for known failure modes
- Policy optimization: Start with proximal policy optimization (PPO) for stable training, Consider hierarchical RL for multi-level verification tasks
- Reward Function Breakdown
  Reward can be calculated from the test results of testcases extracted through RL and KG, using the results of total testcases. The followings are examples of rewards for testcase types and examples of calculating rewards. 'Predicted' in 'Testcase type' column means that our RL network with KG predicted these testcases

should be proceeded with test first. 'Not Predicted' means that these testcases are not predicted by our RL network. 'Yes' in 'The result is changed' column means that the results of these testcases are changed. 'No' means that the results are not changed.

Testcase type	The Result is changed	Reward for each testcase
Predicted	Yes	+10
	No	-10
Not Predicted	Yes	-5

Test Result	Reward
The results of 15 testcases are changed from total 100 testcases. The model predicted 20 testcases and the results are changed in 12 of the predicted testcases. 3 testcases with changed results failed to predict.	12*10 + 8*(-10) + 3*(-5) = 25
The results of 16 testcases are changed from total 100 testcases. The model predicted 15 testcases and the results are changed in 13 of the predicted testcases. 3 testcases with changed results failed to predict.	13*10 + 2*(-10) + 3*(-5) = 95

C. Debugging and Validation Strategies

1) Develop interactive KG explorers with failure propagation paths
2) Enhance verification process with RL for blocks and IPs of SoC changes or verification code changes
3) Specification-Aware Filtering: Constrain backpropagation using SOR-defined boundaries (e.g., excluding physically impossible paths which we call ‘false path’).

IV. Knowledge Graph Schema and Update Mechanism

The KG schema serves as the semantic backbone of the proposed verification framework, encoding both design intent and failure dynamics.

A. Knowledge Graph Schema Design

The Knowledge Graph (KG) represents both static design structures and dynamic verification semantics of the SoC system. Each node and relationship in the KG corresponds to a verifiable entity or interaction extracted from design specifications, simulation data, and debugging logs.

1) Schema Overview

The KG schema follows a hybrid ontology that captures hierarchical, temporal, and behavioral dependencies, The KG consists of the following entity types:

Block: High-level functional or common subsystem within SoC architecture. Groups multiple IPs and serves as primary structural unit for design partitioning**.**
IP (Intellectual Property): Design module within a block. Fundamental unit of design change, integration, and verification impact analysis.
Feature: Functional capability or design intent implemented by one or more IPs. Semantic abstraction layer bridging low-level design components and high-level verification objectives.
Testcase: Executable verification artifact designed to validate specific features, IP behaviors, or system scenarios.
Class: Logical grouping of testcases based on functionality, scenario type, or verification strategy. Enables hierarchical organization and scalable test management.
DE (Design Element): Fine-grained design constructs such as registers, interfaces, configuration parameters, or protocol elements. Enables traceability between high-level design changes and low-level verification effects.
Relationship Types

The following relationship information can be reflected in Graph DB and the relationships among entities are defined as follows:
CONTAINS: Connects hierarchical structures, such as a Block containing multiple IPs or a Class containing multiple Testcases.
IMPLEMENTED_BY: Links a Feature to the IPs responsible for its implementation, enabling feature-centric impact analysis.

LEARNED_CORRELATION: Represents dynamically learned relationships between entities, derived through reinforcement learning. These edges encode execution-level influence and indirect behavioral dependencies that are not explicitly defined in design or test plans.
VERIFIED_BY: Connects Features or IPs to Testcases that validate their behavior, providing a direct mapping between design intent and verification artifacts.
USES_DE: Links IPs or Testcases to the Design Elements they access or depend on, enabling fine-grained tracing of configuration and interface-level interactions.

Examples for Relationship are shown in the table below.

Relationship	**Example**
CONTAINS	POWER CONTAINS DPU_VX.XX
IMPLEMENTED_BY	test-cpustub-pmu_lpg_test_c-BLOCK=AUD-LOCAL_PMU_MASK-PMU_SVA-COUNT=0-AUTOCLKGATE_DISABLE IMPLEMENTED_BY pmu_lpg_test_c
LEARNED_CORRELATION	ICPU LEARNED_CORRELATION test-cpustub-pmu_lpg_test_c-BLOCK=ICPU-PMU_SVA-COUNT=0 POWER LEARNED_CORRELATION RGBP CSTAT LEARNED_CORRELATION ICPU
VERIFIED_BY	DPU VERIFIED_BY test-cpustub-pmu_lpg_test_c-BLOCK=DPU-PMU_SVA-COUNT=0
USES_DE	test-cpustub-pmu_lpg_test_c-BLOCK=ICPU-PMU_SVA-COUNT=0 USES_DE cpustub

2) Embedding and Representation for KG+RL

The proposed KG is implemented as a property graph using Neo4j to enable practical inspection, querying, and visualization of design-verification relationships:

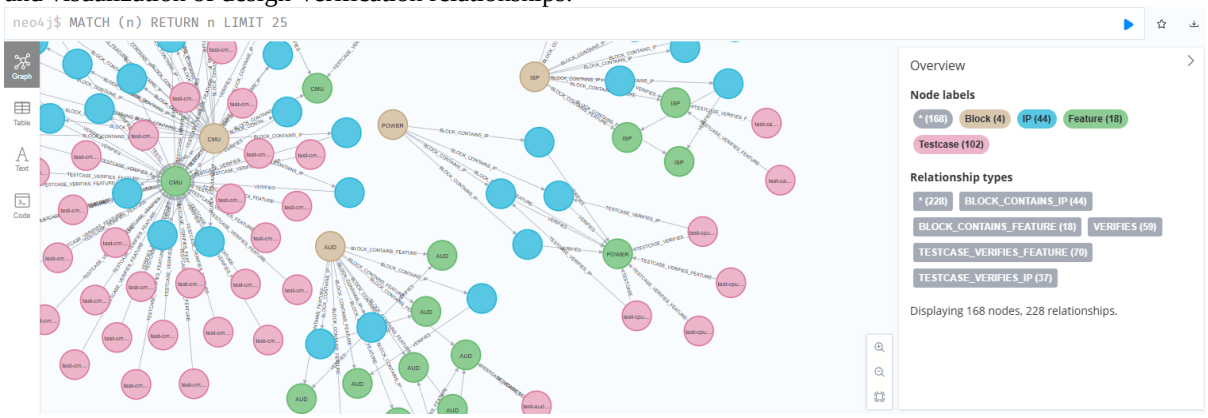

Figure 1 SoC Neo4j-based Graph Database Representation

The figure illustrates hierarchical design structure and verification mappings, together with **PPO**-reinforced **LEARNED_CORRELATION** edges. While static relationships reflect design and test plan information, the learned edges capture dynamic behavioral dependencies. These weights represent the strength of execution-level influence inferred from test result changes, allowing the graph to evolve beyond static connectivity.

B. PPO-based KG-Backpropagation Operation Sequence & Framework Architecture

Step 1: Initialization

1) KG Initializer parses design data to set Block-IP-Feature-Error initial weights and relations
2) Initialize PPO RL Agent's Actor-Critic networks

Step 2: Historical Episode Processing (Sequential through versions)

3) Extract KG embedding(State) from state at that historical point
4) PPO Agent selects action based on state(Testcase selection)

5) Result analysis: Compare test outcome changes between selected and non-selected testcases
6) Reward calculation: Assign rewards based on testcase result change detection
7) PPO Critic estimates state value
8) Advantage computation (GAE)
9) Policy loss calculation using PPO Clipped Objective
10) Value function loss computation
11) Actor-Critic network updates
12) Block-IP-Testcase weight updates (KG backpropagation)

Step 3: Current Version Testcase Selection

13) Extract KG embedding for current version (State)
14) PPO Agent selects action based on state (Testcase selection)
15) Execute Testcase in simulator
16) Result analysis: Confirm testcase outcome changes, calculate rewards

Step 4: PPO Learning Update and Knowledge Graph Backpropagation

17) Compute Advantage using collected data (GAE)
18) Calculate policy loss using PPO Clipped Objective
19) Calculate value function loss
20) Update Actor-Critic networks
21) Update Block-IP-Testcase weights (KG backpropagation)

Step 5: Iteration and Evaluation

22) When new design versions are released, proceed with new states and actions (Testcase selection)
23) Periodic KG validation and PPO learning rate adjustment at evaluation intervals

Key Point: Learning from immutable historical progression of design versions and their corresponding verification results, with simultaneous KG(GNN) ←KG Edge Weights→ PPO Agent network optimization.

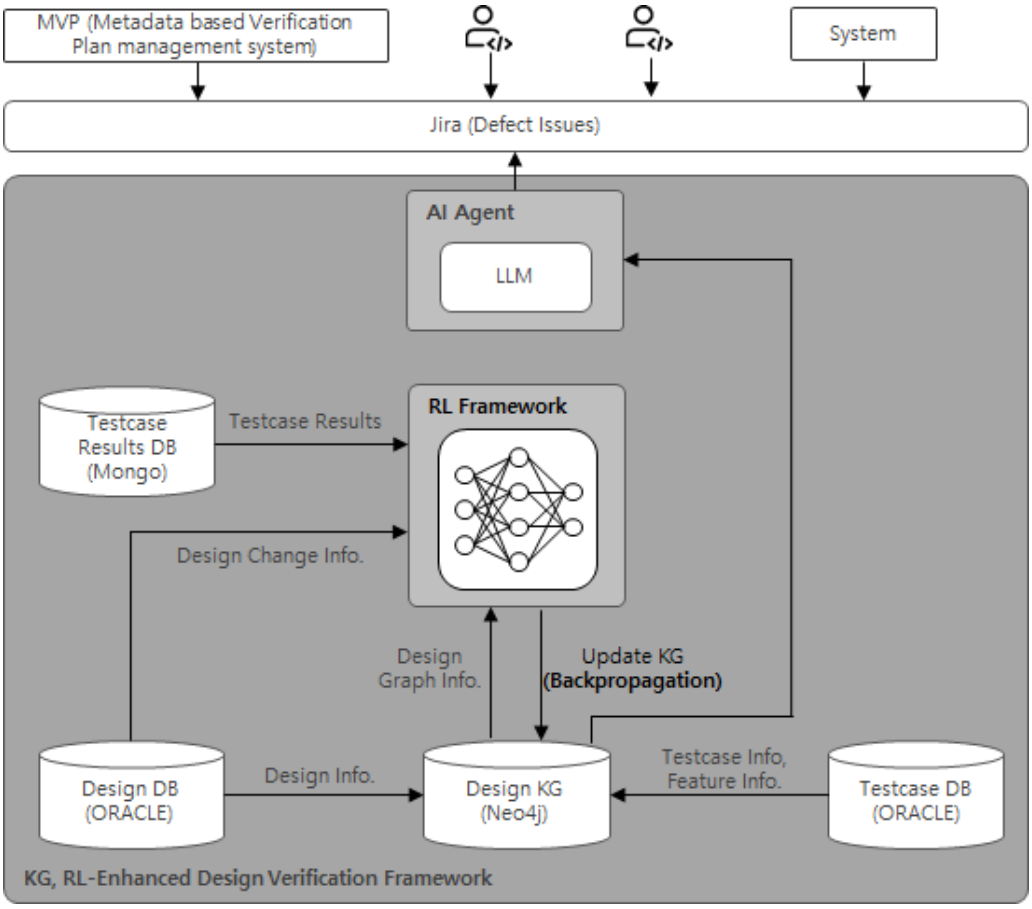

Figure 2 KG, RL-Enhanced Design Verification Framework

C. Experimental Results

- **Evaluation Metrics**

To quantitatively evaluate the effectiveness of the proposed RL-enhanced Knowledge Graph, we define the following metrics.

Table 1. Evaluation Metrics

Metric	Definition
Targeting Accuracy (TA)	Ratio of correctly identified design-changed IPs among all IPs predicted to be affected by test result changes
Coverage Recovery (CR)	Ratio of test coverage correctly attributed back to the actual changed IPs
Redundant Test Ratio (RTR)	Percentage of executed tests whose result changes are unrelated to actual design changes

- **Quantitative Comparison**

The performance of a static KG (constructed only from design and test plan connectivity) is compared against the KG weighted via RL (PPO).

Table 2. Quantitative Benefit Comparison **(A Block: IP_Type1, C Block: IP_Type2, I Block: IP_Type1, P Block: IP_Type2)**

Metric	KG Only	Trace Accuracy	Improvement
Targeting Accuracy (TA)	35/97 = 36.1%	65/97 = 67%	↑ ~31%
Coverage Recovery (CR)	7/112=6.3% (A)	13/112=11.6% (A)	↑ 2.2~9.1%
	61/666=9.2% (C)	122/666=18.3% (C)	
	11/877=1.3% (I)	31/877=3.5% (I)	
	4/26=15.4% (P)	5/26=19.2% (P)	
Redundant Test Ratio (RTR)	93.8%	88.4%	↑ 2.3~9.2%
	90.8%	81.7%	
	98.7%	96.5%	
	84.6%	80.8%	

- **Key Observations**

Based on the quantitative evaluation results, the following key observations can be drawn:

1) Comprehensive Performance Improvement

The RL-enhanced KG demonstrates consistent improvements across all evaluation metrics compared to the static KG approach. The improvements range from 31% in Targeting Accuracy to 2.2-9.2% in Coverage Recovery across different blocks.

2) Enhanced Test Impact Attribution

Coverage Recovery (CR) shows significant improvement from 6.3-15.4% (KG Only) to 11.6-19.2% (KG+RL). The most substantial improvement occurs in Block C (IP_Type2), where CR nearly doubles from 9.2% to 18.3%, indicating the RL framework's ability to better attribute test changes to their root causes.

3) Reduced Verification Redundancy

Redundant Test Ratio (RTR) substantially decreases across all blocks, with improvements of 2.3-9.2%. Block C (IP_Type2) shows the most significant reduction from 90.8% to 81.7%, suggesting that RL learning effectively identifies and eliminates irrelevant test relationships.

4) IP Type-Dependent Performance

The framework exhibits different performance patterns based on IP characteristics:

IP_Type1 blocks (A, I): More modest but consistent improvements

IP_Type2 blocks (C, P): Greater improvements in CR and RTR reduction, indicating the RL framework handles complex interconnect relationships more effectively

5) Trade-off Analysis

While Targeting Accuracy improves significantly (36.1% → 67%), the framework adopts a precision-recall optimization strategy. The increased detection of affected IPs (including those without direct design changes)

provides comprehensive verification coverage at the cost of some precision, which aligns with risk-averse verification practices.

6) Scalability and Consistency

The consistent improvement patterns across diverse block types (ranging from 12-37 IPs) demonstrate the framework's scalability and reliability across different SoC components and verification scenarios.

D. Comparative Analysis: PPO vs DQN

In this study, we employed Proximal Policy Optimization (PPO) as the primary RL algorithm and conducted comparative experiments using Deep Q-Network (DQN) to evaluate performance differences in hardware verification contexts. For SoC verification environments requiring "risk-based prioritization," policy gradient methods (PPO) showed superior generalization performance compared to value-based approaches (DQN). The probabilistic action selection of PPO better aligns with verification tasks requiring balanced exploration of uncertain design spaces while maintaining focus on high-risk areas.

E. Conclusion

This paper demonstrated that static KG constructed solely from SoC design information and test plan connectivity are insufficient for accurately predicting the impact of design changes on verification results. Experimental analysis showed that test result variations frequently appear in IPs without direct design modifications, often exhibiting higher test change ratios than actually changed IPs. This behavior leads to low targeting accuracy, inflated redundancy, and inefficient verification cycles. To address this limitation, we proposed a reinforcement learning–based Knowledge Graph weighting approach using Proximal Policy Optimization (PPO). By incorporating test result changes as reward signals and reverse-tracing their origins, the learned KG captures dynamic execution-level influence relationships rather than simple structural connectivity. As a result, the proposed method significantly reduces redundant tests, accelerates coverage recovery, and improves the accuracy of identifying design-relevant IPs and tests.

Importantly, while the proposed approach enables efficient prioritization and impact-focused verification, it does not eliminate the necessity of full regression testing. For achieving verification completeness, executing full regression tests in the background remains essential to detect hidden or low-probability errors that may not be immediately exposed through targeted testing. The proposed framework should therefore be viewed as a complementary intelligence layer that enhances verification efficiency and responsiveness, while preserving full regression as a safety net for comprehensive validation.

Overall, this work demonstrates that integrating reinforcement learning with KG provides a practical and scalable path toward more intelligent, adaptive, and efficient SoC verification, without compromising the rigor required to uncover hidden design flaws.

F. Future Work

Several research directions remain to enhance the proposed framework:

1. Stage-Specific Graph Models: Future work will develop specialized Graph Neural Networks (GNNs) optimized for different development phases (architectural exploration, block-level integration, full SoC block and IP verification). Multiple specialized models will replace the unified graph approach for improved efficiency and accuracy.

2. Error Semantics Integration: Rather than only tracking test result changes, the system will incorporate error type information (functional mismatches, timing violations, protocol errors). This enables fine-grained learning of design-verification relationships and context-aware impact analysis.

3. Dynamic Pipeline Architecture: A continuous data-training-service pipeline will replace static offline training. The KG, learning models, and inference services will incrementally adapt to new design revisions, test results, and error patterns in real-time.

4. AI Agent Deployment: The ultimate goal is deploying the framework as an AI Agent for automated verification environments. The agent will autonomously analyze results, prioritize tests, guide debugging, and accumulate experience to enhance verification workflows.

These directions aim to transform the approach into a practical, scalable intelligence layer for next-generation design verification automation.

REFERENCES

[1] S. Kim and H. Park, "DRL-Verify: Deep Reinforcement Learning for Exhaustive SoC Verification," 2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1-9, Nov. 2023

[2] Bernal Jim énez Guti érrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, Yu Su, "HippoRAG: Neurobiologically Inspired Long-Term Memory for Large Language Models" NeurIPS 2024

[3] Seonghee Yim, Hanna, Sunchang Choi and Seonil Brian Choi, "Accelerating SOC Verification using Process Automation and Integration" Design Verification Conference (DVCon), 2020

[4] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, Verification Methodology Manual for SystemVerilog. Springer, 2006.