

VeriRepair: Toward Reliable LLM-Based RTL Repair via CoT-Supervised Multi-Objective Fine-Tuning and Hybrid Retrieval

Lei Peng¹, Aijiao Cui¹, Yier Jin²

¹ Harbin Institute of Technology (Shenzhen)

²University of Science and Technology of China

Abstract—Ensuring the reliability of Register-Transfer Level (RTL) designs is critical, yet automated Verilog repair remains challenging due to requirements on synthesizability, timing correctness, and functional consistency. Existing LLM-based approaches rely on heuristic prompting, lack structured reasoning, and are trained on narrow datasets, which limits generalization and leads to logically inconsistent fixes. We present *VeriRepair*, the first novel framework to introduce Chain-of-Thought (CoT) supervision into hardware repair, combined multi-objective fine-tuning with a hybrid retrieval-augmented inference mechanism. We construct a 13k-pair RTL bug-fix dataset, covering more than 40 error types across six categories and enriched with reasoning annotations. The model is jointly fine-tuned on repaired code and reasoning traces, yielding more accurate and interpretable fixes. During inference, a hybrid retriever leverages semantic and structural similarity to guide patch generation. Experiments demonstrate that *VeriRepair* attains 76.6% Top-1 accuracy, surpassing VeriDebug by 20.1% and CirFix by 44.9%. Moreover, the framework is readily deployable in real industrial flows, integrating with pre-synthesis lint/fix pipelines and simulation- or UVM-based verification. The dataset is open source and available on GitHub: <https://github.com/90ICEDA/verirepair>.

Keywords—Code Repair, LLMs, AST, Chain-of-Thought Reasoning, RAG

I. INTRODUCTION

As process technology advances, billions of transistors can now be integrated into a single die. Consequently, the design of highly integrated chips at each high level of abstraction has become more complex. A Register-Transfer Level (RTL) design may consist of tens of thousands of code lines. This results in a greater probability of design errors. However, repairing these errors efficiently is challenging, as RTL design must satisfy strict requirements on synthesizability, timing closure, and functional correctness [1], [3]. As a result, error repair becomes a protracted and increasingly demanding process. Conventional manual repair not only require substantial expert knowledge but also heavily rely on the intensive use of simulation and verification tools. In contrast, automated RTL repair can offer higher efficiency and more robust logic and system-level correctness.

The early automated RTL repair frameworks, such as CirFix [38] and RTL-Repair [39], relied on predefined templates and

manual testbenches to patch designs. Thus, their scalability and applicability were limited. To overcome these constraints, recent research has turned to Large Language Models (LLMs). With their advanced semantic comprehension capabilities, LLMs have shown significant potential in various code-related tasks, including code generation [8], [15], [26], assertion generation [6]-[7], [9], and bug repair [1], [3]-[5], [14], [16], [21], [30]. These strengths make LLMs highly promising for efficient automated RTL code repair. For instance, RTLFixer [1] demonstrated that syntax errors in Verilog could be automatically repaired using prompt-based methods. MEIC [31] further improved prompt design through multi-round interaction. Subsequently, B. Ahmad et al. [3] extended this approach to security-critical vulnerabilities by leveraging carefully crafted prompts and ensembles of multiple LLMs. VeriDebug [4] adopted a dataset-driven training strategy to further enhance repair performance. A common feature across these methods is their focus on automated syntax-level repair of Verilog code, with performance gains achieved through either prompt engineering or dataset-driven training.

Noted that besides correcting syntax, effective RTL repair requires a deep understanding of developer's intent while ensuring patches to be aligned with the program's logic. Existing LLM-based approaches always focus on the optimization of statistical next-token model over code corpora [4] and usually produce patches satisfying compiling but failing to preserve semantic or functional consistency. This limitation is amplified by shortcomings in the training data, since token sequences are excessively long and bug types are narrow, leaving models ill-prepared for the diversity of real RTL designs. Meanwhile, effective repair must be grounded in instance-specific context. However, retrieval in current repair systems remains superficial, as it is typically confined to static knowledge bases or generic documentation. Such retrieval provides only general references while overlooking the structural and constraint-driven nature of RTL design, leaving a significant gap between token-level similarity and the contextual demands of real-world RTL repair.

To address these challenges, we propose *VeriRepair*, a unified RTL repair framework that combines CoT-supervised fine-tuning with a hybrid retrieval mechanism leveraging semantic and AST-structural similarity. Unlike prior methods limited to syntax-level fixes, *VeriRepair* integrates reasoning

supervision with a focus on preserving RTL design consistency. We construct the first large-scale RTL repair dataset of 13k bug-fix pairs with CoT traces across 40+ error types, providing a standardized benchmark. Collectively, these contributions establish a new paradigm for accurate and reliable RTL repair. To the best of our knowledge, *VeriRepair* is the first framework to bring Chain-of-Thought supervision into hardware code tasks and the first to leverage structure-aware retrieval in this domain. Our contributions are summarized in three aspects: as follows:

AST-Driven Dataset and Reasoning-Supervised Training: We construct an RTL repair dataset by integrating AST-guided error injection with LLM-generated CoT reasoning, ensuring correctness, synthesizability, and diverse coverage. Building on this, we adopt a multi-objective training paradigm that supervises repaired code and reasoning traces, embedding stepwise reasoning into the process to enhance accuracy, robustness, and interpretability.

Hybrid Retrieval-Augmented Inference Mechanism: We design a hybrid retrieval approach combining semantic similarity with AST-structural alignment, augmented by filtering. This retrieves structurally consistent bug-fix exemplars and integrates them into inference, enabling context-aware patch generation and bridging the gap between semantic retrieval and RTL-specific constraints.

Empirical Validation and Performance Gains: On benchmarks covering logic, timing, FSM, protocol, memory, and multi-bug errors, CoT-supervised fine-tuning improves repair accuracy by 19.5% over VeriDebug. With retrieval-augmented inference, VeriRepair achieves 76.6%/85.9%/87.2% at Top-1/Top-5/Top-10, surpassing VeriDebug by 20.1%, 19.1%, and 12.0%.

II. BACKGROUND

In this section, we begin by reviewing LLM-based approaches to code repair in Section A. We then introduce CoT reasoning as a paradigm for structured intermediate reasoning in Section B. Finally, we discuss RAG in Section C, which enriches LLM outputs by incorporating external contextual knowledge.

A. Code Repair using Large Language Models

The goal of code repair is to accurately identify errors and generate functionally correct patches. With the advent of LLMs and their strong semantic understanding, traditional manual debugging workflows are being transformed into more efficient automated paradigms. However, this progress introduces challenges: LLMs often hallucinate, producing patches that are syntactically valid but functionally incorrect. This highlights the insufficiency of relying solely on syntactic correctness in hardware code repair. Current LLM-based Verilog repair approaches fall into two paradigms: prompt tuning and parameter fine-tuning [1] [3] [4] [31]. Prompt tuning depends heavily on manually designed prompts, limiting generalization and robustness [1] [3]. Parameter fine-tuning instead demands high-quality domain-specific datasets and clear optimization objectives, introducing significant obstacles in dataset construction and training design [4].

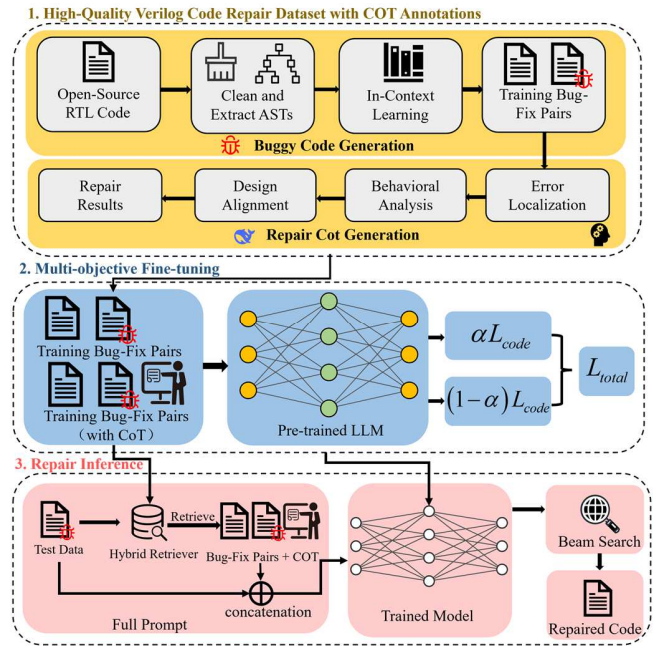


Fig 1. Overall Framework of *VeriRepair*

B. Chain-of-Thought

Chain-of-Thought reasoning is a paradigm where LLMs explicitly generate intermediate reasoning steps before the final output [29]. Unlike direct input-output mapping, CoT decomposes complex tasks into interpretable sub-steps, yielding substantial gains in logical reasoning and problem solving [5]. For code repair, defects often demand semantic reasoning and consistency checks. Incorporating CoT enables stepwise analysis such as error localization, behavioral assessment, specification verification, and repair strategy formulation before patch generation, thereby improving both correctness and interpretability.

C. Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) has become a key paradigm for enhancing LLMs [2]. While LLMs produce fluent, contextually relevant outputs, their knowledge is limited by static memory. RAG overcomes this by dynamically integrating external knowledge during inference. Instead of relying solely on internal parameters, the model retrieves semantically relevant information from curated sources and fuses it with generative reasoning. This not only enriches contextual grounding but also reduces factual errors, yielding more consistent, well-supported outputs. In code generation, prior work shows RAG can significantly improve quality [6] [20].

III. VERIREPAIR

A. Method Overview

As shown in Fig. 1, we propose a unified framework for Verilog code repair with LLMs, consisting of three components: a dataset with CoT annotations for both

supervision and retrieval, a multi-objective fine-tuning stage that jointly optimizes code repair and reasoning, and a hybrid retrieval mechanism at inference that combines semantic and structural cues to generate context-aware repairs.

B. CoT-Enhanced Dataset Construction

Existing datasets lack sufficient diversity and explicit reasoning signals, limiting the effectiveness for realistic RTL repair [4]. To overcome these issues, we construct a systematic dataset using multi-category, multi-level error injection, as outlined in module 1 of Fig. 1. Fig. 2 illustrates this process, showing targeted vulnerability injection and corresponding CoT generation in a representative design. Automated filtering, compilation, and quality scoring ensure dataset validity and representativeness. Finally, CoT annotations are included alongside repair labels, enabling explicit reasoning guidance that enhances model repair performance.

1) Collecting Correct Verilog Designs and Extracting Abstract Syntax Trees

We collected approximately 6,500 Verilog designs from multiple open-source projects, including RTLCoder, OpenCores, CVA6, OpenPiton, and Ethernet protocol implementations [8], [19], [35]. Each design was compiled and functionally tested, then evaluated across three dimensions: code size, module complexity, and synthesizability. A weighted quality score was computed from these factors, and only high-scoring designs were retained as the base sample set for subsequent error injection.

Preliminary experiments reveal that LLMs often injected errors into non-critical regions, such as unused signals or functionally irrelevant branches, which have little impact on actual circuit behavior. For example, in Fig. 2(a), if simply modifying the reset assignment on line 3 from `count <= 4'b0000` to another constant only shifts the initial value without affecting the counter’s saturation behavior, providing limited training value.

To focus on meaningful modifications, we parsed designs into Abstract Syntax Trees (ASTs) using PyVerilog [17] and identified critical regions, including (i) control paths such as if-else branches, (ii) state machine conditions and transitions, and (iii) core logic involving arithmetic operations and register updates. The identified regions were then explicitly encoded into the prompt and passed to the LLM for guided injection. The AST construction process, shown in Fig. 2, maps the original Verilog code (a) to its generated AST (b), making structural dependencies more visible. Error injection was then targeted to these regions, ensuring that training data reflects impactful and behaviorally significant changes in the design. We further required injected designs to compile successfully and exhibit different output behavior in simulation, while equivalence-preserving or trivial modifications were automatically detected and filtered out.

2) Error Pattern Templates and Context-Guided Generation

As illustrated in Table I, we define a taxonomy of common Verilog error patterns across multiple categories, each instantiated into parameterized injection templates that specify

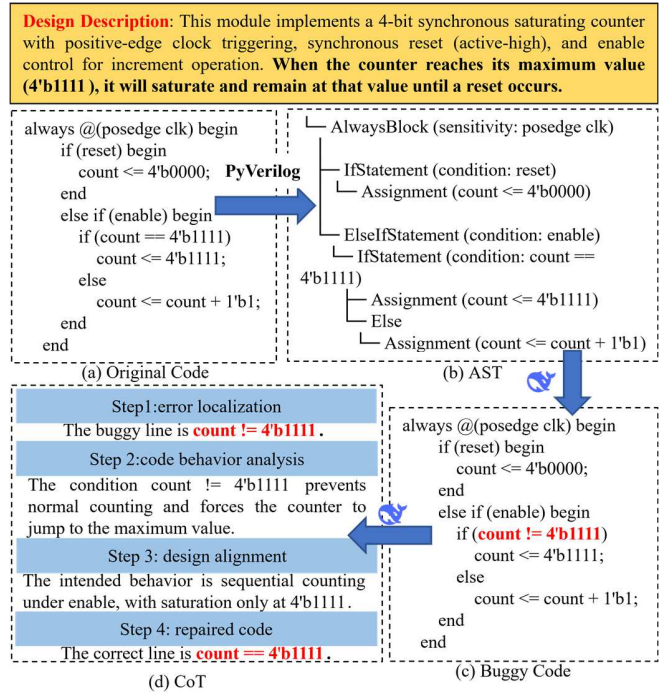


Fig. 2. Example of Bug Injection and CoT Reasoning Generation in Dataset Construction

triggering conditions, target AST nodes, and modification operators. During error synthesis, LLMs are guided with in-context prompts combining the original code, pattern descriptions, and few-shot examples, while AST constraints enforce valid insertion points and template instantiations. This process yields diverse yet semantically valid error variants. Table I shows a subset of representative examples, while the complete taxonomy encompasses over 40 Verilog error types across six major categories.

3) Chain-of-Thought Reasoning Generation

We leverage LLMs to generate CoT sequences for code repair, providing intermediate supervision that captures the reasoning trajectory from error localization to patch generation. The CoT is organized into four stages inspired by typical engineering workflow: error localization, code behavior analysis, design consistency check, and causal repair derivation. Each stage is generated sequentially by the model, culminating in a syntactically valid and specification-compliant patch. As illustrated in Fig. 2(d), this pipeline produces paired reasoning–repair sequences that form the basis of our dataset construction.

4) Quality Filtering and Dataset Balancing

To prevent the dataset from being dominated by easily generated but low-value error patterns, we apply post-generation quality filtering and distribution balancing. Specifically, we (i) enforce stratified coverage across error categories to maintain proportional representation, (ii) constrain the token-length distribution to further mitigate the impact of excessively long sequences on batch size and training stability, and (iii) require all samples to compile and pass structural checks, thereby ensuring retained instances are syntactically correct and synthesizable.

C. CoT-Supervised Multi-Objective Fine-Tuning

Multi-Objective Fine-Tuning is a training strategy that simultaneously optimizes multiple output objectives within a unified model, enabling it to balance diverse capabilities and improve overall performance in complex tasks. In this work, we adopt a multi-objective optimization framework for hardware code repair, jointly learning two complementary sub-tasks: (1) **Bug-Fix Code Generation**: Producing syntactically correct and functionally compliant repaired code that resolves the errors in the input hardware design. (2) **Repair Reasoning Chain Generation**: Generating a structured reasoning sequence that captures the full repair logic, including error localization, behavioral analysis, design-specification alignment, and modification strategy formulation.

Specifically, let the defective code input be represented as a sequence of n tokens $X = \{x_1, x_2, \dots, x_n\}$, where each x_i denotes a single token from the original code, corresponding to a fundamental semantic unit including keywords, identifiers, operators. For **subtask (1)**, the objective is to generate the repaired code sequence $Y = \{y_1, y_2, \dots, y_m\}$, where m denotes the token length of the repaired code. We adopt an autoregressive generation strategy, in which the prediction of the current token y_t depends only on the input sequence X and the previously generated tokens $y_{<t}$:

$$p_\theta(Y|X) = \prod_{t=1}^m p_\theta(y_t | X, y_{<t}) \quad (1)$$

The training objective is to minimize the cross-entropy loss between the ground-truth distribution Q and the model-predicted probability distribution P :

$$L_{code} = -\sum_{t=1}^m Q(y_t | X, y_{<t}) \log P_\theta(y_t | X, y_{<t}) \quad (2)$$

For **subtask (2)**, the goal is to generate a reasoning chain containing error localization, behavioral analysis, design alignment, and modification strategy $Z = \{z_1, z_2, \dots, z_k\}$, where k is the token length of the reasoning chain. Its loss function is defined as:

$$L_{CoT} = -\sum_{t=1}^k Q(z_t | X, y_{<t}) \log P_\theta(z_t | X, y_{<t}) \quad (3)$$

To jointly optimize these two objectives, we compute the respective losses for code repair generation and CoT generation, and combine them as follows:

$$L_{total} = \alpha \cdot L_{code} + (1 - \alpha) \cdot L_{CoT} \quad (4)$$

Where $\alpha \in [0, 1]$ is a tunable hyperparameter that controls the relative importance of the two tasks during joint optimization.

D. Retrieval-Augmented Generation for Repair Reasoning

After model training, we incorporate a RAG mechanism to enhance contextual relevance and reasoning consistency during repair. Given a buggy RTL design, the system retrieves semantically related and structurally similar bug-fix pairs from a pre-built database. These instances serve as external knowledge cues, enriching the model's input. As shown in Fig. 3, both the buggy design and candidate pairs are encoded into

semantic and AST-based embeddings, fused into hybrid representations, and searched to identify the most relevant exemplars. The retrieved pairs are then integrated into the model's input, guiding the LLM toward context-aware and specification-aligned repairs.

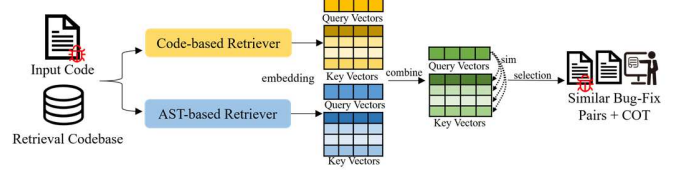


Fig 3. Overview of RAG in the VeriRepair

Limitations of Conventional Retrieval: In existing code-related tasks, retrieval has often relied on information retrieval techniques such as Jaccard similarity [20], which measure lexical overlap. While efficient, these methods fail to capture deeper semantic or structural dependencies. Consequently, performance deteriorates when handling semantically equivalent but syntactically varied code snippets.

Hybrid Semantic-Structural Retrieval: To overcome this, we design a hybrid retriever that combines semantic embeddings from raw code with structural embeddings from ASTs. Semantic embeddings capture contextual meaning, variable usage, and functional intent, while AST embeddings preserve hierarchical constructs such as control flow and data dependencies. Their complementarity ensures that retrieved bug-fix pairs are not only contextually relevant but also structurally consistent—an essential property for RTL repair.

Concretely, source code and its AST are encoded separately. The AST is first linearized via pre-order traversal to form a sequence, and both code and AST sequences are embedded with Transformer encoders followed by mean pooling. We adopt UniXcoder [11] as the embedding model, since its pretraining incorporates AST information, making it naturally suited for our design. The resulting embeddings are fused into a single vector representation. Similarity between the query and candidate bug-fix pairs is then computed via cosine similarity over fused vectors, enabling retrieval that jointly reflects semantic and structural correspondence. Retrieved pairs are subsequently ranked and filtered to provide high-quality exemplars within the token budget. The cosine similarity is defined as:

$$Sim_{cos}(\mathbf{U}, \mathbf{V}) = \frac{\mathbf{U} \cdot \mathbf{V}}{\|\mathbf{U}\| \times \|\mathbf{V}\|} = \frac{\sum_{i=1}^n U_i V_i}{\sqrt{\sum_{i=1}^n U_i^2} \sqrt{\sum_{i=1}^n V_i^2}} \quad (5)$$

Where \mathbf{U} and \mathbf{V} denote the fused embedding of the query code and a candidate bug-fix pair, and n is the embedding dimension. **Filtering Strategy:** While retrieval improves repair performance, adding too many examples risks token overflow and introduces additional noise. To mitigate this, retrieved candidates are ranked by similarity and filtered using a threshold before integration into the prompt. This ensures that only the most relevant and informative exemplars are retained, reducing spurious context while maximizing the practical utility of retrieval under a fixed token budget.

TABLE I

REPRESENTATIVE VERILOG BUG CATEGORIES AND CORRESPONDING FIXES IN THE CONSTRUCTED DATASET

| Bug Type | Bug Description | Buggy Example | Repair Example |
|---------------------|--|--|--|
| Combinational Logic | Missing assignments on some paths cause unintended latches | <code>if (en) y = d;</code> | <code>if (en) y = d; else y = '0;</code> |
| Combinational Logic | Operator precedence error changes semantics | <code>assign y = a + b & c;</code> | <code>assign y = (a + b) & c;</code> |
| Sequential Logic | Data sampling hazard causing metastability | <code>q <= async_signal;</code> | <code>sync1 <= async_signal; sync2 <= sync1; q <= sync2;</code> |
| FSM | Missing state transition | <code>case(state) STATE_INIT: state <= STATE_INIT;</code> | <code>case(state) STATE_INIT: state <= STATE_START;</code> |
| Interface/Protocol | valid/ready handshake ignored | <code>assign out_valid = in_valid;</code> | <code>assign out_valid = in_valid & ready;</code> |
| Memory | RAM not initialized | <code>reg [7:0] mem [0:15];</code> | <code>reg [7:0] mem [0:15] = '{default:0};</code> |
| Expression/Boundary | Sign extension missing causes truncation | <code>wire [15:0] y = x8; // x8 is signed [7:0]</code> | <code>wire [15:0] y = {{8{x8[7]}}, x8};</code> |
| Tri-state Control | Missing high-Z assignment causes bus contention | <code>assign bus = (en) ? data : 0;</code> | <code>assign bus = (en) ? data : 'z;</code> |

IV. EXPERIMENTAL EVALUATION

This section presents experiments evaluating performance on the Verilog repair task and analyzes the factors influencing it.

A. Experimental Setup

We implement our framework using HuggingFace Transformers [18] [25] and optimize training with DeepSpeed ZeRO-2 [13] for efficient distributed execution. The baseline model is DeepSeek-Coder-7b-Instruct [12]. Experiments are conducted on a cluster of eight NVIDIA A100 GPUs. For retrieval-augmented reasoning, we encode both the raw source code and its linearized AST with UniXcoder [11], producing hybrid embeddings that integrate semantic and structural signals. A similarity-based selector filters high-relevance bug-fix examples under the 2k-token budget, ensuring that only the most useful references are injected into the context. This strategy enhances both the relevance and precision of generated repairs.

B. Evaluation Metrics

We evaluate model performance using the pass@k metric, which estimates the probability that at least one correct repair is contained within the top- k generated candidates. This metric has been widely adopted in program repair and code generation research as it directly reflects the usefulness of the model to developers [4-6]. Formally, given N generated candidates and C correct ones, pass@k is defined as:

$$\text{pass@k} = E_{\text{problems}} \left[1 - \frac{C(N-C, k)}{C(N, k)} \right] \quad (6)$$

This formulation accounts for the combinatorial probability that none of the selected k candidates is correct, making the estimate statistically sound. In practice, we report pass@1 , pass@5 , and pass@10 , which measure whether a correct repair appears among the top 1, 5, or 10 candidates. To determine correctness, we first check whether the generated patch exactly matches the ground-truth fix; if not, we additionally apply formal equivalence checking to verify functional consistency and behavioral alignment.

C. Hyperparameter Settings

Supervised Fine-Tuning (SFT) Stage: During the SFT stage, we set both weights to 0.5 to strike a balance between repair accuracy and reasoning consistency. Optimization is performed using the AdamW optimizer [10] with an initial learning rate of 2×10^{-5} . The learning rate schedule includes a 500-step linear warmup followed by linear decay. Training is conducted with a global batch size of 64, gradient accumulation of 4, and FP16 mixed precision. We run for three epochs using DeepSpeed ZeRO-2 across eight NVIDIA A100 GPUs.

RAG Inference Stage: At inference, we employ a RAG mechanism with a similarity threshold of 0.82, derived from sensitivity analysis over [0.75, 0.90], where 0.82 gave the best balance between filtering noise and retaining context. Retrieved candidates above this threshold are ranked and inserted into the model’s input until the 2048-token limit is reached. Decoding uses beam search with size 5. The top-5 patches are then validated for syntax with Icarus Verilog [28] and structural integrity with Yosys [37].

D. CoT-Based Dataset Evaluation

We constructed a dataset of about 13,000 Verilog bug-fix pairs with repair reasoning chains. It spans diverse vulnerability categories, such as combinational logic errors, sequential and timing faults, FSM defects, interface and protocol violations, memory/storage bugs, and syntax mistakes, with combinational logic and FSM errors being most common. As shown in Fig. 4, combinational logic accounts for 26.3% of the training set and 22.2% of the inference set, while FSM and protocol bugs also occupy notable proportions. Category distributions are kept consistent across splits, ensuring representative coverage. To increase diversity, we added about 200 multi-bug cases and some error-free samples. The dataset is partitioned into disjoint training, validation, and test subsets, and we also build 1,000-sample inference test set mirroring bug distributions while remaining isolated from training data, enabling fair evaluation. This inference set further integrates benchmark samples from VeriDebug [4], CirFix [38], and RTL-Repair [39] for

comparison. To ensure fairness, the retrieval database is built independently from held-out validation/test data, with no overlap with training; only the test split is used during evaluation.

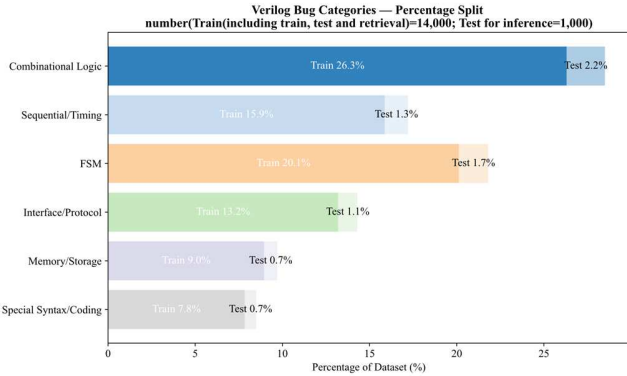


Fig 4. Distribution of Error Types in the Constructed Verilog Dataset

E. Evaluation of Multi-Objective Fine-Tuning

TABLE II
COMPARISON OF VERILOG REPAIR ACCURACY ACROSS
BASELINES AND *VERIREPAIR* VARIANTS

| Type | Model | Accuracy (%) | | |
|-----------|--------------------------------------|--------------|-------------|-------------|
| | | Top-1 | Top-5 | Top-10 |
| Baselines | Deepseek-coder | 8.7 | 20.4 | 36.9 |
| | CirFix | 31.7 | | |
| | VeriDebug | 56.5 | 66.8 | 75.2 |
| Ours | <i>VeriRepair</i> (single) | 49.2 | 60.5 | 69.2 |
| | <i>VeriRepair</i> _{CoT} | 65.3 | 73.2 | 78.7 |
| | <i>VeriRepair</i> _{RAG} | 56.3 | 66.0 | 69.5 |
| | <i>VeriRepair</i> _{CoT+RAG} | 76.6 | 85.9 | 87.2 |

VeriRepair(single): single-objective training on code repair.

Table II summarizes the accuracy of baselines and our proposed variants under Top-1, Top-5, and Top-10 metrics. The generic Deepseek-coder-7b-Instruct performs poorly (Top-1: 8.7%), underscoring its lack of domain adaptation. CirFix achieves 31.7% Top-1 accuracy, better than Deepseek-coder but still significantly lower than dataset-tailored approaches. Training on our dataset (*VeriRepair*(single)) brings substantial gains, reaching 49.2% Top-1 accuracy, which validates the importance of domain-tailored data and fine-tuning. Incorporating CoT (*VeriRepair*_{CoT}) further improves accuracy to 65.3% (+16.1 over single-objective), confirming the effectiveness of multi-objective fine-tuning. RAG alone (*VeriRepair*_{RAG}) provides moderate improvement compared to VeriDebug, but its true strength emerges when combined with CoT. The joint framework (*VeriRepair*_{CoT+RAG}) achieves the best results—76.6%, 85.9%, and 87.2% across Top-1, Top-5, and Top-10—surpassing VeriDebug by +20.1, +19.1, and +12.0 points. These results confirm that CoT supervision and retrieval are complementary, together pushing RTL repair to a new state of the art.

F. Retrieval-Augmented Reasoning Evaluation

TABLE III
RETRIEVAL CONFIGURATION AND ITS IMPACT ON VERILOG
REPAIR ACCURACY IN *VERIREPAIR*

| Retrieval Strategy | Retriever Filter | Accuracy (%) | | |
|--------------------|------------------|--------------|-------------|-------------|
| | | Top-1 | Top-5 | Top-10 |
| No Retrieval | | 65.3 | 73.2 | 78.7 |
| Code only | × | 69.0 | 76.6 | 81.8 |
| Code only | ✓ | 69.6 | 77.1 | 81.9 |
| AST+Code | × | 73.1 | 81.6 | 83.8 |
| AST+Code | ✓ | 76.6 | 85.9 | 87.2 |

Table III reports the impact of different retrieval configurations on Verilog repair accuracy. The results show a clear upward trend: incorporating retrieval consistently improves performance over the no-retrieval baseline (65.3%/73.2%/78.7%). Using code-only embeddings with filtering yields 69.6% Top-1 and 81.9% Top-10 accuracy, confirming that even semantic cues alone provide valuable contextual guidance. Incorporating AST information further strengthens retrieval: the AST+Code setup achieves 73.1%, 81.6%, and 83.8% without filtering, while enabling filtering raises Top-1 accuracy to 76.6% with consistent gains at higher ranks. These results highlight three insights: retrieval supplies valuable contextual cues that guide more accurate fixes, AST structures complement code semantics to yield higher-quality exemplars, and relevance filtering is crucial to suppress noise and maximize the benefit of retrieval.

CONCLUSION

In this paper, we presented *VeriRepair*, a novel RTL-aware framework for automated Verilog repair that unifies CoT supervision with a hybrid retrieval strategy that combines semantic and structural embeddings. Moving beyond prior methods that rely mainly on prompt engineering or code-only fine-tuning, *VeriRepair* introduces a reasoning-augmented multi-objective training paradigm, jointly optimizing syntactic correctness, semantic fidelity, and reasoning consistency. This design yields patches that are not only syntactically valid but also semantically faithful and functionally reliable, thereby addressing RTL-specific requirements. Experimental results validate these advantages, with *VeriRepair* achieving a Top-1 accuracy of 76.6%, achieving an improvement of 20.1 percentage points over VeriDebug. Beyond these improvements, *VeriRepair* is designed for seamless integration into industrial EDA workflows.

It can be deployed as a plugin in pre-synthesis lint/fix pipelines or simulation/UVM-based environments, where its reasoning traces provide interpretable justifications that complement traditional debugging. This plug-in nature reduces adoption barriers while improving the trustworthiness of automated fixes. Future work will extend *VeriRepair* to large-scale SoC designs and integrate formal verification feedback. These directions pave the way for trustworthy, LLM-driven assistants in RTL design and verification.

ACKNOWLEDGMENT

Our work was supported in part by the National Natural Science Foundation of China under Grant 62174045, in part by the Shenzhen Fundamental Science Research Foundation under Project JCYJ20240813104821029, and in part by the CCF-Huawei Hu Yanglin Fund under Grant CCF-HuaweiTC202403.

REFERENCES

- [1] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model," *Proc. DAC*, 2024, pp. 1–6.
- [2] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge Intensive NLP Tasks," *arXiv preprint arXiv:2005.11401*, 2020.
- [3] B. Ahmad *et al.*, "On Hardware Security Bug Code Fixes by Prompting Large Language Models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024.
- [4] N. Wang *et al.*, "VeriDebug: A unified LLM for Verilog debugging via contrastive embedding and guided correction," *arXiv preprint arXiv:2504.19099*, 2025.
- [5] B. Yang *et al.*, "MOREpair: Teaching LLMs to Repair Code via Multi-Objective Fine-Tuning," *ACM Trans. Softw. Eng. Methodol.*, May 2025. [online]. Available: <https://doi.org/10.1145/3735129>
- [6] Q. Zhang *et al.*, "Improving Retrieval-Augmented Deep Assertion Generation via Joint Training," *IEEE Transactions on Software Engineering*, vol. 51, pp. 1232–1247, 2025.
- [7] W. Fang *et al.*, "AssertLLM: Generating Hardware Verification Assertions from Design Specifications via Multi-LLMs," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–1.
- [8] S. Liu *et al.*, "RTLCoder: Fully Open-Source and Efficient LLM-Assisted RTL Code Generation Technique," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, pp. 1448–1461, 2025.
- [9] Y. Bai, G. Hamad, S. Suhaib, H. Ren, "AssertionForge: Enhancing Formal Verification Assertion Generation with Structured Representation of Specifications and RTL," *arXiv preprint arXiv:2503.19174*, 2025.
- [10] D. Kingma, J. Ba, "Adam: A method for stochastic optimization" *arXiv preprint arXiv:1412.6980*, 2014.
- [11] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.
- [12] D. Guo *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence," *arXiv preprint arXiv:2401.14196*, 2024.
- [13] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2020, pp. 3505–3506.
- [14] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: a T5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp.935–947.
- [15] N. Wang *et al.*, "Insights from Verification: Training a Verilog Generation LLM with Reinforcement Learning with Testbench Feedback," *arXiv preprint arXiv:2504.15804*, 2025.
- [16] F. Huq, M. Hasan, M. M. A. Haque, S. Mahbub, A. Iqbal, and T. Ahmed, "Review4Repair: Code review aided automatic program repairing," *Information and Software Technology* 143:106765, 2022.
- [17] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for Verilog HDL," in *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11*. Springer, 2015, pp. 451–460.
- [18] Hugging Face, "Model Repository," 2025. [Online]. Available: <https://huggingface.co/> [Accessed: Jun. 24, 2025].
- [19] OpenCores, "Open Source Hardware Community," 2025. [Online]. Available: <https://opencores.org/> [Accessed: Jun. 24, 2025].
- [20] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2450–2462.
- [21] F. Li, J. Jiang, J. Sun, and H. Zhang, "Hybrid automated program repair by combining large language models and program analysis," *arXiv preprint arXiv:2406.00992*, 2024.
- [22] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Software*, vol. 38(4): pp. 22–27, 2021.
- [23] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1430–1442.
- [24] N. Ho, L. Schmid, and S. Yun, "Large language models are reasoning teachers," *arXiv preprint arXiv:2212.10071*, 2022.
- [25] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
- [26] Q. Zhang, C. Fang, Y. Shang, T. Zhang, S. Yu, and Z. Chen, "No man is an island: Towards fully automatic programming by code search, code generation and program repair," *arXiv preprint arXiv:2409.03267*, 2024.
- [27] PyTorch, "PyTorch Documentation," 2025. [Online]. Available: <https://pytorch.org/> [Accessed: Jun. 24, 2025].
- [28] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, 2002.
- [29] J. Wei, *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 24824–24837.
- [30] X. Chen *et al.*, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.
- [31] K. Xu, *et al.*, "MEIC: Re-thinking RTL Debug Automation using LLMs," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 2024, Article 100, 1–9.
- [32] K. Min, *et al.*, "Improving LLM-Based Verilog Code Generation with Data Augmentation and RL," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.
- [33] I. Loshchilov, F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.
- [34] M. Chen, *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.
- [35] GitHub, "GitHub Repository," 2025. [Online]. Available: <https://github.com/> [Accessed: Jun. 24, 2025].
- [36] J. Zhang, C. Liu, and H. Li, "Understanding and Mitigating Errors of LLM-Generated RTL Code", *arXiv preprint arXiv:2508.05266*, 2025.
- [37] C. Wolf, J. Glaser, and J. Kepler. "Yosys—a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. Vol. 97. 2013.
- [38] H. Ahmad, Y. Huang, and W. Weimer, "CirFix: Automatically Repairing Defects in Hardware Design Code," In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 990–1003.
- [39] K. Laeufer *et al.*, "RTL-Repair: Fast Symbolic Repair of Hardware Design Code," In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 3, 2024, pp. 867–881.