Insights from Rights and Wrongs: A Large Language Model for Solving Assertion Failures in RTL Design

Jie Zhou[1,2], Youshu Ji[2], Ning Wang[3], Yuchen Hu[1,2], Xinyao Jiao[1,2], Bingkun Yao[3], Xinwei Fang[4], Shuai Zhao[5], Nan Guan[3], Zhe Jiang[1,2]
[1]School of Integrated Circuits, Southeast University, China
[2]National Center of Technology Innovation for EDA, China
[3]Department of Computer Science, City University of Hong Kong, Hong Kong
[4]Department of Computer Science, University of York, UK
[5]Department of Computer Science, Sun Yat-sen University, China

***Abstract*—SystemVerilog Assertions (SVAs) are essential for verifying Register Transfer Level (RTL) designs, as they can be embedded into key functional paths to detect unintended behaviours. During simulation, assertion failures occur when the design's behaviour deviates from expectations. Solving these failures, i.e., identifying and fixing the issues causing the deviation, requires analysing complex logical and timing relationships between multiple signals. This process heavily relies on human expertise, and there is currently no automatic tool available to assist with it. Here, we present AssertSolver, an open-source Large Language Model (LLM) specifically designed for solving assertion failures. By leveraging synthetic training data and learning from error responses to challenging cases, AssertSolver achieves a bug-fixing pass@1 metric of 88.54% on our testbench, significantly outperforming OpenAI's o1-preview by up to 11.97%. We release our model and testbench for public access to encourage further research: https://github.com/SEU-ACAL/reproduce-AssertSolver-DAC-25.**

I. Introduction

Functional verification is a crucial step in the modern Electronic Design Automation (EDA) process, ensuring that designs meet their specifications and perform correctly as intended [1], thereby mitigating the costly risks associated with silicon failures [2], [3]. SystemVerilog Assertions (SVAs), as one of the key methods in functional verification [4], capture potential errors in Register Transfer Level (RTL) designs by defining logical conditions and timing requirements. Unlike stimulus-based verification methods (e.g., using testbenches), SVAs not only facilitate stimulus-triggered checks but also enable formal verification [5]. Formal verification, which mathematically proves the consistency of the design under test (DUT) with the behaviour specified by the SVAs, provides higher functional coverage and addresses boundary conditions that stimulus-based methods may often overlook.

Despite the significant advantages of SVAs and progress in automating their generation [6]–[11], automatically solving assertion failures remains challenging. Assertion failures occur when the design exhibits unexpected behaviour during simulation. Effectively identifying and fixing these failures still relies heavily on human expertise to analyse intricate logical dependencies and timing relationships between multiple signals, which is time-consuming and labour-intensive. Fig. 1 illustrates that verification engineers must have an in-depth understanding of the design's functional intent and carefully deduce the causes of complex assertion failures.

Large Language Models (LLMs) have demonstrated significant capabilities in navigating through complex ideas, automating key steps in hardware design. They have improved a range of tasks, from code generation [12]–[16] to design verification [17]–[19], offering potential for solving assertion failures. However, while state-of-the-art (SOTA) LLMs such as OpenAI's GPT-4 and o1-preview demonstrate competence in such task, their performance is often subject to unnoticed changes, and they lack the ability to be retrained or fine-tuned by users to incorporate new data or information.

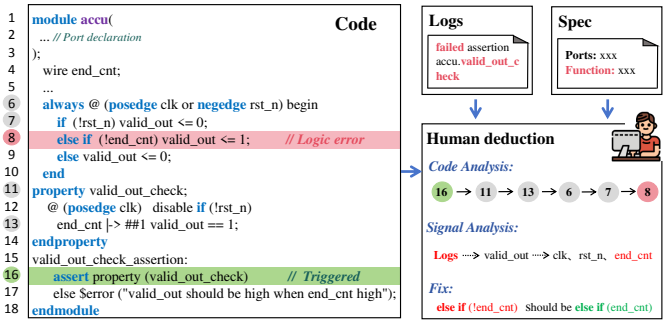


Fig. 1: The process of solving assertion failures in RTL verification. Engineers analyse the design specification and code signals to identify the root causes of assertion failures and implement fixes.

To address this limitation, we introduce an open-source LLM, AssertSolver, specifically designed for automatically solving assertion failures. As shown in Fig. 2, AssertSolver benefits from an innovative data augmentation method that enhances the representation of various assertion failure scenarios in the training dataset, which were previously underrepresented. AssertSolver is retrained and fine-tuned from the Deepseek-Coder-6.7b [20], using the dataset that incorporates not only significantly enhanced examples but also error responses to challenging cases. By learning from these "right" and "wrong" examples, AssertSolver demonstrates up to a 11.97% improvement in solving assertion failures compared to the OpenAI's o1-preview.

The main contributions of this paper are:

- Development of AssertSolver, an open-source domain-specific LLM fine-tuned for solving assertion failures, which we have made publicly available for early adoption;
- Implementation of a data augmentation method that automatically enrich scarce training dataset for assertion failures;
- Adoption of a new training strategy that improves learning from error responses to challenging cases;
- Publication of openly accessible benchmark for solving assertion failures, featuring over 900 instances across various bug types.

The rest of the paper is organised as follows: Section II introduces the data augmentation approach and the preparation of the training dataset. Section III provides a detailed description of our training strategies. Section IV outlines the research questions and the experimental setup. Section V presents a comprehensive evaluation to answer the research questions. Section VI concludes the paper.

II. Data Augmentation

To effectively train an LLM for solving assertion failures, it is essential to have access to a comprehensive training dataset. While current datasets contain a significant number of Verilog code

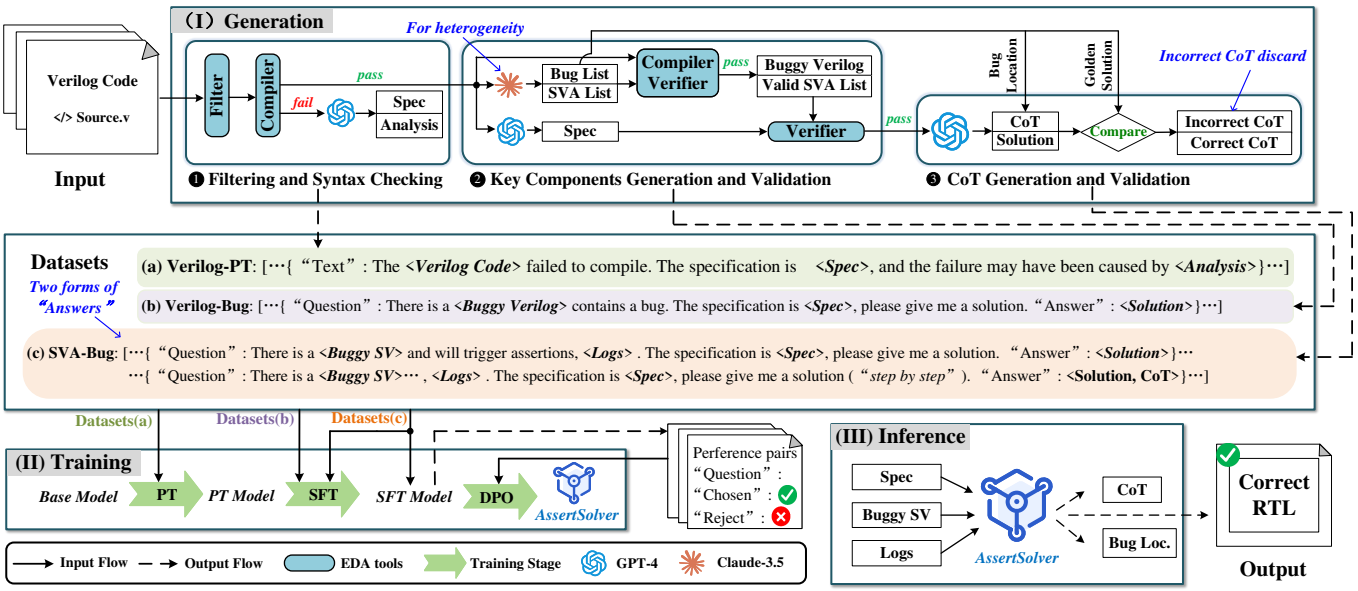


Fig. 2: AssertSolver overview: **(I)** shows the process for augmenting the training data. **(II)** describes the training strategy, consisting of pretraining (PT), Supervised Fine-Tuning (SFT), and Direct Preference Optimisation (DPO) which is used to guide learning from error responses to challenging cases. In the inference phase **(III)**, AssertSolver locates and fixes the bug based on the specification (Spec), buggy SystemVerilog (SV) code, and logs, while also providing bug location (Bug Loc.) and an explanation (CoT).

samples suitable for preliminary pretraining, they lack key components required for fine-tuning the model to this specific task. These components include design specifications, SVAs, details of bugs that could invalidate these assertions, and their corresponding verification outcomes. To address this gap, we propose a method that integrates LLMs and EDA tools to augment existing open-source Verilog code, thereby generating the necessary components for solving assertion failures. Additionally, to help users understand the model's problem-solving process, we have incorporated a Chain of Thought (CoT) [21] in the dataset. This inclusion is designed to enhance the transparency of the model's decision-making process. In this work, we augmented an open-source dataset [22] with 108,971 Verilog code samples from Hugging Face in the following 3 stages, as shown in Fig. 2-**(I)**.

Stage ❶: Filtering and Syntax Checking. Given the Verilog code samples from [22], we employed scripts to filter out code that exhibited certain undesirable characteristics. Specifically, we excluded code samples based on the following criteria:

1) Incomplete code that lacks either the '*module*' or '*endmodule*';
2) Code comprising only initialisation or assignment statements, with no functional logic;
3) Duplicated code segments.

Then, each remaining piece of code underwent a syntax check through the Icarus Verilog compiler [23]. Code with detectable syntax errors (e.g., those that failed the compilation), were excluded from the following process, preventing their advancement to the following stages where assertions could be triggered. However, these incorrect code samples were not discarded; instead, they are included in the **Verilog-PT** dataset as they offer valuable structural insights into Verilog code. After the syntax check, GPT-4 was employed to generate a design specification (Spec) for all code samples, but only those that failed compilation received further analysis. The analysis aims to provide an explanation for the causes of the syntax errors, contributing to the formation of the **Verilog-PT** dataset, as shown in Fig. 2 dataset (a). The **Verlog-PT** dataset, containing 22,646 entries, which is utilised in section III-A.

Stage ❷: Key Components Generation and Validation. For successfully compiled code, we employed Claude-3.5 to generate random bugs and SVAs. The use of Claude-3.5, as opposed to GPT-4, was intended to leverage the heterogeneous design of these models, thereby helping GPT-4 avoid falling into error traps during self-validation in **Stage ❸**.

To mitigate the impact caused by the hallucinations of LLMs [24]–[26], we utilised EDA tools at this stage to validate the correctness of the generated components. We employed a two-step verification process by integrating the compiler with the verifier, SymbiYosys [27]. Each generated SVA was inserted into the corresponding Verilog code and verified using SymbiYosys to ensure the SVAs' validity. Furthermore, we employed the compiler again to identify and eliminate syntax errors introduced during the random bug generation process. The remaining bugs were injected individually into the Verilog code, and each modified version was then verified with the corresponding SVAs using SymbiYosys to ensure that the bug-SVA pair caused assertion failures and to generate logs.

Approximately 90% of the bugs and SVA pairs associated with assertion failures were selected and processed in **Stage ❸**. The remaining 10% is reserved for testing, as detailed in Section IV-C. This selection process is structured to ensure that the training and test datasets are completely separate, involving the following steps:

1) Organise the buggy Verilog code into categories based on the length of the code, with bins defined as: $(0, 50]$, $(50, 100]$, $(100, 150]$, $(150, 200]$, and $(200, +\infty)$;
2) Enumerate the unique module names within each bin;
3) Uniformly select 90% of the module names and their corresponding buggy SystemVerilog (SV) code, Spec, and logs for inclusion in the training dataset.

Bugs that did not cause assertion failures, potentially due to insufficient SVA coverage, were retained as they represent functional issues in the original Verilog code. These bugs, together with the Spec, Verilog code, and solution, were organised into the **Verilog-Bug** dataset. This dataset, as shown in Fig. 2 dataset (b), contains 36,650 entries and is used in section III-B to enhance the model's understanding of the verification process.

Stage ❸: CoT Generation and Validation. To enhance the transparency of our model, we incorporated CoTs into the training data. This integration involves using GPT-4, where we provided Spec, buggy SV code, logs, and the bug location. The task for GPT-4 is to generate a CoT that articulates its reasoning behind identifying the erroneous code and suggesting correction.

Subsequently, we used a script to validate these CoTs by comparing GPT-4's output with the 'golden solution' obtained from Stage ❷. The 'golden solution' was derived from the initial buggy code and its correct counterpart. If GPT-4 identified an error and proposed correction align with it, we consider the CoT to be correct.

In total, approximately 74.55% of the generated CoTs were identified to be correct. Depending on the correctness of these CoTs, we organised two types of entries into the **SVA-Bug** dataset, which totals 7,842 entries as shown in Fig. 2 dataset (c). For entries where the CoT is correct, the 'Question' section includes the phrase '*step by step*', and the 'Answer' details both the buggy line and its corrected code, along with the CoT. Otherwise, the 'Answer' only includes the buggy line and the correct code. This dataset, designed specifically to equip the model with the ability to solve assertion failures and explain the reasoning process, is utilised in section III-B.

III. TRAINING STRATEGY

To maximise the potential of the dataset generated in Section II, we implemented a tailored training strategy for AssertSolver, as shown in Fig. 2-**(II)**. Our strategy started with a foundational **pretraining (PT)** phase using the base model, Deepseek-Coder-6.7b, on the **Verilog-PT** dataset. This step was to strengthen the model's understanding of Verilog code constructs and design specifications. Following this, we applied **Supervised Fine-Tuning (SFT)** to fine-tune the AssertSolver on the **SVA-Bug** and **Verilog-Bug** datasets. This step was designed to equip the model with the necessary skills for solving assertion failures, and to expand its capability for generalising across related Verilog debugging tasks. To further refine the model performance, we revisited unresolved assertion failure cases from the SFT phase, particularly those with incorrect responses, and employed **Direct Preference Optimisation (DPO)** to enable our model to learn from these error responses to challenging cases.

A. Pretraining

Pretraining is essential for LLMs, especially when preparing them to handle specialised languages such as Verilog and SystemVerilog. It is important to infuse domain-specific knowledge during this stage, which forms a robust base of understanding before any targeted fine-tuning starts. Recent research [14], [28] supports that continual pretraining on domain-specific dataset, including unlabeled Verilog code and syntactically similar language like C/C++ code, can substantially boosts the base model's understanding of hardware description languages (HDLs) and improve its performance in downstream tasks. In line with these insights, we implemented continual pretraining with the **Verilog-PT** dataset, comprising the Verilog code that failed in compilation along with its corresponding specifications and analyses of compilation failures. The base model, Deepseek-Coder-6.7b, has been preliminarily trained on a large programming corpus and is lightweight, making it ideal for this application. This focused pretraining strategy is essential as the timing and concurrency property in HDLs differ significantly from software programming paradigms.

During pretraining, each sample $x^{(i)}$ in the **Verilog-PT** dataset $D_{\text{PT}} = \{x^{(i)}\}_{i=1}^{N}$ is treated as a sequence of tokens. These tokens serve as the basic units in natural language processing tasks, allowing LLMs to leverage reasoning over them for next-token predication, which in turn facilitates text generation [29]–[32]. The sequence for each sample is expressed as $x^{(i)} = (w_1^{(i)}, w_2^{(i)}, \ldots, w_{T^{(i)}}^{(i)})$, where $w_t^{(i)}$ denotes the t-th token and $T^{(i)}$ the total number of tokens in the sequence. The training objective in this stage focuses on minimising the negative log-likelihood loss across the dataset:

$$L_{\text{PT}}(\theta) = -\sum_{i=1}^{N}\sum_{t=1}^{T^{(i)}} \log P(w_t^{(i)} \mid \text{context}_t^{(i)}; \theta)$$

Here $\text{context}_t^{(i)}$ refers to the series of preceding tokens which serve as the basis for predicting $w_t^{(i)}$, and $P(w_t^{(i)} \mid \text{context}_t^{(i)}; \theta)$ represents the probability of predicting t-th token based on this context, as determined by the model's parameter θ. This pretraining lays the groundwork for the subsequent fine-tuning process and equips the model capable of handling tasks within the hardware design domain.

B. Supervised Fine-Tuning (SFT)

Following the unsupervised pretraining phase, where the model primary learning was to predict the next token, Supervised Fine-Tuning (SFT) aims to shift the model's capabilities. This shift moves the focus from mere text continuation to solving specific question-answering challenges presented by assertion failures, which require precise and supervised responses.

To this end, we used the **SVA-Bug** dataset, which includes Spec, buggy SV code and logs. These elements are organised into the model's input x , as shown in the 'Question' section in Fig. 2 dataset (c). The model output the 'Answer' y must include, at a minimum, the bug line snippet and the corresponding correct code. Additionally, if the CoT is verified as correct in **Stage ❸** of section II, it is also included in y, enhancing the answer with detailed reasoning steps, marked by the keyword 'step by step' in x. Furthermore, we integrated the **Verilog-Bug** dataset as an auxiliary task to further enrich the training data with a broader spectrum of Verilog debugging scenarios. The data pairs $\langle x, y \rangle$ from this dataset are structured to provide the model with both the buggy Verilog code in the input 'Question' and the repair plan in the 'Answer', which lists the buggy line and alongside the corrected version, as shown in Fig. 2 dataset (b). This combination of datasets in the SFT process ensures a comprehensive learning experience for the model.

The objective of SFT is to refine the model's ability to predict the next token in $y^{(i)}$ based on the contextual interplay between the input $x^{(i)}$ and the sequence of previously generated tokens $y_{<t}^{(i)}$. This approach is designed to train the model in recognising and replicating the correct relationships between given 'Question' and 'Answer':

$$L_{\text{SFT}}(\theta) = -\sum_{i=1}^{N}\sum_{t=1}^{T_y^{(i)}} \log P(y_t^{(i)} \mid y_{<t}^{(i)}, x^{(i)}; \theta)$$

where $y_t^{(i)}$ denotes the t-th token in the output sequence $y^{(i)}$, and $y_{<t}^{(i)}$ represents the sequence of tokens preceding $y_t^{(i)}$. The likelihood $P(y_t^{(i)} \mid y_{<t}^{(i)}, x^{(i)}; \theta)$ indicates the probability of predicting the token $y_t^{(i)}$ given the $y_{<t}^{(i)}$ and the input context $x^{(i)}$, as governed by the model parameters θ. The SFT allows the model to produce the answer in the expected format and to develop a deeper understanding of the underlying hardware description language.

C. Learning from Error Responses to Challenging Cases

At the SFT stage, AssertSolver is primarily exposed to correct responses, which limits its ability to process or learn from errors - a critical aspect of human learning. As noted in research by [33], [34], effective learning involves not only assimilating correct responses but also reflecting on and learning from mistakes to prevent future errors. Current training paradigms often overlook or discard erroneous data. Inspired by recent research [35]–[37], we propose to equip

TABLE I: Bug types leading to assertion failures and examples

Type	Description	Expected Form	Unexpected Form	Assertion†
Direct	Bug signal appears directly in the assertion.	out <= **in;**	out <= **in + 1;**	assert(out == **in**)
Indirect	Bug signal does not appear directly in the assertion.	**temp <= in;** out <= temp;	**temp <= in + 1;** out <= temp;	assert(out == in)
Var	Incorrect variable name or type.	out = **in**;	out = **input**;	–
Value	Incorrect variable values, constants, or signal bit widths.	out = **4'b1010;**	out = **4'b1110;**	–
Op	Misuse of operators.	out = a **|** b;	out = a **&** b;	–
Cond	Bug in conditional statement (e.g., *if, always*).	**if (valid)** out <= in;	**if (!valid)** out <= in;	–
Non_cond	Bug unrelated to conditional statements.	if (valid) **out <= in;**	if (valid) **out <= input;**	–

† The distinction between *Direct* and *Indirect* type depends on whether the assertion failure is caused by the directly protected signal. Other types of bugs may cause assertion failures but are not necessarily directly reflected in the assertions.

TABLE II: Distribution of SVA-Bug and SVA-Eval across code length intervals and bug types (counts of instances)

Length Interval	(0, 50]	(50, 100]	(100, 150]	(150, 200]	(200, +∞)
SVA-Bug	3400	2444	921	431	646
SVA-Eval	431	260	102	58	64
Bug Type	Direct	Indirect	Var	Value	Op
SVA-Bug	5478	2364	546	5104	2254
SVA-Eval	615	300	47	601	274
Bug Type	Cond	Non_cond	–	–	–
SVA-Bug	1573	6269	–	–	–
SVA-Eval	204	711	–	–	–

AssertSolver with the ability to learn from its errors, thus enhancing its decision-making process.

To facilitate this, we evaluate the SFT model using all samples within the **SVA-Bug** dataset. Each sample, as illustrated in Fig. 2 dataset (c), includes a 'Question' section, which serves as the model's input. For each input, the model generates 20 responses. Correctness is then evaluated by comparing the buggy line suggested by the model with the correct 'Answer' in the dataset. Samples yielding at least one incorrect response among these 20 outputs are categorised as *challenging cases*, representing instances where the model struggles despite prior exposure to correct solutions. In each challenging case, the 'Question' is denoted as x and the correct 'Answer' as p. The incorrect responses to x are denoted as $n[k]$, where $k < 20$.

We abstract them into triples: $D_{\text{DPO}} = \{(x^{(i)}, p^{(i)}, n[k]^{(i)})\}_{i=1}^{N}$, where N denotes the number of *challenging cases*. For this preference dataset D_{DPO}, the objective is to train the model to prioritise generating the correct response $p^{(i)}$ over the error responses $n[k]^{(i)}$ for each $x^{(i)}$. This goal is achieved through the DPO loss function that encourages the model to maximise the probability of generating the correct response $p^{(i)}$, while minimising the probability of generating the error response $n[k]^{(i)}$:

$$L_{\text{DPO}} = -\mathbb{E}_D \left[\log \sigma \left(\beta \log \frac{\pi_\theta(p^{(i)}|x^{(i)})}{\pi_{\text{ref}}(p^{(i)}|x^{(i)})} - \beta \log \frac{\pi_\theta(n[k]^{(i)}|x^{(i)})}{\pi_{\text{ref}}(n[k]^{(i)}|x^{(i)})} \right) \right]$$

In this function, π_θ and π_{ref} represent the current model and the reference model (SFT model, in this context). The terms $\pi_\theta(p^{(i)}|x^{(i)})$ and $\pi_{\text{ref}}(p^{(i)}|x^{(i)})$ denote the likelihood of generating the correct response $p^{(i)}$ given input $x^{(i)}$ under the respective models. Similarly, $\pi_\theta(n[k]^{(i)}|x^{(i)})$ and $\pi_{\text{ref}}(n[k]^{(i)}|x^{(i)})$ correspond to the probabilities of generating the k-th error response $n[k]^{(i)}$ for the same input. The log-ratios of these probabilities quantify the divergence between the two models. The difference between the log-ratios captures the preference of π_θ for generating the correct response $p^{(i)}$ over the error response $n[k]^{(i)}$, guiding the model producing correct answers. The scaling factor β, set to 0.1, controls the weight of the log-ratio terms, and the sigmoid function σ maps the log-ratio values to a probability in the range $[0, 1]$, facilitating smooth learning while ensuring training stability. This approach allows AssertSolver not only to learn from errors but also to improve its response accuracy in challenging cases, leading to more robust decision-making capabilities.

IV. Evaluation

To evaluate the effectiveness of our trained model, we conduct extensive experiments designed to answer four key research questions. This section outlines the dataset property, benchmark and SOTA counterparts, evaluation metrics, and implementation details.

A. Research Questions

The evaluation is structured to investigate the following four research questions:

- **RQ1:** How does the incorporation of learning from error responses influence the performance of the model as measured by metrics of *pass@1* and *pass@5*?
- **RQ2:** How does the effectiveness of our model in solving assertion failures compare to that of other SOTA LLMs or models of similar complexity?
- **RQ3:** How does the model's performance vary when addressing randomly generated bugs versus human-crafted cases?
- **RQ4:** How is the model's performance impacted by design variability, specifically in relation to different bug types and variations in code length?

B. Dataset Property

The challenge of solving assertion failures varies considerably across different bug types and code lengths. Some categories are inherently more complex than others. For example, timing-related bugs that do not directly trigger assertion failures require deep reasoning and analysis, making them more complex for verification engineers. Similarly, longer code lengths may increase the complexity of debugging, as they often involve more intricate logic and a higher potential for subtle errors. Recognising this, we classified the types of bugs leading to assertion failures, as shown in Table I. We also analysed the code lengths and the number of instances falling into each identified bug category within our training and testing datasets, as illustrated in Table II. This classification is essential for evaluating the performance of LLMs across different categories, as it helps determine whether LLMs can effectively address bug types consistent with our expectations.

TABLE III: Model performance as pass@k (grey shading: the best performance across models).

Metric	Base Model	SFT Model	AssertSolver
Pass@1	4.35%	84.66%	88.54%
Pass@5	15.62%	91.64%	90.00%

C. New Open-Source Benchmark and SOTA Counterparts

Given the lack of open-source benchmarks for evaluating tasks related to solving assertion failures, we have developed a benchmark, **SVA-Eval**, to address this gap. This benchmark consists of 877 samples generated by LLMs (**SVA-Eval-Machine**), as described in Section II, and 38 manually curated samples (**SVA-Eval-Human**) derived from the RTLLM dataset [38], ensuring both the scale of the dataset and the inclusion of real-world scenarios. Each sample in **SVA-Eval** includes the Spec, buggy SV code, logs, and correct solutions, providing a comprehensive resource for evaluation. The dataset is publicly available on **https://github.com/SEU-ACAL/reproduce-AssertSolver-DAC-25** to support further research in this domain.

For the comparison between AssertSolver and the current SOTA LLMs, we have chosen several commercially available closed-source models as benchmarks. These include Claude-3.5, GPT-4 [39], and OpenAI's latest o1-preview, all of which have demonstrated exceptional capabilities in RTL generation and debugging tasks. Furthermore, we have extended our comparative analysis to include open-source models such as CodeLlama-6.7b [40], Llama-3.1-8b, and our base model, Deepseek-Coder-6.7b [20]. This inclusive approach aims to provide a comprehensive overview of AssertSolver's performance across a spectrum of platforms and development environments.

D. Evaluation Metrics

To evaluate the performance of our LLM in solving assertion failures, we employ the ***pass@k*** metric, widely used in the evaluation performance of hardware designs [22], [41], [42]. This metric quantifies the effectiveness of LLMs by measuring their ability to generate correct solutions for each buggy SV code that causes assertion failures. For each instance, the model is provided with the buggy SV code alongside its corresponding specifications and logs, from which it generates n possible solutions. We then assess these solutions, deeming c of them effective if they successfully solve the assertion failure. This approach offers an unbiased estimate of the likelihood that at least one of the top k selections will address the problem, as shown by the following equation:

$$pass@k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

In this study, we set $n = 20$ and $k = \{1, 5\}$.

- ***Pass@1*** assesses the model's accuracy and consistency by requiring the correct solution to be produced on the first attempt. An improvement in *pass@1* suggests that the model is becoming more adept in identifying and responding accurately to the bugs, thus likely improving its precision for such tasks.
- ***Pass@5*** evaluates whether model can provide at least one correct solution within five attempts. An increase in *pass@5* indicates that there is an enhancement in the model's ability to generate diverse solutions, reflecting its flexibility in problem-solving.

E. Implementation Details

Training. We fine-tuned the Deepseek-Coder-6.7b model using 8 A800-80G GPUs and accelerated the process with DeepSpeed ZeRO-3 [43]. We opted for full-parameter fine-tuning to achieve optimal performance and set an initial learning rate of 10^{-4} for pretraining and SFT, incorporating a warm-up phase during the first 10% of training steps. For DPO, a lower learning rate of 10^{-6} was used, as it focuses on learning the difference between correct and incorrect answers, rather than directly optimizing for explicit answers.

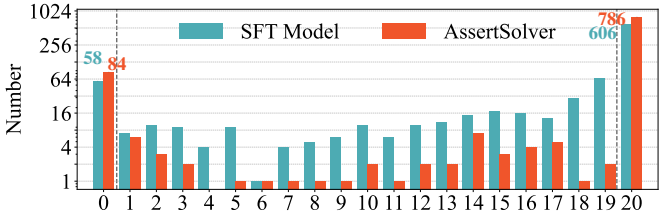


Fig. 3: Histogram of correct answers across 20 responses. (x-axis: c (number of correct solutions in 20 responses))

Inference. We combined the Spec, buggy SV code, and logs from the benchmark, requiring LLMs to return responses in a JSON format with a candidate buggy line, suggested fix, and CoT (Fig. 2-**(III)**). In experiments, we found open-source models often deviated from the prompt format, so we iteratively refined prompts until $n = 20$ JSON responses were generated per assertion failure case for *pass@k* evaluation. The temperature was 0.2 for consistent yet diverse outputs, except for o1-preview, which has a fixed temperature that cannot be adjusted through the application programming interface.

V. Results and Analysis

RQ1: To answer this question, we compared three models: the base model, the SFT model, and AssertSolver, using the **SVA-Eval** dataset. As shown in in Table III, the base model shown the *pass@1* and *pass@5* rates below 5% and 16%, respectively. In contrast, both the SFT model and the AssertSolver, which underwent pre-training and fine-tuning process with our augmented dataset, consistently achieved more than 80% *pass@1* and *pass@5*. This presents a significant performance improvement, with over a 16-fold improvement in *pass@1* and a 5-fold increase in *pass@5*.

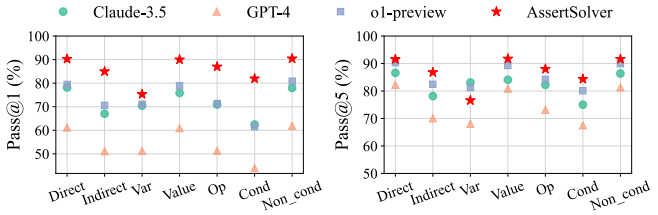
In evaluating the performance between the SFT model and the AssertSolver, a clear differences is observed. The AssertSolver, which was further trained on errors from challenging cases, demonstrated an improvement in *pass@1* performance, increasing from 84.66% to 88.54%. However, this was accompanied by a slight decline in *pass@5* performance, from 91.64% to 90%, compared to the SFT model. As outlined in Section IV evaluation metric, *pass@1* and *pass@5* reflect distinct characteristics of the underlying model. This observation suggests that although the model's precision improves with additional training on errors from challenging cases, its ability to generate a diverse range of solutions decreases.

Further analysis examined the performance of 915 test cases in the **SVA-Eval** dataset. In each cases, it generated $n = 20$ possible solutions. We evaluated these solutions, deeming c of them effective. These test cases were then categorised into 21 distinct outcomes ranging from '$c = 0$' (indicating no effective solutions) to '$c = 20$' (where all solutions were effective). Intermediate values suggested varying levels of success and correlated with increased uncertainty as shown in Fig.3. In the figure, the AssertSolver generally outperforms the SFT model in deterministic scenarios (i.e., '$c = 0$' and '$c = 20$') but fell short in non-deterministic ranges. This finding, as supported by Table III, indicates that while adding challenging cases enhances the model's precision, it may limit the diversity of the solutions.

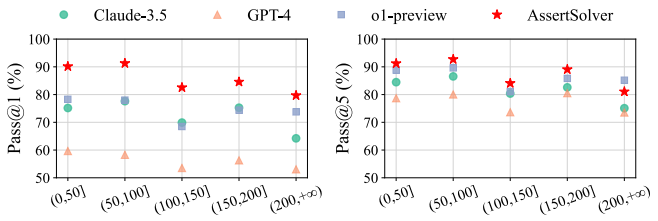
RQ2: For this research question, we compared the performance of AssertSolver with leading commercial and open-source LLMs, including the recently released o1-preview from OpenAI, focusing

TABLE IV: Performance comparison between AssertSolver and other LLMs (grey shading: the best performance across models).

Model	SVA-Eval-Machine		SVA-Eval-Human		SVA-Eval	
	pass@1 (%)	*pass@5 (%)*	*pass@1 (%)*	*pass@5 (%)*	*pass@1 (%)*	*pass@5 (%)*
Claude-3.5	74.86	84.10	66.58	77.48	74.52	83.83
GPT-4	58.04	78.45	54.74	74.01	57.90	78.27
o1-preview	76.96	87.73	67.50	87.94	76.57	87.74
Deepseek-coder-6.7b	4.41	15.85	2.89	10.27	4.35	15.62
CodeLlama-7b	5.95	17.06	4.47	12.85	5.89	16.89
Llama-3.1-8b	20.18	32.41	14.08	24.48	19.92	32.08
AssertSolver	89.04	90.38	76.97	81.35	88.54	90.00



(a) Performance comparison across different bug types



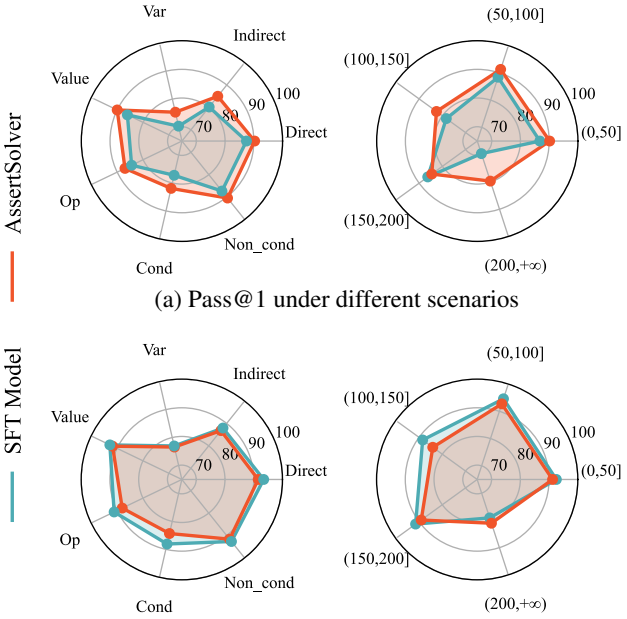
(b) Performance comparison across different code length intervals

Fig. 4: Comprehensive comparison with closed-source LLMs.

on *pass@1* and *pass@5* metrics within the **SVA-Eval** dataset. These comparisons are detailed in Table IV. AssertSolver outperforms all other models, achieving more than 80% in the *pass@1* metric (reaching 88.54%) and 90% in *pass@5*, while the second-best model, the o1-preview, scored 76.57% and 87.74% respectively. Further analysis, divided by the methods used to generate bugs, indicated that AssertSolver consistently performs best in all categories, except for *pass@5* in the **SVA-Eval-Human**. Considering the results from RQ1, where the AssertSolver demonstrated a preference for precision over diversity—thereby trading off performance in *pass@1* for *pass@5*—these outcomes align with expectations.

RQ3: As shown in Table IV, we observed that the performance for both *pass@1* and *pass@5* in **SVA-Eval-Human** consistently underperforms compared to **SVA-Eval-Machine**, with the exception of the *pass@5* metric on the o1-preview model. Across all tested models, there was an average relative decline of approximately 19% in *pass@1* and 15% in *pass@5*, which were calculated by averaging the ratio of the *pass@1* and *pass@5* rates between the **SVA-Eval-Machine** and **SVA-Eval-Human** datasets. This observation suggests there may be a systemic difference between machine and human-generated bugs, but requiring further investigation to confirm.

RQ4: To answer this research question, we evaluated AssertSolver's performance across different bug types and code lengths against closed-source LLMs, as shown in Fig. 4. AssertSolver consistently surpasses the performance of compared LLMs in *pass@1* across all tested scenarios and outperforms in *pass@5* in 10 out of 12 (83%) scenarios. Despite slight underperformance in the '*Var*' and in code length exceeding 200, AssertSolver demonstrates better results in



(a) Pass@1 under different scenarios

(b) Pass@5 under different scenarios

Fig. 5: The performance of STF Model and AssertSolver in various bugs types and code length intervals.

all remaining cases. Particularly, for shorter code (under 100 lines) and the bugs classified as '*Direct*', '*Value*', and '*Non_cond*', both *pass@1* and *pass@5* reached over 90%, showcasing AssertSolver's effectiveness across various design scenarios.

Further analysis, as shown in Fig. 5, highlights how learning from error responses to challenging cases improves the *pass@1* across nearly all scenarios, particularly for code lengths exceeding 200 lines, with the exception of code between 150-200 lines. While there is a slight decrease in *pass@5* in these instances, this drop is not unexpected. AssertSolver prioritises precision over diversity, leading to performance that inherently favours *pass@1* results.

VI. Conclusion

We presented AssertSolver, the first open-source LLM designed to address assertion failures in RTL design. To overcome the challenge of data underrepresentation in training datasets, we implemented a data augmentation method that automatically enriches the training data with diverse assertion failure scenarios. Also, we introduced a novel training strategy that enables the model to learn not only from the augmented data but also from errors in challenging cases, thereby enhancing its capability to solve assertion failures effectively.

Our experimental results show that AssertSolver achieves *pass@1* and *pass@5* rates of 88.54% and 90.00%, respectively, on a comprehensive test set of over 900 instances. This performance surpasses the recently released o1-preview by 11.97% and 2.26% in the *pass@1* and *pass@5* metrics, respectively. Furthermore, by learning errors from challenging cases, AssertSolver exhibits increased *pass@1* rate. This significant performance underscores the model's suitability for applications that require high reliability and consistency, such as solving assertion failures in hardware design.

Moreover, we have made AssertSolver publicly available for early adoption and released an openly accessible evaluation benchmark for solving assertion failures, featuring various bug types generated by both machine and domain experts. Our work demonstrates that domain-specific fine-tuning of LLMs, combined with effective data augmentation and training strategies, can significantly advance the automation of solving assertion failures in hardware design.

REFERENCES

[1] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[2] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[3] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[4] R. K. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: Leveraging formal for debugging," in *Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 648–651.

[5] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Elsevier, 2023.

[6] K. Maddala, B. Mali, and C. Karfa, "Laag-rv: Llm assisted assertion generation for rtl design verification," in *2024 IEEE 8th International Test Conference India (ITC India)*. IEEE, 2024, pp. 1–6.

[7] S. S. Miftah, A. Srivastava, H. Kim, and K. Basu, "Assert-o: Context-based assertion optimization using llms," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, 2024, pp. 233–239.

[8] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms," *arXiv preprint arXiv:2402.00386*, 2024.

[9] V. Pulavarthi, D. Nandal, S. Dan, and D. Pal, "Assertionbench: A benchmark to evaluate large-language models for assertion generation," *arXiv preprint arXiv:2406.18627*, 2024.

[10] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using llms to facilitate formal verification of rtl," *arXiv e-prints*, pp. arXiv–2309, 2023.

[11] C. Sun, C. Hahn, and C. Trippel, "Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions," in *First International Workshop on Deep Learning-aided Verification*, 2023.

[12] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[13] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[14] N. Wang, B. Yao, J. Zhou, X. Wang, Z. Jiang, and N. Guan, "Large language model for verilog generation with golden code feedback," *arXiv preprint arXiv:2407.18271*, 2024.

[15] C.-T. Ho, H. Ren, and B. Khailany, "Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool," *arXiv preprint arXiv:2408.08927*, 2024.

[16] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[17] K. Xu, J. Sun, Y. Hu, X. Fang, W. Shan, X. Wang, and Z. Jiang, "Meic: Re-thinking rtl debug automation using llms," *arXiv preprint arXiv:2405.06840*, 2024.

[18] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

[19] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[20] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[22] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[23] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.

[24] G. Perković, A. Drobnjak, and I. Botički, "Hallucinations in llms: Understanding and addressing challenges," in *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*. IEEE, 2024, pp. 2084–2088.

[25] G. P. Reddy, Y. P. Kumar, and K. P. Prakash, "Hallucinations in large language models (llms)," in *2024 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, 2024, pp. 1–6.

[26] S. Tonmoy, S. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das, "A comprehensive survey of hallucination mitigation techniques in large language models," *arXiv preprint arXiv:2401.01313*, 2024.

[27] C. Wolf *et al.*, "Symbiyosys," *URL: https://symbiyosys. readthedocs. io/.[Cited on page 6.]*, 2022.

[28] B. Yao, N. Wang, J. Zhou, X. Wang, H. Gao, Z. Jiang, and N. Guan, "Location is key: Leveraging large language model for functional bug localization in verilog," *arXiv preprint arXiv:2409.15186*, 2024.

[29] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot *et al.*, "Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp," *arXiv preprint arXiv:2112.10508*, 2021.

[30] M. Qi, Y. Huang, Y. Yao, M. Wang, B. Gu, and N. Sundaresan, "Is next token prediction sufficient for gpt? exploration on code logic comprehension," *arXiv preprint arXiv:2404.08885*, 2024.

[31] R. Sennrich, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[32] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, "Byte pair encoding: A text compression scheme that accelerates pattern matching," 1999.

[33] N. Mercer, "Talk and the development of reasoning and understanding," *Human development*, vol. 51, no. 1, pp. 90–100, 2008.

[34] T. Reich, A. Kaju, and S. J. Maglio, "How to overcome algorithm aversion: Learning from mistakes," *Journal of Consumer Psychology*, vol. 33, no. 2, pp. 285–302, 2023.

[35] Y. Tong, D. Li, S. Wang, Y. Wang, F. Teng, and J. Shang, "Can llms learn from previous mistakes? investigating llms' errors to boost for reasoning," *arXiv preprint arXiv:2403.20046*, 2024.

[36] K. Chen, C. Wang, K. Yang, J. Han, L. Hong, F. Mi, H. Xu, Z. Liu, W. Huang, Z. Li *et al.*, "Gaining wisdom from setbacks: Aligning large language models via mistake analysis," *arXiv preprint arXiv:2310.10477*, 2023.

[37] S. An, Z. Ma, Z. Lin, N. Zheng, J.-G. Lou, and W. Chen, "Learning from mistakes makes llm better reasoner," *arXiv preprint arXiv:2310.20689*, 2023.

[38] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[39] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[40] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[41] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards llm-powered verilog rtl assistant: Self-verification and self-correction," *arXiv preprint arXiv:2406.00115*, 2024.

[42] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[43] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.