OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection

Fan Cui
School of Integrated Circuits, Peking University
pku_cf@stu.pku.edu.cn

Chenyang Yin
Peking University
ycy@stu.pku.edu.cn

Kexing Zhou
School of Integrated Circuits, Peking University
zhoukexing@pku.edu.cn

Youwei Xiao
School of Integrated Circuits, Peking University
shallwe@pku.edu.cn

Guangyu Sun
School of Integrated Circuits, Peking University
gsun@pku.edu.cn

Qiang Xu
The Chinese University of Hong Kong
qxu@cse.cuhk.edu.hk

Qipeng Guo
Shanghai AI Laboratory
qpguo16@fudan.edu.cn

Demin Song
Shanghai AI Laboratory
songdemin@pjlab.org.cn

Dahua Lin
Shanghai AI Laboratory
lindahua@pjlab.org.cn

Xingcheng Zhang
Shanghai AI Laboratory
zhangxingcheng@pjlab.org.cn

Yun (Eric) Liang*
School of Integrated Circuits, Peking University
ericlyun@pku.edu.cn

ABSTRACT

Recent studies have demonstrated the significant potential of Large Language Models (LLMs) in generating Register Transfer Level (RTL) code, with notable advancements showcased by commercial models such as GPT-4 and Claude3-Opus. However, these proprietary LLMs often raise concerns regarding privacy and security. While open-source LLMs offer solutions to these concerns, they typically underperform commercial models in RTL code generation tasks, primarily due to the scarcity of high-quality open-source RTL datasets. To address this challenge, we introduce OriGen, a fully open-source framework that incorporates self-reflection capabilities and a novel dataset augmentation methodology for generating high-quality, large-scale RTL code. Our approach employs a code-to-code augmentation technique to enhance the quality of open-source RTL code datasets. Furthermore, OriGen can rectify syntactic errors through a self-reflection process that leverages compiler feedback.

Experimental results demonstrate that OriGen significantly outperforms other open-source alternatives in RTL code generation. It surpasses the previous best-performing open-source LLM by 12.8% and even exceeds GPT-4 Turbo in the pass@1 metric on the VerilogEval-Human benchmark. Moreover, OriGen exhibits superior capabilities in self-reflection and error correction, outperforming GPT-4 by 19.9% on a benchmark designed to evaluate self-reflection capabilities.

OriGen is open source at GitHub(https://github.com/pku-liang/OriGen)

*Corresponding author

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM 979-8-4007-1077-3/24/10…
https://doi.org/10.1145/3676536.3676830

1 INTRODUCTION

Recent advancements in large language models (LLMs) have demonstrated exceptional capabilities in natural language comprehension and generation [18]. Studies have shown that LLMs exhibit considerable proficiency in code generation tasks [28]. Commercial LLMs, such as GPT-4 [1] and Claude3-Opus [2], have showcased their ability to generate high-quality software code for common programming languages like C++ and Python, significantly enhancing coding productivity. Moreover, LLMs have demonstrated impressive capabilities in hardware code generation, particularly for Register Transfer Level (RTL) code. The generation of RTL code from natural language instructions presents an innovative approach to enhance hardware development productivity. This method has the potential to revolutionize existing Hardware Description Language (HDL) coding workflows by alleviating the burdensome task of HDL coding for designers. Similar to High-Level Synthesis (HLS), LLMs can streamline the design process by enabling code generation from high-level specifications, thus simplifying complex hardware design tasks [5, 8].

While commercial LLMs have demonstrated proficiency in generating RTL code, they often raise concerns regarding privacy and security. These issues are particularly critical in the domain of hardware design, as RTL code frequently contains valuable intellectual property. Furthermore, the closed-source nature of these LLMs restricts researchers from conducting in-depth analyses and customizations, impeding further fine-tuning of the models in specific domains, such as hardware design. In contrast, open-source models provide enhanced privacy and security while facilitating further improvement and customization. Open-source models, such as StarCoder2 [22], DeepSeek-Coder [12], and CodeQwen-V1.5 [3], have demonstrated promising results in code generation for prevalent programming languages like Python and C/C++. However,

their performance in RTL code generation still falls behind GPT-3.5. This performance gap underscores the pressing need to develop a specialized open-source LLM tailored for RTL code generation.

Nonetheless, there is a notable scarcity of high-quality, large-scale RTL code datasets compared to those available for other popular programming languages. This shortage poses a significant challenge in achieving satisfactory results for RTL generation tasks. To address this issue, several studies have focused on collecting open-source code snippets [10, 26, 29, 30]. However, these open-source datasets may include low-quality code that can impair model performance [20]. Alternative approaches have been developed to synthesize RTL code using LLMs [6, 20, 21]. VerilogEval [20] generates datasets by adding descriptions to open-source Verilog code but suffers from overall low quality. RTLCoder [21] selects hardware-related keywords as seeds to generate natural language instructions, which are then converted into corresponding RTL code using GPT-3.5. While this dataset meets quality standards, its scale is limited due to the restriction of generated data to selected keywords.

Self-reflection is another critical factor in RTL code generation. In this context, self-reflection refers to the model's ability to assess its generated code based on feedback from hardware compilers and simulators. This process involves identifying error causes and refining results accordingly, mirroring the hardware design process where RTL code undergoes rigorous evaluation to ensure correctness and adherence to design specifications.

Existing open-source LLMs primarily focus on code generation, often neglecting the interactive process of compilation and simulation integral to hardware design methodologies. To address these limitations, several studies have integrated self-reflection methodologies into their frameworks [31, 32, 34]. However, both RTLFixer [32] and AutoChip [31] rely on commercial LLMs for self-reflection, raising potential privacy and security concerns as previously discussed. Open-source LLMs generally exhibit weaker self-reflection capabilities compared to their commercial counterparts, primarily due to limited training data. Consequently, there is a pressing need to develop methodologies for constructing datasets specifically designed to enhance the self-reflection abilities of open-source LLMs in the context of RTL code generation.

In this paper, we introduce OriGen, a fully open-source framework for RTL generation featuring self-reflection capabilities and a dataset augmentation methodology. We propose a novel code-to-code augmentation technique to improve the quality of open-source RTL code datasets, which are typically large-scale but of limited quality due to the lack of rigorous review and validation processes.

Our approach involves extracting high-level code descriptions from open-source RTL code and refining the code based on these descriptions using the Claude3-Haiku model as a teacher. Commercial LLMs, trained on vast amounts of high-quality data, possess strong programming language understanding and generation capabilities. By leveraging these models as teachers, we can distill their knowledge to refine open-source RTL code, potentially achieving performance superior to the teacher model through rigorous review and validation.

To evaluate the LLM's self-reflection capability in RTL code, we construct a benchmark named VerilogFixEval, comprising 221 cases of failed compilation from VerilogEval [20]. OriGen can correct syntactic errors by leveraging a self-reflection process based on compiler feedback. When generated code fails to pass compiler verification, the model initiates the self-reflection process, accepting erroneous code and compiler error messages to fix the RTL code, thereby enhancing its correctness and reliability. The model's self-reflection ability is facilitated by a carefully constructed error-correction dataset, which includes natural language instructions, erroneous code, compiler error messages, and corrected code generated by Claude3-Haiku. This dataset serves as a valuable resource to bridge the gap in self-reflection capabilities between open-source and commercial closed-source LLMs. By training on this dataset, OriGen can acquire self-reflection abilities comparable to those of commercial LLMs, enabling more reliable and accurate RTL code generation.

Our contributions are summarized as follows:

- We introduce OriGen, which significantly outperforms other alternatives designed for RTL code generation and achieves performance comparable to the latest version of GPT-4 Turbo. *OriGen is open source at https://github.com/pku-liang/OriGen.*
- We propose a novel code-to-code augmentation methodology for generating high-quality, large-scale RTL code datasets, enhancing the model's training data.
- We introduce a self-reflection mechanism that enables OriGen to autonomously fix syntactic errors by leveraging compiler feedback, improving its code generation accuracy. Furthermore, we construct a dataset to improve the model's self-reflection capability based on compiler error messages and develop a benchmark to evaluate this capability.

Experimental results demonstrate that OriGen significantly outperforms other open-source alternatives in RTL code generation [19–21, 24, 26]. It surpasses the previous best-performing open-source LLM by 12.8% and even exceeds GPT-4 Turbo in the pass@1 metric on the VerilogEval-Human benchmark. Furthermore, OriGen exhibits superior capabilities in self-reflection and error correction, outperforming GPT-4 by 19.9% on the benchmark designed to evaluate self-reflection capabilities.

2 BACKGROUND AND RELATED WORK

2.1 LLMs for Verilog Generation

Large Language Models (LLMs) have emerged as powerful tools for code generation across various programming languages. Trained on vast amounts of code and natural language data, these models can learn the statistical patterns and relationships within the training data, enabling them to generate code that adheres to the syntax and style of the target programming language. Recently, researchers in the field of hardware design have shown a growing interest in leveraging LLMs for generating RTL code. RTL is a crucial abstraction level in hardware design, describing the flow of data and control signals between registers and combinational logic. By fine-tuning LLMs on RTL code datasets, models are capable of generating RTL code, potentially accelerating the hardware design process.

Among the pioneering endeavors, DAVE [25] represents an initial exploration into the generation of Verilog code from natural language instructions through the fine-tuning of the GPT-2 model. Subsequently, recognizing the notable scarcity of large-scale Verilog datasets relative to more common programming languages, Verigen [30] collects a comprehensive Verilog dataset from GitHub's

open-source repositories and various textbooks. However, the performance of its optimally fine-tuned model based on the collected dataset remains inferior to that of GPT-3.5 [23]. This can be attributed to the varying quality of the collected open-source code, which was not filtered and processed.

Both VerilogEval [20] and RTLCoder [21] acknowledge the importance of dataset quality and adopt data synthesis methods using commercial LLMs to improve the quality of their datasets. VerilogEval proposes a synthesis approach that utilizes GPT-3.5 to add descriptions to each code snippet in the filtered VeriGen [30] dataset. This approach allows the LLM trained on the dataset to learn not only the syntax of Verilog but also to understand the correspondence between Verilog code and natural language. However, due to the average low quality of code in the dataset, VerilogEval still falls short of GPT-3.5. RTLCoder [21] adopts a synthesis approach to generate RTL code from natural language specifications. It selects one or two keywords from a prepared list of hundreds of digital design keywords and utilizes GPT-3.5 to generate RTL design instructions based on these keywords. Subsequently, GPT-3.5 is employed again to generate corresponding RTL code from these instructions. However, the reliance on a limited set of keywords poses challenges for further expansion of this dataset, as it may restrict the scale and complexity of the generated RTL code. In contrast, the dataset synthesized through the code-to-code approach is not only of high quality, as demonstrated by the experimental results, but also extensive in scale, encompassing a diverse range of RTL code samples. This approach overcomes the limitations of above synthesis and enables the generation of a comprehensive and high-quality dataset that can effectively support the training of LLMs for RTL code generation.

Apart from fine-tuning open-source models, several studies [4, 7, 11] investigate the direct application of commercial LLMs for generating RTL code. Chip-Chat [4] employs a conversational interface to design and verify an 8-bit accumulator-based microprocessor using GPT-4. ChipGPT [7] introduces a four-stage, zero-code logic design framework that leverages GPT model.

2.2 Reflection for Verilog Generation

Existing open-source LLMs designed for RTL code generation have mainly focused on the generative aspects, neglecting the crucial stages of verification and reflection that are integral to hardware design methodologies. The effectiveness of self-reflection and receptivity to feedback in addressing real-world issues has been demonstrated by SWE-agent [36], which successfully resolved problems in GitHub's repositories. This highlights the potential benefits of incorporating self-reflection mechanisms into LLMs for RTL code generation, as it could enable the models to learn from feedback and iteratively improve the generated code, aligning with the rigorous verification processes inherent to hardware design methodologies.

Recognizing the limitations of previous research on LLMs for RTL generation, AutoChip [31] and RTLFixer [32] decide to introduce self-reflection into the their framework. Specifically, RTLFixer employs GPT to generate and reflect on code. It includes approximately 80 erroneous code snippets that fail compilation, each paired with guidance. During the RTL code generation process, if the generated code fails to compile, RTLFixer [32] queries the database for the guidance most closely related to the error. However, the generalization of this approach is limited due to the constraint of the guidance database. In contrast, AutoChip [31] directly provides the compiler error messages to the LLM rather than querying a database for guidance.

The error-correction dataset we construct is diverse and not limited to a specific set of errors, enabling the models to handle a broader range of issues encountered during RTL code generation. By leveraging this dataset, the models can improve their capabilities of self-reflection.

2.3 Hardware Programming

Hardware description languages such as Verilog or VHDL are widely adopted for hardware design. As an alternative, High-Level Synthesis (HLS) is a crucial methodology for hardware design [5, 8]. Similar to using LLMs to write RTL code, HLS enables designers to specify hardware functionality using high-level programming languages like C, C++, or SystemC. These specifications are then automatically converted into hardware description languages such as VHDL or Verilog [9, 13, 17, 35]. By automating this translation process, HLS simplifies the design of complex hardware systems, reducing the time and effort required to manage intricate low-level coding, allowing designers to optimize and manipulate designs more efficiently [15, 16].

3 METHODOLOGY

In this section, we provide comprehensive details of OriGen, a powerful framework designed to enhance Verilog code generation.

3.1 Overview

The overview of the code-to-code augmentation methodology and OriGen's generation and self-reflection framework are illustrated in Figure 1 and Figure 2, respectively.

As shown in Figure 1, our code-to-code augmentation process generates two datasets: an enhanced code dataset and an error-correction dataset. The enhanced code dataset pairs Verilog code with natural language descriptions, enabling the LLM to learn the mapping between RTL code structures and high-level semantics. The error-correction dataset contains instructions erroneous, code samples and their corrections, enhancing the LLM's self-reflection and error correction capabilities. This dataset is constructed by collecting cases where generated code fails evaluation, documenting the specific errors and compiler feedback, and providing corrected versions of the code that pass evaluation.

As shown in Figure 2, in the code generation and error correction stage, OriGen comprises a base LLM and two trained LoRA (Low-Rank Adaptation) models: Gen LoRA and Fix LoRA. Gen LoRA is fine-tuned on the enhanced code dataset. It specializes in interpreting natural language instructions and generating initial RTL code that aligns with the given specifications. Fix LoRA is fine-tuned on the error-correction dataset and built upon Gen LoRA. This component is responsible for analyzing compiler error messages, identifying code issues, and proposing and implementing fixes to erroneous code. OriGen receives natural language instructions as input, and Gen LoRA generates initial RTL code based on these

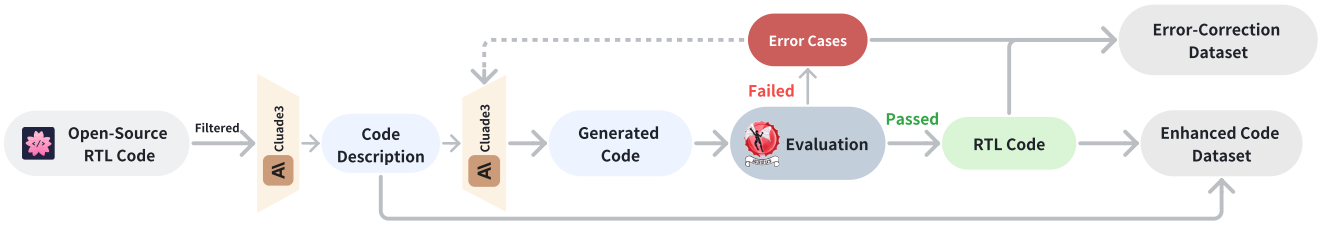

Figure 1: Code-to-Code Augmentation

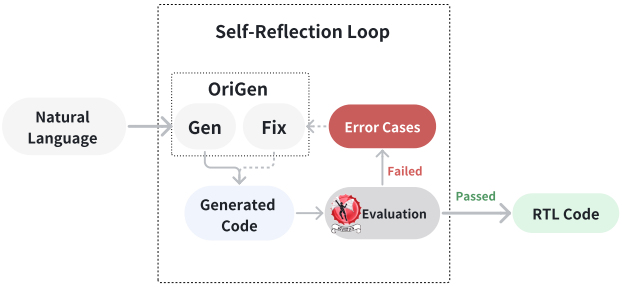

Figure 2: Generation and Self-Reflection

instructions. The generated code then undergoes compilation evaluation. If the code fails to pass evaluation, it enters the self-reflection loop. In this loop, Fix LoRA analyzes the error messages and the erroneous code, proposes and implements fixes to address the identified issues, and the corrected code is re-evaluated. This process iterates until the code passes verification or reaches a predefined maximum number of iterations.

3.2 Code-to-Code Augmentation

The code-to-code augmentation process aims to transfer advanced RTL code generation and correction capabilities from commercial LLMs to our model. To achieve this, a carefully filtered collection of comprehensive open-source RTL code samples is utilized as a foundation.

To extract a valuable dataset from open-source RTL code samples [22, 30], a rigorous filtration process is applied. Initially, due to the constraints imposed by the model's context window and the challenges associated with incomplete descriptions for longer code snippets, samples exceeding 300 lines or 1536 tokens are excluded.

Additionally, samples with an average of more than 30 tokens (approximately 90 characters) per line are considered non-standard and are consequently eliminated, ensuring that only concise and standard code samples are retained. Subsequently, to ensure the meaningfulness and substantive content of the code snippets, a keyword-based filtration approach is employed. Each sample must contain both the `module` and `endmodule` keywords, alongside at least one occurrence of keywords related to procedural blocks, `always` (inclusive of variants like `always_comb`, `always_ff`, `always_latch`, *etc.*) or `assign`. This criterion guarantees that the selected snippets are representative of functional and logical hardware designs. Lastly, all comments within the code samples are removed. This step is crucial to prevent extraneous information from influencing the generation of accurate and relevant specifications.

Following the implementation of the aforementioned filtering procedures, the closed-source LLM Claude3-Haiku is utilized to generate detailed descriptions corresponding to the filtered code samples as shown in Figure 1. The prompt we use is illustrated in Figure 3. These descriptions are then used to regenerate RTL code, which replaces the original code in the dataset. During the regeneration process, the generated code undergoes verification using the open-source compiler Icarus Verilog (Iverilog) [33]. If the code fails to compile, the compiler's error messages and the erroneous code are fed back into the LLM for regeneration to fix error. Simultaneously, code samples that fail compilation are utilized

Description Prompt
Explain the high-level functionality of the Verilog module.
Task:
Please analyze it and provide a detailed description of its signals and functionality. Use as many high-level concepts that are directly applicable to describe the code, but do not include extraneous details that aren't immediately applicable. Speak concisely as if this was a specification for a circuit designer to implement. You should only reply with descriptive natural language and not use any code.
Code:
{code}
Response:

Figure 3: Prompt for Generating Code Descriptions

as foundational elements for the error-correction dataset, as detailed in Section 3.3. This iterative process is repeated until the code successfully compiles or until the maximum number of iterations is reached, further enhancing the dataset's quality. This method facilitates the transfer of knowledge from the commercial LLM to our model, thereby improving its capability to generate and correct RTL code efficiently.

A full example of dataset augmentation is shown in Figure 4. The process begins with the original Verilog code, which defines a flip-flop with synchronous reset and enable signals. The LLM generates a description of this code, explaining its functionality, specifically, how the output q is updated based on the input d on the rising edge of the clock clk. Using this description, the LLM produces an augmented version of the code. The enhanced code presents a more concise and organized structure while maintaining the same functionality, demonstrating the effectiveness of LLMs in improving RTL code readability and quality.

Original Code:

```
module flip_flop(clk,d,q, rst, ena);
input  wire clk, rst, ena;
input  wire d;
output reg  q;
always @(posedge clk) begin
    if(rst) begin
        q <= 0;
    end
    else begin
        if(ena) begin
            q <= d;
        end
    end
end
endmodule
```

Description (Generated by LLM):

This Verilog module is a flip-flop with synchronous reset and enable signals. It updates its output 'q' based on the input 'd' on the rising edge of the clock signal 'clk'. If the reset signal 'rst' is asserted, the output 'q' is set to 0. Otherwise, if the enable signal 'ena' is asserted, the output 'q' is updated with the value of the input 'd'.

Augmented Code:

```
module flip_flop (
    input  wire clk,
    input  wire d,
    input  wire rst,
    input  wire ena,
    output reg  q
);
    always @(posedge clk) begin
        if (rst) begin
            q <= 1'b0;
        end
        else if (ena) begin
            q <= d;
        end
    end
endmodule
```

Figure 4: Example of Dataset Augmentation

3.3 Error-Correction Dataset

Although OriGen, after being trained on the enhanced code dataset, demonstrates performance comparable to that of advanced closed-source LLMs in RTL code generation, it exhibits weaker self-reflection capabilities compared to commercial LLMs like other open-source LLMs.

Given the observed performance gap, we decide to adopt the similar method to acquire stronger error understanding and self-reflection capabilities from the closed-source LLM. During the code-to-code augmentation process discussed in Section 3.2, the closed-source LLM generates a diverse set of RTL code samples that fail to compile, which are then corrected in the subsequent self-reflection process. We select the code samples generated in the code-to-code augmentation process that pass compilation after correction, along with the corresponding samples that failed to compile. The natural language instruction descriptions and compiler error messages associated with these samples serve as data sources for constructing the error-correction dataset. This approach enables the model to be exposed to a broad spectrum of error types and their respective corrections, facilitating the development of robust error understanding and self-reflection capabilities.

Subsequently, these data samples are further filtered to ensure that the model learns to make modifications within the module's body rather than altering the module's declaration. Specifically, during the self-reflection and rectification process, the model may attempt to correct syntactic errors by modifying declarations to pass the compilation check though it is explicitly prohibited in the prompt. To prevent this, we remove samples with such modifications from the collected code. This approach is crucial for preserving the overall structure and interface of the module, which is essential for maintaining compatibility with other modules in the design. Meanwhile, by restricting modifications to the module's body, we encourage the model to focus on identifying and correcting errors within the actual implementation logic. Through the above methods, we have successfully constructed a high-quality error-correction dataset. This dataset comprises a large number of effectively repaired code samples generated by the Claude3-Haiku model, along with the corresponding erroneous code, compiler error messages, and natural language instructions.

To verify the effectiveness of the error-correction dataset, we introduce a benchmark named VerilogFixEval detailed in Section 3.5.

3.4 Code Generation and Fix

As shown in Figure 2, in the code generation and error correction stage, OriGen comprises a base LLM and two trained LoRA models: Gen LoRA and Fix LoRA. Following training on the enhanced code dataset, Gen LoRA has developed robust RTL code generation capabilities but exhibits limitations in self-reflection. To further enhance its capabilities of self-reflection, Gen LoRA is trained on the error-correction dataset, resulting in Fix LoRA.

The reason for employing two LoRA models, Gen LoRA and Fix LoRA, instead of a single LoRA is based on experimental results, which indicate that Fix LoRA, trained on the error-correction dataset, exhibits inferior performance compared to Gen LoRA in the RTL code generation task. This performance degradation may be attributed to the fact that training on a dataset containing syntactic errors could potentially weaken the model's overall performance. Therefore, OriGen adopts a two-LoRA approach, where Gen LoRA is responsible for generating the initial code, and Fix LoRA is tasked with rectifying syntactic errors, utilizing its specialized training on the error-correction dataset.

Figure 5 illustrates an example of self-reflection in the correction of Verilog code for a module that implements the functionality of the 7458 chip, which includes four AND gates and two OR gates.

The generated code initially contains a syntactic error: wires 'ply_temp' and 'p2y_temp' are incorrectly assigned values within always blocks. In Verilog, wires cannot be assigned values on the left-hand side of procedural assignments. This error is corrected by using the Fix LoRA model during the self-reflection process, following the instruction template shown in Figure 6. The regenerated

Task Name: 7458
The 7458 is a chip with four AND gates and two OR gates. Create a module in Verilog with the same functionality as the 7458 chip. It has 10 inputs and 2 outputs. In this circuit, p1y should be the OR of two 3-input AND gates: one that ANDs p1a, p1b, and p1c, and the second that ANDs p1d, p1e, and p1f. The output p2y is the OR of two 2-input AND gates: one that ANDs p2a and p2b, and the second that ANDs p2c and p2d.

Original Wrong Response:
```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y
);
    wire p1y_temp;
    assign p1y = p1y_temp;
    always @(p1a or p1b or p1c or p1d or p1e or p1f)
    begin
        p1y_temp = (p1a&p1b&p1c) | (p1d&p1e&p1f);
    end
    wire p2y_temp;
    assign p2y = p2y_temp;
    always @(p2a or p2b or p2c or p2d)
    begin
        p2y_temp = (p2a & p2b) | (p2c & p2d);
    end
endmodule
```

Errors:
11: error: 'p1y_temp' is not a valid l-value for a procedural assignment.
7:'p1y_temp' is declared here as a wire.
17: error: 'p2y_temp' is not a valid l-value for a procedural assignment.
13:'p2y_temp' is declared here as a wire.

Regenerated Code:
```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y
);
    assign p1y = (p1a & p1b & p1c) | (p1d & p1e & p1f);
    assign p2y = (p2a & p2b) | (p2c & p2d);
endmodule
```

Figure 5: Example of Self-Reflection

code resolves these issues by employing continuous assignments for 'p1y' and 'p2y', rather than assigning values within always blocks, ensuring that the logic accurately reflects the intended operations.

3.5 VerilogFixEval Benchmark

To assess the ability of various models to reflect and improve from Verilog compiler error messages, we developed the VerilogFixEval benchmark. This benchmark consists of code samples that failed compilation verification, along with corresponding natural language instructions and compiler error messages. The faulty RTL code samples were selected from those generated by LLMs that performed similarly to GPT-3.5 on the VerilogEval benchmark. This selection method was adopted because OriGen achieves a relatively high compilation pass rate. By including code samples from poorly performing LLMs, the benchmark ensures a diversity of errors while minimizing potential bias in test results that could favor OriGen due to errors produced by OriGen itself.

During the evaluation process, the model will receive natural language instructions, faulty RTL code, and compiler error messages to correct the RTL code. The final evaluation metrics consist of two components: syntactic correctness and functional correctness.

Debug Instruction
As a professional Verilog designer, you are tasked with debugging a Verilog module that has some errors. Below are the details of the assignment, the original Verilog code with some syntax and functional errors, and the error messages produced by the compiler. Please review this information and provide a corrected version of the code.
Task:
{task}
Original Code:
{original_code}
Compiler Error Message:
{error}

Figure 6: Debug Instruction Template

4 EVALUATION

4.1 Experimental Setting

For our pre-trained model, we selected the DeepSeek-Coder-7B-Instruct model, as it exhibits the best performance in Verilog code generation among all 7B models, to the best of our knowledge. It is to ensure that our pre-trained model possesses strong capabilities in the domain of Verilog code generation, providing a solid foundation for further fine-tuning and evaluation.

To evaluate the performance of Verilog code generation, we selected two representative benchmarks: VerilogEval [20] and RTLLM [23]. The former, VerilogEval, originated from approximately 150 Verilog tasks on the HDLBits website, which were manually converted to create VerilogEval-Human and generated by GPT-3.5 to produce VerilogEval-Machine. The latter benchmark, RTLLM, consists of 29 Verilog tasks with more diverse levels of difficulty, closely aligned with real-world design tasks. Both benchmarks employ the widely adopted pass@k evaluation metric to assess the correctness of the generated code's functionality. In this metric, if any one of the k samples passes the unit test, the problem is considered solved.

$$pass@k := \mathop{\mathbb{E}}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

where we generate $n \geq k$ samples for each instruction in which $c \leq n$ samples pass testing. We choose $n = 10$ in experiments.

To assess the models' capability for self-reflection, we utilized the VerilogFixEval benchmark, as discussed in Section 3.5. This benchmark employs the pass@1 metric to evaluate the syntactic and functional accuracy of the rectified RTL code.

Table 1: Comparison of functional correctness on VerilogEval [20] and RTLLM [23]

Source	Name	VerilogEval-human(%)			VerilogEval-machine(%)			RTLLM(%)
		pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@5
Commercial LLM	GPT-3.5 [1]	35.6	48.8	52.6	49.4	72.7	77.6	44.8
	GPT-4 2023-06-13 [1]	43.5	55.8	58.9	60.0	70.6	73.5	65.5
	GPT-4 Turbo 2024-04-09 [1]	54.2	**68.5**	**72.4**	58.6	71.9	76.2	65.5
	Claude3-Haiku [2]	47.5	57.7	60.9	61.5	75.6	79.7	62.1
	Claude3-Sonnet [2]	46.1	56.0	60.3	58.4	71.8	74.8	58.6
	Claude3-Opus [2]	**54.7**	63.9	67.3	60.2	75.5	79.7	**69.0**
Open Source Models	CodeLlama-7B-Instruct [27]	18.2	22.7	24.3	43.1	47.1	47.7	34.5
	CodeQwen1.5-7B-Chat [3]	22.4	41.1	46.2	45.1	70.2	77.6	37.9
	DeepSeek-Coder-7B-Instruct-v1.5 [12]	31.7	42.8	46.8	55.7	73.9	77.6	37.9
Verilog-Specific Models	ChipNeMo [19]	22.4	-	-	43.4	-	-	-
	VerilogEval [20]	28.8	45.9	52.3	46.2	67.3	73.7	-
	RTLCoder-DeepSeek [21]	41.6	50.1	53.4	61.2	76.5	81.8	48.3
	CodeGen-6B MEV-LLM [24]	42.9	48.0	54.4	57.3	61.5	66.4	-
	BetterV-CodeQwen [26]	46.1	53.7	58.2	68.1	79.4	84.5	-
OriGen (ours)		54.4	60.1	64.2	**74.1**	**82.4**	**85.7**	65.5

4.2 Model Training

We employ the LoRA (Low-Rank Adaptation) [14] method to train the model's capability in generating RTL code. This approach allows for enhancing the model's specific abilities in RTL code generation while minimizing the impact on its other capabilities. For all the training processes, we employ the float16 mixed precision method, although the model is trained in bfloat16 precision. We use the AdamW optimizer with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, along with cosine learning rate decay for scheduling. The warm-up ratio is set to 0.03, and the batch size is 8.

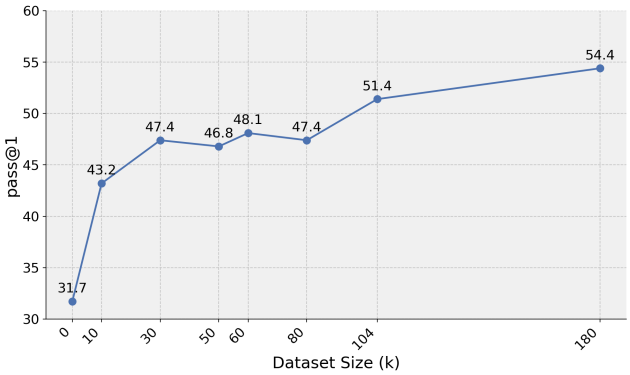

Figure 7: Pass@1 Variation with Dataset Size

4.3 Functional Correctness

Table 1 presents the results on the VerilogEval benchmark. To ensure fairness, we did not utilize the self-reflection feature for this comparison and generated code in a single attempt, like other models. The models compared include closed-source commercial LLMs such as GPT-3.5/GPT-4, Claude3-Haiku/Sonnet/Opus, general open-source code models [3, 12], and models customized for RTL code generation [19–21, 24, 26].

In the VerilogEval benchmark [20], OriGen achieves pass@1 scores of 54.4% in the Human category and 74.1% in the Machine category, outperforming remarkably all other alternatives designed for RTL code generation. For instance, compared to the best-performing fully open-source RTLCoder [21], OriGen surpasses it by 12.8% on the Human benchmark. Additionally, compared to commercial closed-source models, OriGen demonstrates exceptional performance. It significantly outperforms GPT-4 (2023), which was previously considered a key benchmark. Furthermore, it exceeds the teacher model Claude3-Haiku, underscoring the effectiveness of filtering and self-reflection in code augmentation.

Among the state-of-the-art models, OriGen outperforms Claude3-Haiku/Sonnet across all metrics and surpasses GPT-4 Turbo in the pass@1 metric while being only slightly inferior to Claude3-Opus by less than 1% on the Human benchmark pass@1 metric. Moreover, as shown in Table 1, it achieves the best performance on the Machine benchmark, notably outperforming other models including Claude3-Opus and GPT-4 Turbo. The performance gap between Human and Machine categories primarily stems from descriptions generated by GPT-3.5 for the Machine category. In contrast, OriGen is trained on a vast amount of synthesized data, enabling it to excel with problems generated by LLMs.

We also conduct experiments to investigate the relationship between dataset size and model performance, as depicted in Figure7. The performance is evaluated using the pass@1 metric on VerilogEval Human. We examined dataset sizes ranging from 0 (representing the base model without fine-tuning) to approximately 180,000 samples (utilizing the entire fine-tuning dataset).

The results demonstrate a clear positive correlation between dataset size and model performance. Starting from a pass@1 score of 31.7 for the base model, we observe a sharp initial increase to 43.2 with just 10k samples. This is followed by a more gradual but consistent improvement as the dataset size increases, eventually reaching a peak performance of 54.4 with the full dataset of 180k samples. Notably, the performance gains are not linear. The most substantial improvements occur in the early stages of data addition,

Table 2: Comparison of Models on VerilogFixEval

Model	Syntactic correctness(%)	Functional correctness(%)
CodeQwen1.5-7B-Chat [3]	27.6	10.4
RTLCoder [21]	46.6	15.3
DeepSeek-Coder-7B [12]	50.7	19.0
GPT-3.5 [1]	40.7	13.1
GPT-4 Turbo 2024-04-09 [1]	69.2	37.1
Claude3-Haiku [2]	48.9	23.5
Claude3-Opus [2]	71.9	**39.8**
OriGen (Ours)	**89.1**	33.5

with diminishing returns as the dataset size approaches its maximum. This pattern underscores the critical role of data quantity and diversity in model performance for further improvements beyond a certain scale.

To prevent our model from over-fitting on the VerilogEval benchmark, another benchmark RTLLM [23] is also used to evaluate. Similar results are observed on RTLLM as shown in Table 1, where OriGen significantly outperforms other models, achieving performance comparable to GPT-4 Turbo and Claude3-Opus.

In summary, OriGen significantly outperforms all non-commercial models across all metrics on both benchmarks and achieves comparable performance to the current state-of-the-art commercial closed-source models.

4.4 Capability of Self-Reflection

The evaluation metrics of VerilogFixEval consist of two components: syntactic correctness and functional correctness. Syntactic correctness assesses whether the generated rectified RTL code successfully compiles. Functional correctness further evaluates whether the generated rectified RTL code passes simulation tests.

As illustrated in Table 2, experimental results on VerilogFixEval demonstrate that OriGen achieves the best performance in syntactic error correction, surpassing GPT-4 Turbo by 19.9%. This indicates that after training on the code-correction dataset, the model acquires powerful self-reflection and code rectification capabilities. For functional correctness, OriGen significantly outperforms RTLCoder by 18.2% and falls short of the best-performing model, Claude3-Opus, by only 6.3%. This showcases OriGen's ability to consider the functional correctness of the generated code while rectifying syntactic errors.

4.5 Ablation Studies

We perform two ablation experiments to investigate the efficacy of the code-to-code augmentation method and to examine the model's self-reflection capability before and after training on the error-correction dataset. For the ablation study of the code-to-code data augmentation method, we compared the performance of models trained on RTL code dataset before and after code-to-code augmentation method.

Table 3 demonstrates that across multiple evaluation metrics, the model trained on the dataset enhanced by code-to-code augmentation significantly outperforms the model trained on the unaugmented dataset. This indicates that our model's outstanding performance is primarily attributed to the code-to-code augmentation methodology, as training on the unaugmented dataset does not yield substantial improvements.

Table 3: Ablation Study of Code-to-Code Augmentation

Benchmark	Metric	Baseline(%)	Augment(%)
VerilogEval-Human [20]	Pass@1	41.6	54.4
VerilogEval-Machine [20]	Pass@1	62.5	74.1
RTLLM [23]	Pass@5	41.4	65.5

Table 4: Ablation Study of Error-Correction Dataset on VerilogFixEval

Model	Syntactic Correctness(%)	Functional Correctness(%)
baseline	53.8	23.4
baseline + error message	63.5	25.6
finetune	82.7	31.9
finetune + error message	89.1	33.5

For the ablation study of the error-correction dataset, the results are presented in Table 4. We evaluate four scenarios, considering the model's performance before and after training on the error-correction dataset, as well as with and without the use of error messages from the compiler. The former is denoted as "baseline" and "finetune," indicating whether the model is trained on the error-correction dataset, while the latter is represented by "error message."

The results demonstrate that after training on the error-correction dataset, the model significantly outperforms its pre-training performance in both syntactic correctness and functional correctness. Additionally, the trained model exhibits a reduced reliance on error messages from the compiler.

5 CONCLUSION

This paper introduces OriGen, an open-source framework for RTL code generation. It proposes a novel code-to-code augmentation methodology to generate high-quality, large-scale RTL code datasets, which enhances the model's training data. The framework also introduces a self-reflection mechanism that allows OriGen to autonomously fix syntactic errors by leveraging compiler feedback, thereby improving its code generation accuracy. Furthermore, we construct a dataset to improve the model's capability of self-reflection based on compiler error messages and erroneous code and develop a benchmark to evaluate this capability. Experimental results demonstrate that OriGen remarkably outperforms other open-source alternatives in RTL code generation, surpassing the previous best-performing LLM by 12.8% on the VerilogEval-Human benchmark and is comparable with GPT-4 Turbo. Moreover, OriGen exhibits superior capabilities in self-reflection and error rectification, surpassing GPT-4 by 19.9% in syntactic correctness on the VerilogFixEval benchmark.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of China (Grant No. T2325001).

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[4] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 1–6.

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 33–36.

[6] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, et al. 2024. Data is all you need: Finetuning LLMs for Chip Design via an Automated design-data augmentation framework. *arXiv preprint arXiv:2403.11202* (2024).

[7] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[8] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. 2009. An introduction to high-level synthesis. *IEEE Design & Test of Computers* 26, 4 (2009), 8–17.

[9] Steve Dai and Zhiru Zhang. 2019. Improving scalability of exact modulo scheduling with specialized conflict-driven learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[10] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, and Stefan De Gendt. 2023. A deep learning framework for verilog autocompletion towards design and verification automation. *arXiv preprint arXiv:2304.13840* (2023).

[11] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. 2023. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[13] Hsuan Hsiao and Jason Anderson. 2019. Thread weaving: Static resource scheduling for multithreaded high-level synthesis. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[14] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[15] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 865–870.

[16] Liancheng Jia, Yuyue Wang, Jingwen Leng, and Yun Liang. 2022. EMS: efficient memory subsystem synthesis for spatial accelerators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 67–72.

[17] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[18] Katikapalli Subramanyam Kalyan, Ajit Rajasekharan, and Sivanesan Sangeetha. 2021. Ammus: A survey of transformer-based pretrained models in natural language processing. *arXiv preprint arXiv:2108.05542* (2021).

[19] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176* (2023).

[20] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[21] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2023. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617* (2023).

[22] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[23] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. 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, 722–727.

[24] Bardia Nadimi and Hao Zheng. 2024. A Multi-Expert Large Language Model Architecture for Verilog Code Generation. *arXiv preprint arXiv:2404.08029* (2024).

[25] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 27–32.

[26] Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. 2024. BetterV: Controlled Verilog Generation with Discriminative Guidance. *arXiv preprint arXiv:2402.03375* (2024).

[27] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[28] Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. 2024. A Survey of Neural Code Intelligence: Paradigms, Advances and Beyond. arXiv:2403.14734

[29] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[30] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2023. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* (2023).

[31] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887* (2023).

[32] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543* (2023).

[33] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal* 2002, 99 (2002), 3.

[34] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2024. Chateda: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[35] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A multi-level intermediate representation for hardware synthesis methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[36] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent Computer Interfaces Enable Software Engineering Language Models.