UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ITERATIVE LLM-DRIVEN RTL DESIGN USING HDLAGENT

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Mark Zakharov

June 2024

The Thesis of
Mark Zakharov is approved:

Professor Jose Renau, Chair

Professor Dustin Richmond

Professor Yi Zhang

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Iterative LLM-Driven RTL Design using HDLAgent

by

Mark Zakharov

Trained on billions of lines of GitHub code, Large Language Models (LLMs) have emerged as formidable programming assistants, demonstrating pronounced proficiency in popular programming languages such as Python and C++. This proficiency is attributed to the extensive availability of open-source code in these languages. However, the training of LLMs is an immensely resource-intensive process. For instance, Meta's Llama2-7B model required 184,320 GPU hours for training on 2048 A100 GPUs, incurring costs nearing one million dollars. Moreover, the construction of a robust model necessitates vast datasets. The substantial amount of time, money, and data needed to train an LLM on a new language may dissuade the creation of future languages, as they will not be compatible with LLMs "out of the box".

In response to these challenges, this work introduces HDLAgent, an innovative LLM agent accompanied by a language analysis framework. HDLAgent employs chain-of-thought reasoning and transfer learning to equip LLMs with knowledge of languages absent from their original training datasets. Focusing primarily on Hardware Description Languages, HDLAgent utilizes summaries of language tutorials and a selection of straightforward code examples as contextual foundations to facilitate circuit design in the specified target language. The efficacy of HDLAgent is further enhanced

through a compiler error feedback loop. This loop adopts a retrieval-augmented strategy, mapping compiler error messages to their corresponding explanations and corrective suggestions. Through this approach, HDLAgent significantly accelerates the process of enabling LLMs to understand and operate within previously unfamiliar programming languages, potentially revolutionizing the adaptability and utility of LLMs in the ever-evolving landscape of programming language development.

Acknowledgments

I would like to express my deepest gratitude to Professor Jose Renau for his unwavering support and guidance over the past four years. His mentorship and the opportunity he provided for me to embark on this unforgettable journey have been invaluable.

This endeavor would not have been possible without the tireless efforts of Farzaneh Rabiei, with whom I have had the privilege to collaborate for nearly a year. Her development of the HDLEval benchmark was vital to the success of this project.

I am also deeply thankful to Professors Dustin Richmond and Yi Zhang for their continuous support, tutelage, and invaluable advice regarding the industry.

Finally, I extend my heartfelt thanks to my friends, family, and loved ones. Their strength, encouragement, and inspiration have fueled my passion for the wonderful field of computing.

Chapter 1

Introduction

1.1 Introduction

Recent advancements in Large Language Models (LLMs), such as OpenAI's GPT, Google's Gemini, and Mistral AI's Mixtral, are revolutionizing the field of programming languages. LLMs are making it easier for newcomers to learn and use programming languages by providing intelligent assistance, generating code snippets, and offering context-aware suggestions. LLMs have the potential to significantly reduce the entry barriers to programming.

The contribution of this work is to enable LLMs to operate with various Hardware Description Languages (HDLs). HDLs often form niche communities where existing LLMs may underperform. Although Verilog remains the most popular HDL, it is showing its age[1] and languages like Chisel3 [4], PyRTL [7], and DSLX [12] are being

[1] The original Verilog was designed in 1983, and modern versions like System-Verilog are semantically compatible with it.

designed as alternatives. Unless off-the-shelf LLMs can work with new HDLs, there is a significant disincentive to create new HDLs.

Some of the most popular and high-performing LLMs are closed-source. This means that to make HDLs accessible to a wider community, the solution must work with LLMs that cannot update their training. Given the significant impact of LLMs, successful HDLs should support off-the-shelf models without requiring years of waiting for the next LLM training cycle to potentially incorporate the new HDL. If LLMs can handle new HDLs, they can facilitate easier adoption. Conversely, if LLMs cannot handle new HDLs, they become an additional obstacle that impedes the adoption of new HDLs.

The key contribution of this work is to enable the effective utilization of both open-source and proprietary LLMs across a diverse range of HDLs. This specifically addresses the challenge of generating accurate and functional code in HDLs that have proven difficult for existing LLMs.

To achieve the outlined contributions, this paper proposes an AI Agent (**HDLAgent**) that incorporates state-of-the-art AI coding agent techniques, specifically adapted to support HDLs with limited LLM support. To test the effectiveness of HDLAgent, the evaluation employs both Verilog-specific and HDL-neutral benchmarks. This will underscore how much of the current testing infrastructure for RTL design is tailored specifically to the Verilog language and is not suitable for assessing performance across multiple HDLs, similiar to current LLMs' knowledge bases.

AI Agents [45] utilize multiple LLM queries and external tools, employing a

state machine to manage the LLM's workflow and guide its interactions. Many agents leverage three main techniques: self-reflection, Retrieval Augmented Generation (RAG), and grounding. Self-reflection techniques like Chain-of-Thought (CoT) [34] use LLMs to improve their own responses. RAG provides context or examples for queries. Grounding uses external tools to reduce hallucinations or mistakes.

HDLAgent incorporates these state-of-the-art AI coding agent techniques but adapts them to the challenges of supporting HDLs. The self-reflection adaptations are tailored to manage module inputs/outputs specific to each HDL. RAG includes an HDL summary, provides few-shot examples, and compile-error fix examples for LLMs that may have limited knowledge of the given HDL. Grounding leverages compiler feedback to improve accuracy and reliability.

HDLAgent uses context (RAG) and compiler feedback (grounding) to carefully formulate queries for unfamiliar HDLs, where traditional few-shot approaches consistently fail. For instance, using HDLAgent with Mix-8x7B yields a 44% success rate when writing Chisel and just 3% tests pass without HDLAgent. Other LLM like GPT-3.5o, go from 3% success rate for DSLX to 48% with HDLAgent. HDLAgent also benefits LLMs with Verilog. For Mix-8x22B, the success rate goes from 13% to 53%.

An alternative path not explored in this work is to fine-tune the model. It is possible to fine-tune an LLM for a given language like some LLMs fine-tune for Verilog [27]. This work does not perform fine-tuning because it is not always possible. For LLMs like OpenAIs GPT-4T, fine-tuning is not allowed at the time of this work, limiting it to in-context learning.

LLMs are evaluated using benchmarks like HumanEval [5] to quantify the LLMs' coding capacity. Since HumanEval tests Python, a more recently proposed HumanEval-X [43] extends the tests to cover multiple languages. HumanEval-X creates a test for each supported language to verify the correctness of a response, but it focuses only on popular languages, not HDLs. VerilogEval [18] follows the HumanEval principle, but it employs Verilog tests instead of Python. As a futher parallel, HDLEval [**?**] has all of its problems written in plain English, allowing them to be implemented in any HDL, similiar to HumanEval-X's range of languages. Hence, the testing strategy mirrors the established benchmarks for LLM programming performance, such as HumanEval and HumanEval-X, but is uniquely tailored to assess RTL design, thereby extending the benchmarking paradigm to the domain of HDLs.

In summary, the paper contributions is HDLAgent, which enhances the performance of multiple HDLs across multiple LLMs. It shows that HDL design can leverage LLMs even before training examples or data is available. The evaluation also provides several insights that future HDLs designers to avoid LLM support issues.

Chapter 2

Related Works

2.1 Related Work

There are two main approaches to improve LLM output: Fine-tuning and Retrieval Augmented Generation (RAG). These techniques can be iteratively combined to develop Agents that produce even better results.

Fine-tuning is the process of adjusting the parameters of a LLM on a specific dataset or task to improve its performance. As such, fine-tuning can be applied to optimize a given LLM for a new language. RTLCoder [19] fine-tunes a 7B mistral model with GPT generated synthetic Verilog data. HDLAgent uses off-the-shelf LLMs without fine-tuning.

URIAL [16] bypasses the need for fine-tuning by enriching prompts with illustrative examples. These prompts resemble the few-shot format used by HDLAgent, incorporating both format and examples. While URIAL has shown effectiveness in

circumventing the need for instruction alignment, HDLAgent further illustrates the possibility of learning previously unknown languages.

Agents [45] iterate through LLMs with three main techniques to improve performance: Self-reflection, few-shot with RAG, and grounding.

Self-reflection techniques use a sequence of interactions with the LLM instead of a simple question/answer. CoT [34] is an example of self-reflection. Lumos [40] uses CoT to enable simpler LLMs to outperform more advanced LLMs. These studies highlight some significant progress in this rapidly evolving field. Recent works [36] propose an optimization method to find the best prompt.

Few-shot in-context learning [3, 17] with RAG [15] employs instructions and several examples (few-shot) related to the question or prompt to enhance efficiency. Various techniques exist for constructing such prompts with an extended context. This technique of querying an embedding database to augment the context is known as Retrieval Augmented Generation (RAG).

Grounding involves verifying or checking the LLM's response using an external tool. While this isn't always feasible, in code generation, a compiler or testbench can validate and find issues with the LLM-generated response. This triggers further iterations with the LLM.

Agents with self-reflection, RAG, and grounding have been applied to improve code generation. If we ignore the HDL target, and focus on generic programming languages like Python or C++, several works [26, 44] show that many errors can be fixed by grounding the generated code against a compiler or test. Recent works [8, 20–

22, 32, 35] leverage this fact and propose Agents that ground the code generation with a compiler or tests. Other works [11, 23, 41] propose Agents to iterate over test bench results to fix the generated code.

Besides CoT, some notable self-reflection for code generation include: Self-planning [14] proposes a planning stage or self-reflection before code generation; Self-Debug [6] proposes how to improve code generation by generating explanations in the intermediate steps; ChatCoder [33] uses self-reflection to paraphrase and elaborate on the initial question.

Early work [28, 29, 37] with LLMs and Verilog avoid the Agents because LLMs like GPT-4 are already reasonable trained for Verilog. Several AI-based chip design competitions [9, 10] required designs implemented in Verilog. Looking at the top performers, they tend to use GPT-4 and focus on combinational modules where the top level module IO is fully specified. In all the cases, the human-in-the-loop guides the LLM to fix problems with the generated code and iterate over the testbench results.

To avoid a human-in-the-loop, a coding Agent can be applied to Verilog generation. The same coding Agent ideas with self-reflection, RAG, and grounding can be applied to Verilog. Concurrent works include AutoChip [30], RTLFixer [31], and HDLdebugger [39].

AutoChip [30] uses testbench feedback to ground the generated Verilog. It is similar to Self-Edit [41] and Self-Repair [23], but with a Verilog focus. Since the focus is simulation errors, there are no clear few-shot contexts like in HDLAgent, where a few-shot can be generated for each error/warning message.

RTLFixer [31] uses ReAct [38] for self-reflection, and RAG for grounding compiler errors. Similiarly, HDLAgent uses HDL descriptions and few-shot examples to guide code generation, and compiler fix samples to address compiler errors. These compiler fix samples resemble RTLFixer human-generated explanations for various error messages.

HDLDebugger [39] fine-tunes CodeLlama, but instead of fine-tuning like RTL-Coder to generate better Verilog, it fine-tunes CodeLlama to fix code generation. HDLDebugger uses the compiler error messages to ground the generation, and applies it to the fine-tuned CodeLlama to fix the code. HDLDebugger is a different approach than when available (Publication August 2024) could be applied to HDLAgent in the steps to fix compiler errors. One issue is that it will require fine-tuning for each HDL. From the provisional paper, HDLDebugger does not seem to apply self-reflection.

Chapter 3

HDLAgent

HDLAgent is an AI Agent (Section 2.1) that leverages the latest novelties in AI coding Agents but adapts them to HDLs where the LLM may be unfamiliar with the given HDL.

LLMs are capable of transfer learning [24,42]. HDLAgent leverages this feature to allow LLMs to handle HDLs with little training data. HDLAgent allows the LLM to transfer knowledge from other HDLs that it trained on, like Verilog, to the new HDL. This enables the LLM to adapt learned programming languages to target programming languages, similar to human learning processes [25].

Figure 3.1 illustrates HDLAgent's execution flow. The main drawback that HDLAgent resolves is low knowledge of an HDL. To address this, HDLAgent provides a "main context" (Section 3.0.1) and a "compiler context" (Section 3.0.2). The "main context" provides an HDL summary and few-shot examples. The "compiler context" helps ground the LLM code generation with compiler error suggestions.

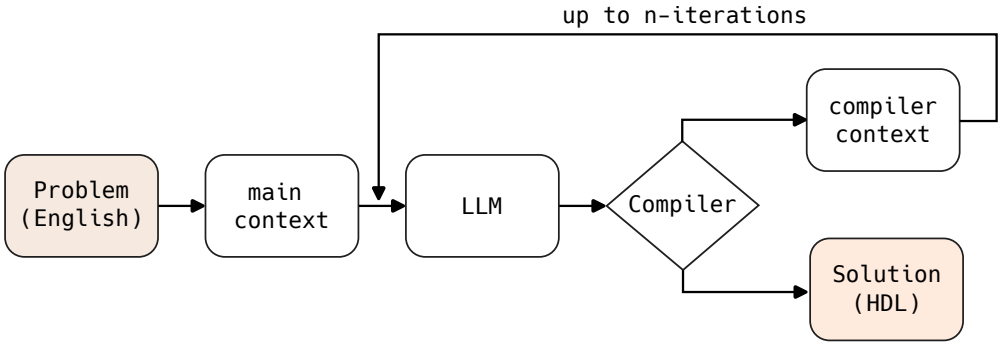

Figure 3.1: HDLAgent flow leveraging compiler feedback.

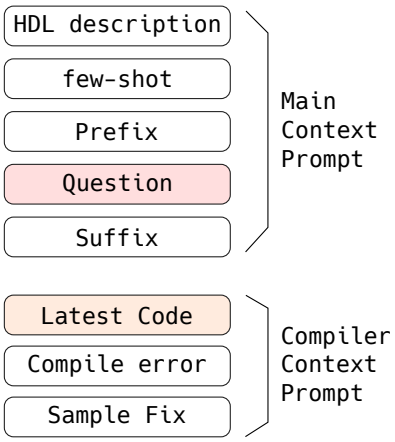

Figure 3.2: HDLAgent Main and Compiler context prompt components.

3.0.1 Main Context

The ”main context” in HDLAgent informs the LLM about the specific HDL used. Figure 3.2 depicts the main context, comprised of four elements: HDL description, few-shot, Prefix, Question, and Suffix.

The HDL description summarizes the HDL and depends on the LLM's familiarity with the HDL. As the evaluation shows, the HDL description can be optimized per LLM, but for simplicity, we pick the HDL description that performs best in Mix-8x7B

and GPT-3.5n. The reason is that these two LLMs have lower knowledge of HDLs like Chisel, DSLX and PyRTL. The HDL description is not helpful only when the LLM is proficient in a given HDL, such as Verilog.

Even for LLMs that can accommodate the entire HDL reference manual within their context window, success rate and token cost are enhanced when using a summary of the manual. As demonstrated in the evaluation, utilizing an HDL description significantly improves the success rate.

Since the whole tutorial is not practical or beneficial for the evaluated LLMs. We instead use and HDL description summary. For the HDL Description, we use LLMs with large context to summarize the HDL reference manual. For summarizing, we try GPT-4 and GPro-1.0 because these two models can handle larger contexts.

The evaluation shows the results for several prompt options, but for PyRTL and Chisel the best prompt was generated by GPT-4 when asking "PyRTL is a Hardware Description Language with the following reference documentation and tutorial. Create a documentation that is useful for LLMs trying to generate PyRTL code. The generated documentation should have some code snippets and any language syntax that it is a not typical HDL".

For DSLX, the best summary was created by GPro-1.0. The prompt was the same as the PyRTL and Chisel best prompt, but it added the following sentence at the end of the prompt "Be concise and avoid examples with similar syntax." As it can be observed, there is only a small variation on the prompt.

In addition to the HDL description, the "main context" provides few-shot ex-

amples to cover common HDL operations or areas of confusion. The patterns addressed are bit operations, reductions, loops, multiple options, and a multiply-add block.

Bits correspond to a simple example of bit-manipulation and concatenatation. Reduction is a simple NOR reduction over a given input.

Loops are an example of a confusing process. In DSLX all variables are immutable, but loops have a special syntax for reduction variables. Adding something like the text in Listing 3 to the HDL context solves cases where a loop is created by the LLM.

To introduce more complex examples, the few-shot also includes a simple ALU that selects between different operations, and a multiply-add example. Clearly, other examples could be added, but LLMs seem capable to perform knowledge transfer from just few examples.

The Prefix follows the few-shot. It is a brief statement directing the original Question with something like: "The following statements describe the problem to be addressed in DSLX.".

The Suffix, added post-question, limits the scope of the problem to address. It includes instructions such as "respond with valid program syntax only, without additional English explanations" and specifically tailors HDL input and output formats. Handling IOs is crucial for HDLs with multiple output options, necessitating directives to maintain output integrity and naming consistency. For DSLX, Suffix includes directives like "do not split the outputs into individual bits" and "variables assigned to the output struct should have the same name as the struct fields". This Suffix con-

cept is essential even when using Verilog because it can use interfaces, structs, or plain Verilog-2001 syntax.

The HDLAgent Suffix allows different HDLs to interface with Verilog. Nevertheless, keeping names and common syntax is an essential problem the Suffix must address, even for a single HDL use case.

3.0.2 Compiler Context

HDLAgent's compiler context iteratively corrects inaccurately generated HDL code. It grounds the LLM generated code to correct and provide feedback on hallucinations.

In the case that there is a compiler error, HDLAgent builds a query (Figure 3.2) that starts with "main context", ignoring non-code responses. It follows with the latest code snippet, then a sentence saying that "the previous code has the following compile error" followed by the compiler error message. If HDLAgent has a compiler example fix of how to address the compiler error, the context finishes adding the "sample fix".

The example fix is similar to RTLFixer [31] in that it provides an explanation to fix Verilog messages. We instead extend it to cover multiple HDLs and explain that a given HDL requires a special way to handle some syntax.

HDLAgent shows the whole latest code snippet. We tried a method inspired by CWhy [2], which provides a few lines of code around the compiler error message. This worked fine for some LLMs like GPT-4, but failed with other LLMs. This delta approach reduces token usage, but it increased error rates so we did not use it in the

evaluation. As LLMs improve, it may be something worth reconsidering.

3.0.3 Prompt Optimizations

Besides the previous main and compiler context there are also several subtle but important optimizations:

- Placing the prompt after the context achieves better results [13].
- HDLAgent approach avoids the chat-like history with all the previous code generations and fixes. Keeping the original question iteration but not the compiler error fixes achieves better results [30]. We did a quick test with HDLAgent and DSLX. Avoiding a history with all the error fixes had a 5% improvement in GPT-4 and a 27% in Mix-8x7B.
- Most LLMs generate code snippets in quoted sections, but not always. Even worse, it is common to write english explanations even thought the prompt explicitly asks to just write code. To address this, for each language we have a filter/detector that removes English and finds code boundaries. For example in Verilog it allows preprocessor directives and code between module and endmodule. Without this, some smaller LLMs fail very frequently.

3.0.4 LLM Cost

Our approach approximates LLM cost by the number of tokens utilized. This metric serves as a practical proxy for monetary cost and the compute resources needed.

As the context length increases, so do the costs and computational demands.

While the cost of input and output tokens might vary, we simplify by assuming fewer tokens equate to lower costs. This token-centric view influences decision-making regarding iteration methods and context types. Future studies could investigate efficiency metrics like error rate $\times$ tokens.

Our current focus prioritizes accuracy while striving to reduce token usage. A crucial aspect to consider is the "stateless" nature of LLMs, meaning each call requires a complete context. APIs like OpenAI and Mixtral lack a "chat-like" interface that accumulates context across queries. GPro-1.0 is able to do so. This means that, depending on the cost model for LLMs like GPro-1.0, it may be more efficient to keep the history. Nevertheless, GPro-1.0 context is not long enough to handle several iterations. For this work, we flush the history and ignore cost models and consider just total token usage after all of the HDLAgent iterations.

3.0.5 Usage and Features

HDLAgent processes two types of input: JSON files and "Spec" files, the contents of which are used to formulate the LLM queries. For the benchmarking results discussed in this paper, JSON files were utilized, each containing specific fields and their corresponding content: "instruction" provides an English description of the desired circuit behavior, "pipeline stages" is an integer indicating the appropriate LEC script for combinational or pipelined circuits, "response" includes the correct Verilog implementation for comparison in LEC, and "interface" contains the Verilog-style mod-

ule declaration to standardize the I/O and adjust the LLM's implementation to align with the "response" for LEC verification.

Spec files, intended for non-benchmarking user interactions, are formatted as YAML files. These include all JSON fields except "response," as there is no Verilog implementation for comparison. The YAML format was chosen for Spec files as it is user-friendly and easily editable, offering a straightforward way for users to modify the LLM's output if needed. HDLAgent can also use an LLM to generate a Spec file from a simple text document outlining and refining the desired behavior. This provides the benefits of automatic formatting and, more importantly, an increase in clarity from the user-supplied prompt.

Having an LLM implement a design based on an improved description that it previously generated from an original human-written prompt is an instance of self-reflection. This prompting technique leads the LLM to iterate upon and correct its own work. Past research, such as the Reflexion framework [**?**], has shown that self-reflection improves results in the HumanEval benchmark. HDLAgent incorporates a similar approach with its "testbench" flow, initiated after the initial RTL version is generated.

A separate conversation is created, prompting the LLM to generate an assert-based Verilog testbench. The results of the testbench and the testbench itself are fed back into the primary conversation if there are any failures. This feedback prompts the LLM to correct the RTL. The testbench conversation does not have access to the RTL, preventing it from "cheating" by designing the testbench based on the RTL. Instead, it

relies solely on the English description of the desired behavior. This process produces a testbench independent of the implementation, designed by the LLM, and used to critique and improve its own work through self-reflection.

The creation of a separate conversation for testbench generation allows for the use of a different LLM from the one working on the RTL. Past works have shown that using a multi-agent debate model can significantly improve results [**?**], even with just two distinct agents. HDLAgent permits the use of multiple "testbench" LLMs, each critiquing the implementation. However, for evaluation, only self-reflection and the two-agent debate model were tested, seen in 5.1.5.

Chapter 4

Setup

4.1 Setup

Table 4.1 lists all the languages used in the evaluation and the compiler versions used by this paper. When a date is provided it corresponds to the top-of-tree version at that given month. For Quality of Results (QoR), we use Yosys synthesis results.

Table 4.1: Language Tools and Versions

Language	**Tool**	**Version**
Verilog	Yosys	0.35
Chisel	FIRRTL	3.5.0-RC2
pyRTL	pyRTL compiler	0.10.2
DSLX	XLS	3/2024

Table 4.2 shows the LLMs used. Many LLMs, including GPT-3.5o, are not deterministic. They have produced differing outcomes for the same example under identical prompt conditions. OpenAI recently proposed a new API to address this issue, providing a seed, but this solution still needs to be fully implemented across all

Table 4.2: LLMs used in the evaluation

LLM	**Version**	**Updated**	**Context**
GPT-4	gpt-4-1106-preview	4/2023	128000
GPT-3.5n	gpt-3.5-turbo-0125	9/2021	16385
GPT-3.5o	gpt-3.5-turbo-1106	9/2021	16385
GPro-1.0	gemini-1.0-pro-001	2/2024	32720
Mix-8x7B	Mixtral-8x7B-instruct-v0.1	12/2023	32768
Mix-8x22B	Mixtral-8x22B-v0.1	3/2024	32768

LLMs. For fair evaluation, we avoid the deterministic settings and perform 1, 5, or 10 runs depending on the pass@k parameter.

Chapter 5

Evaluation

5.1 Evaluation

5.1.1 Overall Results

To understand the benefits of HDLAgent across LLMs, we plot each of the HDLs (Chisel, PyRTL, DSLX, and Verilog) against four benchmark tests: VH stands for VerilogEval-Human, VM stands for VerilogEval-Machine, HC stands for HDLEval-Comb, and HP stands for HDLEval-Pipe.

VerilogEval tests consist of several Verilog-specific questions. However, their evaluation does not fully demonstrate the potential of the LLM/HDLAgent as effectively as HDLEval (HC, HP) does. This is a common issue across HDLs with the exception of Verilog. The plots include VH and VM for reference, but the focus of the evaluation is HDLEval-Comb.

Each bar has five components to showcase the impact of different aspects of

HDLAgent. *Base* is the baseline or typical zero-shot LLM evaluation that does not use HDLAgent yet still keeps the HDLAgent prompts to format IOs and overall directions for code generation; *Description* adds the HDL Description (Section 3.0.1); *Few-shot* adds the few-shot related to the language used; *Compile* adds the compiler feedback and iterates up to 8 times to fix the code; and *Fixes* performs the same iterations but for each iteration provides a suggestion, alongside a generic example, on how to fix that given compiler error.

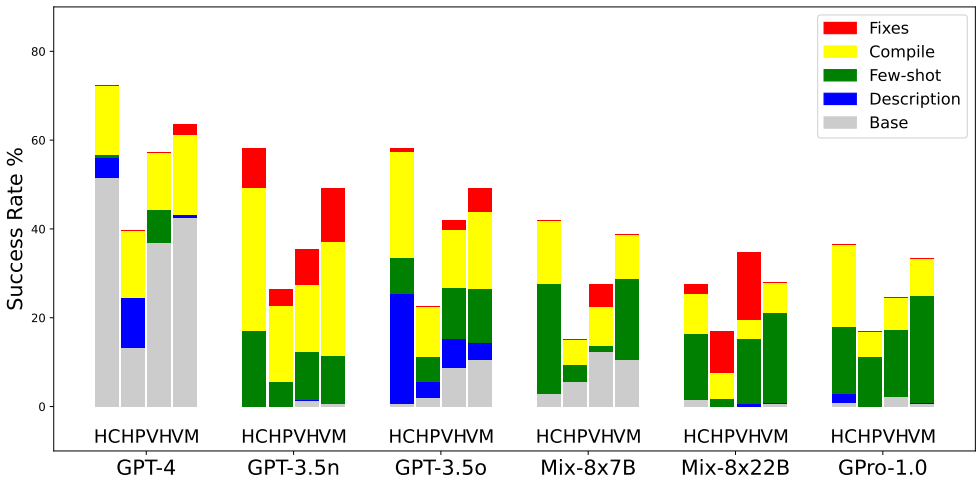

Figure 5.1: HDLAgent improves Chisel across all LLMs.

Chisel (Figure 5.1) is based on Scala; several LLMs know the basic syntax but only GPT-4 performs decently with Chisel. All the LLMs except GPT-4 had less than a 3% success rate with Chisel. Both ”main context” (Description + Few-shot) and ”compiler context” (Compile + Fixes) provide substantial benefit, demonstrating that all these components are necessary. Moreover, GPT-3.5o and GPT-3.5n perform better than high performing LLMs (GPT-4) without HDLAgent. HDLAgent is also able to improve

GPT-4, reaching a 69% success rate.

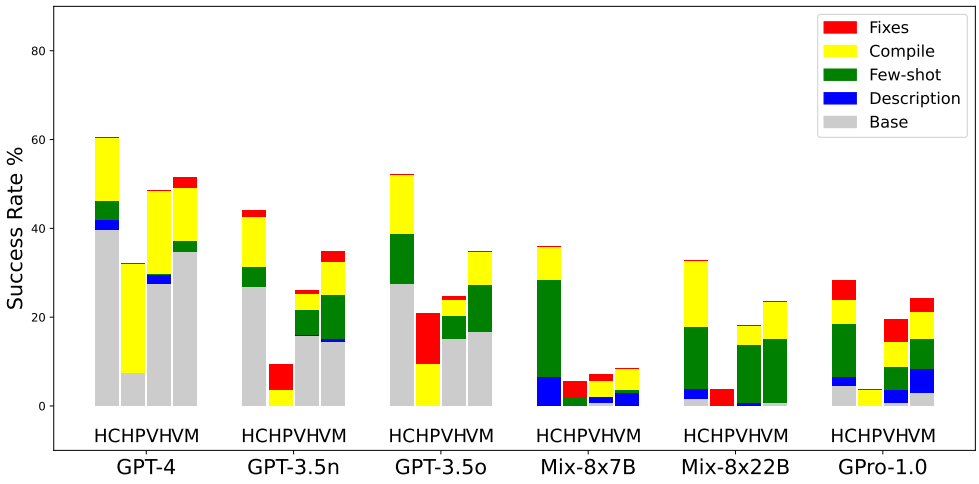

Figure 5.2: HDLAgent improves PyRTL across all LLMs.

PyRTL (Figure 5.2) is a Python-based HDL. OpenAI LLMs (GPT-4, GPT-3.5n, GPT-3.5o) can pass several tests without HDLAgent (Base); however, even these models have a low success rate, between 27% and 40% in PyRTL. When HDLAgent is enabled, the success rate increases to between 44% to 59%. Similar to the Chisel case, HDLAgent speedup comes from several factors.

As with Chisel, all the HDLAgent components are important but the "compiler context" (*Compile* + *Fixes*) is crucial for both PyRTL and Chisel. This is because these HDLs are Domain Specific Languages (DSLs), so the LLMs gets confused mixing the DSL hosts (Scala, Python) with their HDL-specific syntaxes. The compiler context is able to guide and solve the issues.

DSLX (Figure 5.3) is a Rust-like language. Since DSLX does not allow arbitrary pipelining, it cannot be evaluated against HDLEval-Pipe and it exhibits very poor per-

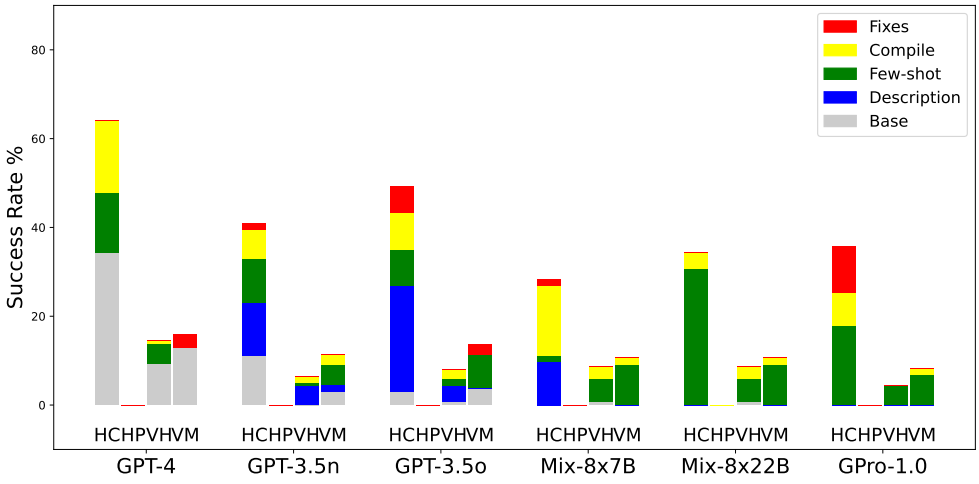

Figure 5.3: HDLAgent improves DSLX HDLEval-Comb across all LLMs.

formance with VerilogEval. GPT-4 has some knowledge of DSLX, and once again, HDLAgent significantly enchances the results across all LLMs. Unlike Chisel and PyRTL, DSLX is not a DSL. In this case, the "main context" (*HDL Description* + *Few-Shot*) is the biggest HDLAgent factor in this improvement, because explaining the Rust-like syntax and providing a few examples is more important than grounding the results with compile errors.

Verilog (Figure 5.4) has the best overall performance for the *Base*, which is when no HDLAgent is active. This is the expected behavior due to extensive training with Verilog syntax. It is also the only fair case where VerilogEval can be used to evaluate an HDL. Overall, HDLAgent has little impact on the models that are already proficient in Verilog; however, it has a significant impact on Mix-8x7B and Mix-8x22B, which have little Verilog knowledge. This is an interesting observation displaying how knowledge transfer can work even when the LLM is not familiar with Verilog.

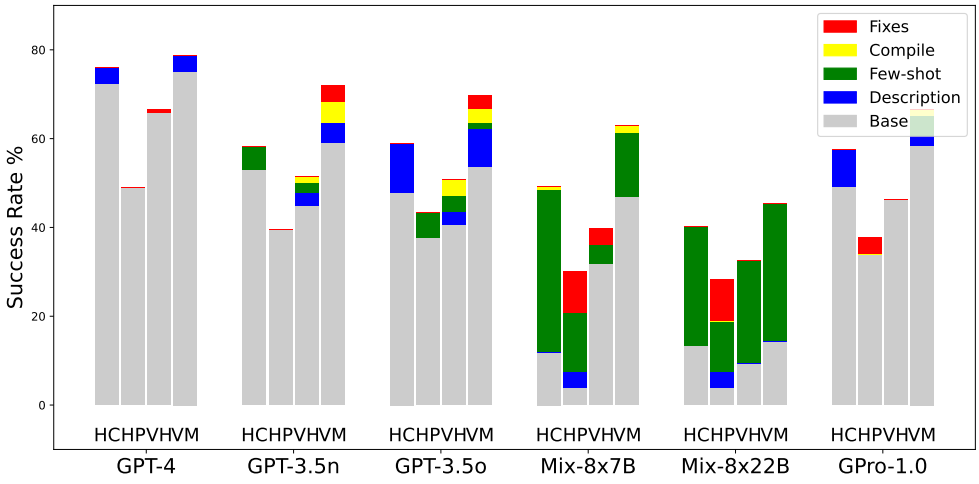

Figure 5.4: Verilog succeeds across benchmarks and LLMs .

One interesting observation regarding Verilog is that ”compiler context” does not help recover from many errors. Part of the reason is that Verilog error messages use Yosys, which has error messages more cryptic than PyRTL (Python), Chisel (Scala), or DSLX. A stricter Verilog compiler with more readable errors could have a different distribution, but over 80% of the Verilog generated code in HDLEval-Comb compiles correctly at a first attempt, so there are limits on what compiler error grounding can provide.

When comparing LLMs, one unexpected behavior is that Mix-8x22B has lower performance than Mix-8x7B. The model was recently released, but we observe a high failure rate when it is attempting to follow directions. Using a ”instruct” model will achieve better results. We kept the ”base” model because it provides interesting insights on how different models improve and behave with HDLAgent.

One of the main HDLAgent objectives is to enable LLMs to use new HDLs.

When comparing different LLMs like GPT-3.5o and GPT-3.5n across various HDLs, trends indicate that the performance of HDLs remains relatively consistent regardless of the specific LLM used. For instance, with GPT-4, Verilog shows the highest success rate at 75%, which is close to its lowest for PyRTL with 59%. This pattern of performance is consistent across all tested LLMs. The worst performing LLM (Mix-8x22B) has a 53% Verilog success rate, while PyRTL the worst performing HDL Mix-8x22B has a 28% success rate. Without HDLAgent, many LLMs had a zero success rate.

5.1.2 HDLAgent Context Insights

This section discusses HDL Description and few-shot selection. One straightforward approach is to use the HDL reference manual directly. While this is feasible with GPT-4, Mix-8x7B, and GPro-1.0 due to their large context windows, its generally less effective than using a summarized HDL description. For example, using a full reference instead of a summary does not change results for GPro-1.0, but reduces success rate from 77% to 66% for GPT-4, and from 59% to 33% for Mix-8x7B. This indicates that future LLMs need to improve handling of length contexts, as all evaluated LLM struggle with this.

Figure 5.5 shows the DSLX, PyRTL, and Chisel success rate as different reference manuals are summarized for HDLAgent. Each bar shows a different LLM reference summarization prompt (Section 3.0.1) sorted by accuracy. The breakdown is the contribution of the few-shot examples and the HDL description. Interestingly, adding Few-shot always improves results, and removing HDL Description and just keeping

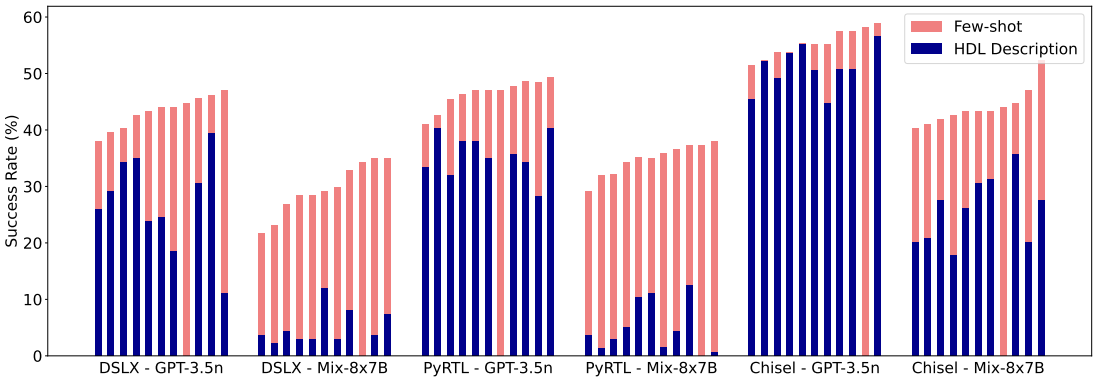

Figure 5.5: HDL description and few-shot help LLMs to improve results.

few-shot examples is a reasonable alternative. In some HDL/LLM combinations like Chisel/GPT-3.5n, using either Few-shot or Description works. For other combinations like DSLX/Mix-8x7B, HDL Description helps but Few-shot is necessary. Optimal results require both Few-shot and HDL Description.

5.1.3 Pass Sensitivity

Pass@k is a popular method that measures how results can be improved by generating multiple attempts. A k=5 means that when 5 LLM tries are used, at least one has the correct code generation. Table 5.1 shows tests passed for HDLEval-Comb for multiple LLMs and multiple pass@k values (1,5,10). Only the HDLEval-Comb results are shown, as it is the most representative benchmark for a multi-lang evaulation.

Less popular HDLs benefit more from higher pass@k values. For example, DSLX shows a 1.22 to 2.08 times improvement in test pass rates from pass@1 to pass@10. Verilog has between 1.16 and 1.45 times. This discrepancy is likely because the LLM,

		Verilog			Chisel			PyRTL			DSLX		
		k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10	k=1	k=5	k=10
GPT-4	Base	97	103	111	69	88	92	53	79	85	46	79	85
	HDLAgent	102	109	111	97	103	107	81	92	98	86	100	104
GPT-3.5n	Base	71	96	100	0	5	9	36	63	67	15	32	41
	HDLAgent	78	93	98	80	97	100	59	79	88	55	80	88
GPT-3.5o	Base	64	93	99	1	6	14	37	60	71	4	19	25
	HDLAgent	79	92	100	79	91	99	70	78	89	65	86	91
GPro-1.0	Base	66	97	105	1	5	12	6	17	31	0	0	0
	HDLAgent	77	96	99	49	84	88	38	66	77	48	74	82
Mix-8x7B	Base	16	39	50	4	12	17	0	1	2	0	0	0
	HDLAgent	66	86	95	60	80	86	48	71	82	38	72	79
Mix-8x22B	Base	18	65	78	2	12	18	2	8	13	0	0	6
	HDLAgent	72	96	101	35	79	89	39	67	72	47	75	81

Table 5.1: Pass@k results for HDLEval-Comb for different LLMs with just a Base query or with HDLAgent.

unfamiliar with the language, starts from an incorrect baseline and struggles to correct errors through compiler feedback. Not being able to fix is very rare in Verilog but over 10% of the DSLX tests have this problem. The higher the pass@k, the easier to avoid. Once the code compiles correctly, the failure rate for all the HDLs is comparable. This means that if a future HDLAgent improved the iterations or better selection point, it could further improve results.

Figure 5.6 provides more insights on the pass@k results. It shows the increase in accuracy as HDLAgent iterates with the compiler like GPro-1.0 across HDLs. We choose GPro-1.0 because it is one of the LLMs that need more iterations to converge. For Verilog, it converges very fast but for the other HDLs it needs 6 to 8 iterations to converge. More iterations do not improve, but changing the starting point like pass@5

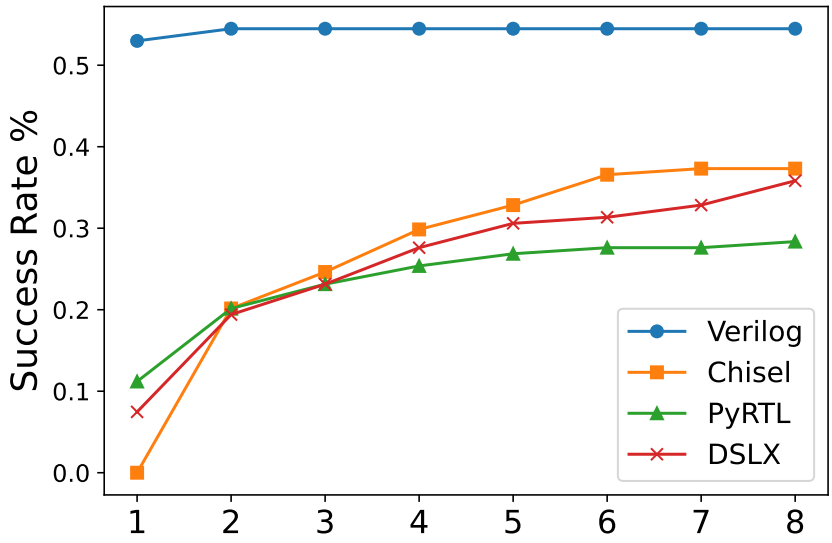

Figure 5.6: GPro-1.0 converges in a few iterations.

improves the results. Overall, 8 iterations is enough across languages because increasing iterations does not help in success rate but hurts in token usage.

When pass@5 and 8 iterations are used (Table 5.1), HDLAgent HDLs (Chisel, PyRTL, DSLX) perform better or equal to the same LLM with Verilog (Base). This is a main contribution on the paper showing that HDLAgent enables the use of small community HDLs.

5.1.4 Time and QoR

Execution time is an important metric for any AI Agent. Figure 5.7 shows the execution time boxplot for HDLEval-Comb with different LLMs, where both successful and failed tests are considered. All the languages besides Verilog go through a translation process to Verilog that adds overhead. In HDLAgent, the execution time is

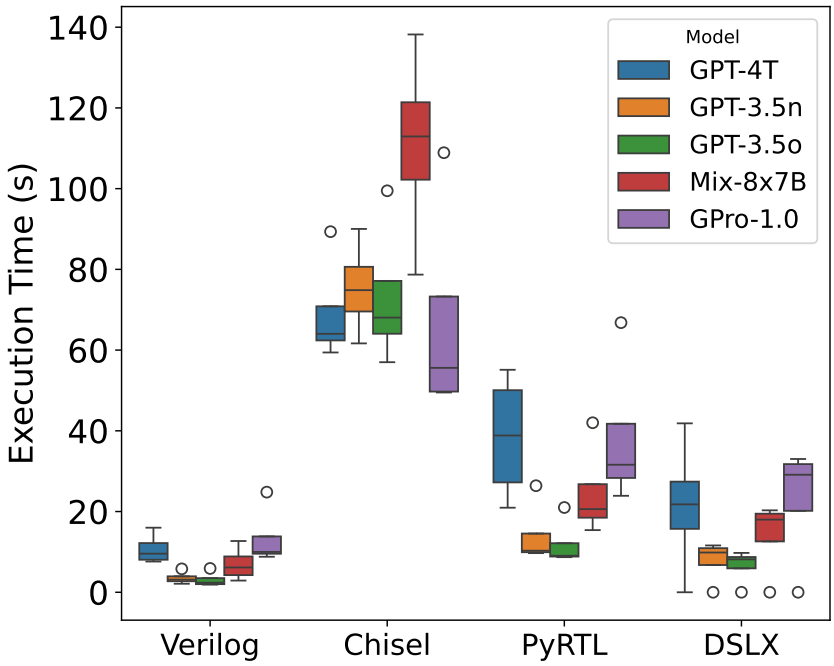

Figure 5.7: LLM and HDL affect total HDLAgent execution time.

a function of $\frac{tokens}{second}$, number of iterations, and external compiler speed.

Comparing across HDLs, the main outlier is Chisel. Around 2/3 of the execution time is spent in the FIRRTL compiler to generate Verilog. GPT-4 is faster because it has less errors and hence less iterations. PyRTL and DSLX are also slower than Verilog, but this is in part due to the additional iterations.

Comparing across LLMs, GPT-3.5n and GPT-3.5o tend to be faster overall as they combine less error iterations and speed of results. External $\frac{tokens}{second}$ benchmarking [1] points that GPro-1.0 is roughly 30% faster than GPT-3.5n and 4 times faster than GPT-4. HDLAgent results are different because of iterations and speed.

The Quality of Results (QoR) is crucial in hardware generation. Figure 5.8

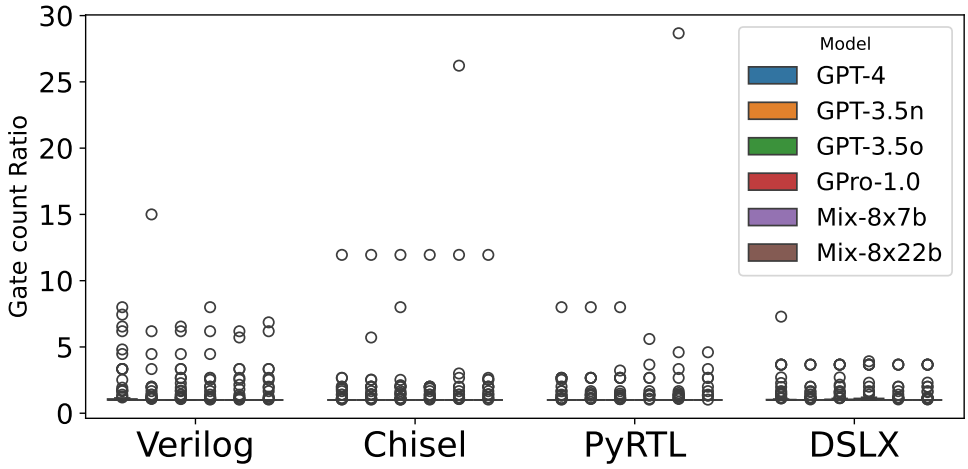

Figure 5.8: QoR is consistent across LLMs but different across HDLs.

shows the gate count ratio compared to the best implementation. A ratio of 1 indicates optimal gate count, while 2 indicates double the gate count. This figure only includes the successful runs using HDLAgent with HDLEval-Comb. The plot reveals significant Qor variation compared to the best implementation. Typically, the average varies due to one or two outliers. For instance, in PyRTL generated by Mix-8x7B, the average gate count is 1.63, but removing two outliers brings it down to 1.12. This means that the LLMs sometimes generate very inneficient code but it is not so frequent. An second observation suggests that GPT-4 may underperform; however, it successfully handles larger and more complex tests that affect the results. A third observation is that the code generated by various LLMs displays comparable efficiency. Among these, DSLX appears to be slightly more efficient. In DSLX generated by GPT-4, 80% of the generated code is the optimal with a 1 ratio. This seems to imply that an efficient compiler like XLS combined with a popular syntax can result in better results for generated HDL code.

5.1.5 Self-Reflection with Multi-Agent Debate

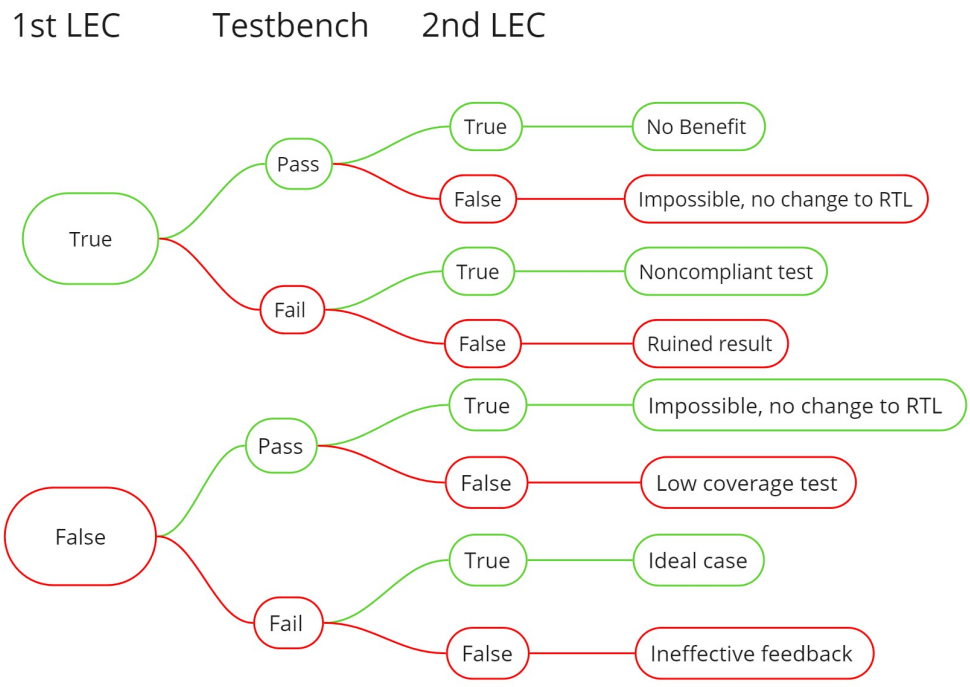

Figure 5.9: All possible scenarios of the

As mentioned in Section 3.0.5, HDLAgent includes an option to generate Verilog testbenches for the produced code. This section examines the results of passing the design through a testbench and iterating upon it in case of any testbench failures before attempting LEC. Figures 5.10 and 5.11 show the results of using the best-performing and worst-performing LLMs, GPT-4 and Mix-8x7B, respectively, as both designers and testers when writing Verilog for HDLEval-comb. If the initial version of the design failed its testbench, a second version would be generated and this version would then attempt LEC. The results are categorized into test cases that fail and those that pass LEC in their initial version, with the red part of the stacked bar representing those that failed

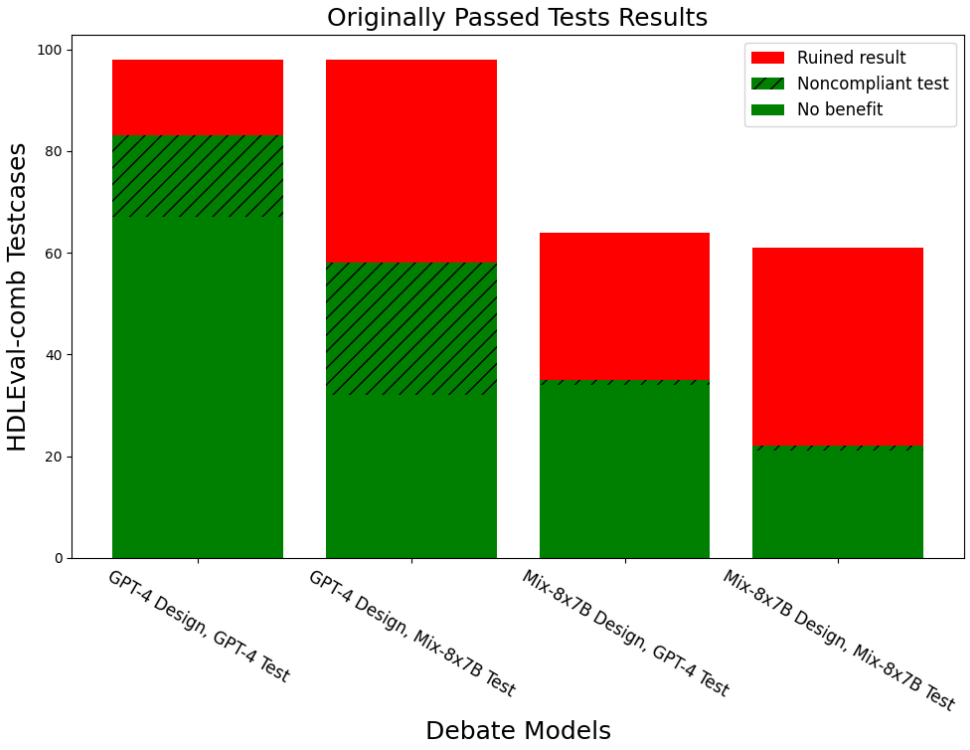

Figure 5.10: Introducing a testbench into the generative loop worsens results

LEC in their second version, and the green part representing those that passed LEC in their second version.

Figure 5.9 illustrates all possible scenarios in the HDLAgent testbench debate model. Two of these scenarios are labeled ”impossible” because the result of LEC would be different despite the design not changing. The remaining scenarios, along with the results shown in Figures 5.10 and 5.11, are explained in the rest of this section.

This paragraph discusses cases where the initial version of the design would have passed LEC. The ”no benefit” case occurs when the testbench passes, meaning the design did not undergo a second iteration, and the testbench made no difference. The

Figure 5.11: Introducing a testbench into the generative loop provides little benefit

”noncompliant test” case occurs when the initial version of the design would have passed LEC, but the testbench failed; however, the second iteration of the design still passes LEC. This suggests that the designer LLM has a strong understanding of the design requirements, consistently passing LEC, while the test failed, indicating noncompliance with the design requirements. This points to LLMs being better designers than testers and ignoring feedback from tests that do not align with the originally stated requirements. In contrast, the ”ruined result” case occurs when both the testbench and the second LEC fail, despite the initial version of the design passing LEC. Here, feedback from the testbench failure misled the LLM, prompting unnecessary and ultimately in-

correct changes, ”ruining” the design. Figure 5.10 shows this is a somewhat common scenario, especially when Mix-8x7B is the tester, implying that this debate model is disadvantageous.

This paragraph discusses cases where the initial version of the designs would not have passed LEC. The ”low coverage test” case occurs when the testbench passes, meaning there is no second version of the design, so LEC remains a failure. The test was too simple to provide any insights on how to improve the design, possibly due to the LLM’s poor understanding of the design requirements. The ”ideal case” is when the testbench fails, providing feedback that allows for iteration on the design, resulting in the second version passing LEC. Figure 5.11 shows this is an uncommon scenario compared to the ”ruined result” scenario seen in Figure 5.10, further indicating that the debate model decreases performance. The final case is where the testbench fails, and its feedback provides no benefit, as LEC is not passed with the second iteration. This is referred to as ”ineffective feedback”.

Overall, the results suggest that HDLAgent’s current multi-Agent debate scheme provides no advantage for RTL design. Some tests that would have otherwise passed fail due to the feedback from failed testbenches. Developing a more cohesive arbitration scheme for inter-Agent debate, as well as fine-tuning for testbench generation and interpretation, is left for future work.

5.1.6 Insights for HDLs at the age of LLMs

The goal of this section is to show some shortcomings in HDLs that need to be addressed to improve accuracy in an LLM world.

5.1.6.1 Verilog

Verilog is the language that LLMs know the best. For top-performing LLMs like GPT-4, the main challenge lies in handling pipelining. Verilog allows for unrestricted pipelining, which deviates from the traditional Von Neumann architecture or non-hardware program structure. GPT-4 effectively generates combinational logic because a typical program without recursion or memory access can be directly translated to Verilog or combinational logic. Improving pipelining remains an open research question that needs to be addressed to enhance LLMs' performance hardware tasks.

5.1.6.2 Chisel

Besides the common pipelining issue, Chisel LLM code generation needs help to interface Chisel code to Verilog. As a part of compilation process, the Verilog generated module's IO appends "io_" to all names. Additional clock and reset signals are created by default, even if unused in the original Chisel code. Listing 1 shows the resulting Verilog from a compiled Chisel implementaion of a full adder circuit.

To interface modules, HDLAgent adjusts the IO to perform testing. Postprocessing is used to remove the unused signals as well as renaming those modified to their originals to match the circuit specification. This is necessary as the first step of the LEC

checks that the two modules' IOs match, otherwise a truthful comparison is impossible and the LEC fails.

In addition, Chisel shares a problem with PyRTL of being a Domain Specific Language (DSL), and the LLMs use incorrect syntax.

```
module full_adder(
  input   clock,
  input   reset,
  input   io_a,
  input   io_b,
  input   io_cin,
  output  io_sum,
  output  io_cout
);
```

Listing 1: Chisel IOs have name changes.

5.1.6.3 PyRTL

PyRTL shares common problems with Verilog and Chisel, but it also has a problem with semantics.

A DSL problem is when the LLM generates Python syntax to implement logic instead of the PyRTL syntax. Listing 2 invalid cases uses Python "inp 1" instead of the PyRTL shift right logical library call. Many of those problems generate errors which are catched by HDLAgent, and it is able to fix with further HDLAgent iterations.

Besides DSL problems, PyRTL has errors due to inconsistent semantics. In Verilog and Chisel, a right shift logical of a positive number reduces the size. For example if inp has 4 bits, and it is right shifted once, the output has 3 bits. Not in PyRTL, it has 4 bits but the most significant bit is a zero. Listing 2 showcases the

```
inp = pyrtl.inpput(4, 'inp')
out = pyrtl.Output(4, 'out')
out <<= inp ^ pyrtl.shift_right_logical(inp, 1)
# equivalent: out <<= pyrtl.concat(inp[3] ^ 0, inp[3] ^ inp[2], inp
    [2] ^ inp[1], inp[1] ^ inp[0])
# CORRECT    : out <<= pyrtl.concat(inp[3]    , inp[3] ^ inp[2], inp
    [2] ^ inp[1], inp[1] ^ inp[0])
# INVALID    : out <<= inp ^ (inp>>1)  # Invalid, >> is a python shift
     not PyRTL
```

Listing 2: PyRTL issues generating right shift.

problem in one HDLEval test. The most significant bit is xored with zero which is not the expected result.

5.1.6.4 DSLX

```
fn add_7_to_11() -> Outputs {
   //add values from 7 to 11 (exclusive)
   let base = u16:7;
   let res = for (i, accum): (u16, u16) in u16:0..u16:4 {
     accum + base + i
   }(u16:0);
   Outputs { result: res }
}
```

Listing 3: Rust DSLX special loop syntax.

DSLX presented a different set of challenges compared to DSLs like Chisel and PyRTL. Since it does not support unrestricted pipelining, only combinational logic is used in this section's feedback.

DSLX shares input/output generation issues with Chisel and PyRTL but faces even greater challenges. DSLX has a single unnamed output for functions named "out" during Verilog generation. DSLX solution to multiple outputs is to use a struct. HD-

LAgent addresses it by post-processing the generated Verilog and modifying the output port name to match the desired name. A better solution that requires DSLX semantic changes would be to adopt a Go-like syntax that allows for multiple named outputs and ensures Verilog generation respects those outputs.

Another interesting source of errors stems from DSLX being ”similar to Rust”. If the HDLAgent's HDL Description mentions that ”DSLX is similar to Rust...” it generates even more errors. Even without this sentence, the LLM sometimes generates legal Rust but illegal DSLX code. Some differences are easy to spot, such as DSLX's ”assert(cond)” versus Rust's ”assert_eq!(cond),” while others, like the presence of Rust annotations like ”#[test]” in DSLX code, are more subtle. To address the ”similar but not the same” syntax issues, it is suggested to avoid mentioning the similarity and catch any discrepancies during compilation time, generating a compile error for HDLAgent to fix.

A more complicated case involves semantic changes. Since DSLX cannot describe circuits with mutable variables, its expressions cannot describe state changes over a loop, making it incompatible with the Rust loop semantics. Instead, these expressions have an accumulator value separate from the iterator, creating a return value calculated by the body of the for loop. As shown in Listing 3, the for loop body sums the values between 7 and 11 by accumulating the base value of 7 and the iterator value in the range of 0 to 4 each loop ”iteration.” This deviation from standard loop semantics required a dedicated code snippet and explanation in both the initial and supplemental contexts to correct the LLM's often incorrect assumptions about DSLX's generative for loop syn-

tax. Addressing these changes will help LLMs to perform better with less HDLAgent iterations.

Chapter 6

Future Work and Conclusions

6.1 Future Work and Conclusions

Large Language Models (LLMs) have the potential to revolutionize computer science. This paper introduces HDLAgent, an AI Agent that significantly improves LLMs' HDL code generation for less popular HDLs like Chisel, PyRTL, and DSLX. Supporting multiple HDLs without LLM tuning is crucial for pioneering new HDLs that exploit LLM capabilities. The paper presents several challenges and recommendations for existing and future HDLs and AI Agents for chip design. The evaluation results point to pipelining vs combinational performance issues that the AI Agent community should address.

Evaluation reveals that HDLAgent enables emerging HDLs improving across all LLMs. The best performing LLM is GPT-4, without HDLAgent, GPT-4 had a 72% success rate for Verilog and 34% for DSLX. Once HDLAgent is applied, all the

HDLs have between 72% and 82% in pass@10. This means that HDLAgent is able to use knowledge transfers across HDLs, and boost the score beyond the GPT-4 Verilog baseline. This same property is shared across LLMs. The performance of all the HDLs with HDLAgent are always better than the Verilog performance when HDLAgent is not applied in pass@10.

This paper's evaluations of HDLAgent highlight the challenges faced by the AI Agent community in chip design. These include dealing with QoR outliers, decreasing execution time and token count, difficulties encountered with DSLs, improving compile error messages to guide LLMs, dealing with large outputs, pipelining…

In summary, HDLAgent enables the use of LLMs for HDLs beyond Verilog. The code for HDLAgent will be open-sourced to further benefit the community pointing to several insights and challenges.

Bibliography

[1] Artificial Analysis. `https://artificialanalysis.ai/models/gpt-35-turbo`. Online; accessed on April 2024.

[2] CWhy. website, November 2023.

[3] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. *arXiv preprint arXiv:2207.04237*, 2022.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens

Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[6] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023.

[7] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017.

[8] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023.

[9] efabless. Efabless 1st competition winners. `https://efabless.com/genai/challenges/1`, 2023.

[10] efabless. Efabless 2nd competition winners. `https://efabless.com/genai/challenges/2-winners`, 2023.

[11] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. Improving automatically generated code from codex via automated program repair. *arXiv preprint arXiv:2205.10583*, 2022.

[12] Google. XLS Website. https://github.com/google/xls/, 2022.

[13] Pranab Islam, Anand Kannappan, Douwe Kiela, Rebecca Qian, Nino Scherrer, and Bertie Vidgen. Financebench: A new benchmark for financial question answering, 2023.

[14] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models, 2023.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[16] Bill Yuchen Lin, Abhilasha Ravichander, Ximing Lu, Nouha Dziri, Melanie Sclar, Khyathi Chandu, Chandra Bhagavatula, and Yejin Choi. The unlocking spell on base llms: Rethinking alignment via in-context learning, 2023.

[17] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better

and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022.

[18] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. *arXiv preprint arXiv:2309.07544*, 2023.

[19] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution, 2024.

[20] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

[21] Seungjun Moon, Yongho Song, Hyungjoo Chae, Dongjin Kang, Taeyoon Kwon, Kai Tzu iunn Ong, Seung won Hwang, and Jinyoung Yeo. Coffee: Boost your code llms by fixing bugs with feedback, 2023.

[22] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. *arXiv preprint arXiv:2302.08468*, 2023.

[23] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2023.

[24] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[25] Jean Clarice Scholtz. *A study of transfer of skill between programming languages*. The University of Nebraska-Lincoln, 1989.

[26] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: An empirical study, 2024.

[27] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation, 2022.

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

[29] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation, 2023.

[30] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth

Garg, and Ramesh Karri. Autochip: Automating hdl generation using llm feedback, 2023.

[31] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language models, 2024.

[32] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing, 2023.

[33] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. Chatcoder: Chat-based refine requirement improves llms' code generation, 2023.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[35] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[36] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2023.

[37] Kaiyuan Yang, Haotian Liu, Yuqin Zhao, and Tiantai Deng. A new design approach of hardware implementation through natural language entry. *IET Collaborative Intelligent Manufacturing*, 5(4):e12087, 2023.

[38] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[39] Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models, 2024.

[40] Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Lumos: Learning agents with unified data, modular design, and open-source llms, 2023.

[41] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada, Jul. 2023. Association for Computational Linguistics.

[42] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2023.

[43] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

[44] Li Zhong and Zilong Wang. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2024.

[45] Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, Shiding Zhu, Jiyu Chen, Wentao Zhang, Ningyu Zhang, Huajun Chen, Peng Cui, and Mrinmaya Sachan. Agents: An open-source framework for autonomous language agents, 2023.