RTL-BenchMT: Dynamic Maintenance of RTL Generation Benchmark Through Agent-Assisted Analysis and Revision

Jing Wang[†], Shang Liu[†], Hangan Zhou, Zhiyao Xie[*]
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{jwangjw,sliudx,hzhoubu}@connect.ust.hk, eezhiyao@ust.hk

ABSTRACT

This paper introduces RTL-BenchMT, an agentic framework for dynamically maintaining RTL generation benchmarks. Large Language Models (LLMs) assisted automated RTL generation is one of the most important directions in EDA research. However, current RTL benchmarks face two critical challenges: **(1) flawed cases in the benchmarks** and **(2) overfitting to the benchmarks**. Both challenges are difficult to resolve purely by manual engineering effort. To address these issues and *systematically reduce human maintenance cost*, we propose an automated agentic framework, **RTL-BenchMT**. RTL-BenchMT focuses on two key applications: **(1) automatically identifying and revising flawed benchmark cases** and **(2) automatically detecting and updating overfitting cases**. With the assistance of RTL-BenchMT, we conduct a thorough, in-depth analysis of flawed and overfitting cases and produce a refined benchmark suite that will be open-sourced to the community.

1 INTRODUCTION

The EDA field is undergoing a transformative shift with the emergence of large language models (LLMs). One of the most important applications is LLM-based automated RTL generation [2, 4, 5, 7–14, 16–18], in which LLMs generate the desired RTL design from a natural-language description. These works rely on open RTL benchmarks [7, 10, 14] to measure functional correctness and compare models. Despite tremendous advances, we find that existing RTL generation benchmarks suffer from two fundamental issues.

Challenge 1. Flawed cases in the benchmark. Existing RTL benchmarks inevitably contain flawed cases, which can misrepresent the true capability of LLMs. Some tasks exhibit inconsistencies between the design description and the reference testbench. Some others omit critical implementation details. Such flaws can cause otherwise correct designs to be marked as failures, leading to unfair or misleading evaluation. However, systematically identifying and revising flawed cases across large benchmarks is highly labor-intensive and requires substantial RTL and verification expertise.

Challenge 2. Overfitting on the benchmark. Public RTL benchmarks are easily overfitted by LLMs, leading to increasingly over-optimistic performance results. Since benchmarks must be publicly available for research purposes, new LLM solutions tend to overfit the benchmark data [3] either by memorizing training examples or exploiting superficial patterns in the descriptions. Detecting such overfitting is essential for a fair and unbiased evaluation of LLMs. To the best of our knowledge, there is no practical framework that can automatically detect and quantify overfitting on RTL generation benchmarks.

In this work, we propose **RTL-BenchMT**, an agentic framework for dynamic maintenance of RTL generation benchmarks.

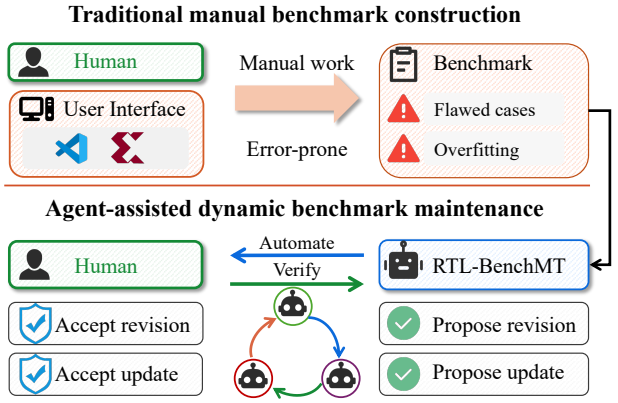

Figure 1: *(1) Flawed cases* and *(2) overfitting* are two significant challenges for RTL generation benchmarks. *RTL-BenchMT* resolves the challenges by dynamically maintaining benchmarks. *RTL-BenchMT* contributes in two important aspects: (1) automatically identifying and revising flawed cases and (2) automatically detecting and updating overfitting cases.

As illustrated in Figure 1, RTL-BenchMT organizes multiple specialized agents into an automated workflow that continuously analyzes benchmarks, revises flawed cases, and probes overfitting. The framework supports two main capabilities: **(1) automatically identifying and revising flawed cases** and **(2) automatically detecting and updating overfitting cases**. Human engineers shall remain in the loop to review RTL-BenchMT suggestions and approve the final benchmark updates.

To support effective maintenance, we design multiple specialized agents with distinct roles. RTL-BenchMT orchestrates these agents through three processes. *Process 1* (*failure analysis*) involves the *failure analysis agent*, which inspects logs of failed cases and identifies potentially flawed benchmark cases. *Process 2* (*benchmark revision*) revises the design descriptions of the identified cases. In this process, a *description revision agent* first proposes a candidate revision, and a *description review agent* validates the revision according to strict rules. *Process 3* (*overfitting detection*) tackles overfitting by rewriting design descriptions without changing their semantics; this process involves a *description update agent* and a *description review agent*. Each agent is implemented under an *iterative reasoning* paradigm, where the agent follows a three-step loop: *generating thought*, *taking action*, and *obtaining observation*.

Flawed case identification and revision. RTL-BenchMT automatically pinpoints problematic tasks and proposes refined descriptions. With the help of RTL-BenchMT, we propose a set of **refined benchmarks**, which will be open-sourced to the public. Using *Process 1* and *Process 2*, RTL-BenchMT runs multiple LLMs

[*]Corresponding author.
[†]Equal contribution.

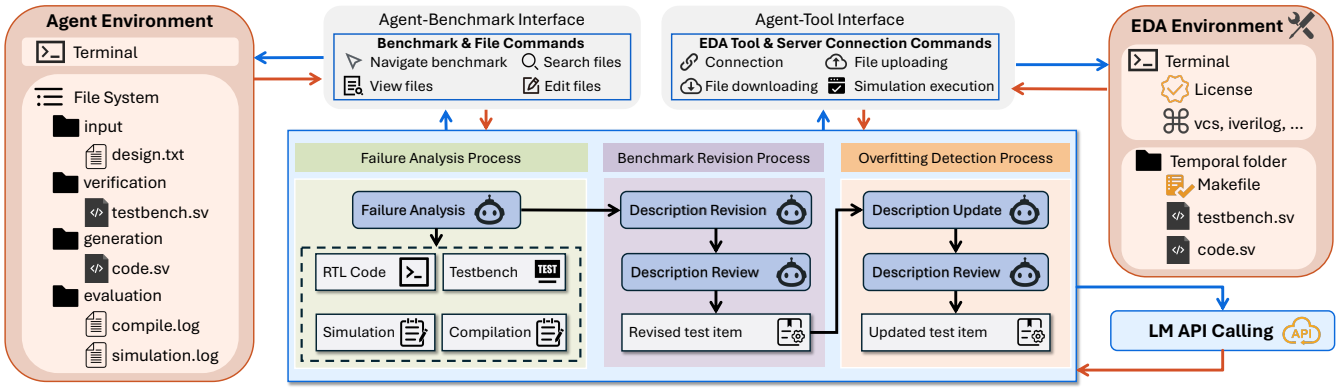

Figure 2: Overview of RTL-BenchMT agentic framework. The multi-agent system interacts with the environment through specified interfaces. The agent has three important automated phases: *(1) failure analysis process, (2) benchmark revision process, and (3) overfitting detection.*

on the benchmarks and aggregates consistently failing cases. Then RTL-BenchMT drives the revision agent to identify the flawed cases. Then, the analysis agent will propose a revision of the flawed cases. Finally, the review agent will validate the revision with strict rules.

Overfitting detection and updating. RTL-BenchMT rewrites descriptions to expose possibly overfitting models that rely on superficial patterns. *Process 3 (overfitting detection process)* controls description-update and review agents that generate semantically equivalent descriptions. The LLMs will be evaluated and compared based on updated descriptions. A model is considered overfitted if it passes on the original description but fails on a rewritten one. This simple criterion provides an automatic per-case and per-model signal of overfitting strength, while the rewritten descriptions also increase benchmark diversity for future evaluations.

The rest of this paper is organized as follows. Sec 2 introduces the RTL-BenchMT framework and agent design. Sec 3 discusses identified flawed cases and the corresponding revision strategies. Sec 4 provides quantitative results on flawed-case identification and overfitting detection.

2 RTL-BENCHMT AGENTIC FRAMEWORK

In this section, we introduce **RTL-BenchMT**, an agentic framework for dynamic maintenance of RTL generation benchmarks, as shown in Figure 2. In the following section, we first provide an overview of *RTL-BenchMT* (Section 2.1), including both the execution processes and agents. Then we provide a detailed introduction of techniques utilized in *application 1: flaw identification and revision* (Section 2.2), and *application 2: overfitting detection and updating* (Section 2.3). Finally, we introduce the infrastructures (Section 2.4), including the environments and interfaces.

2.1 Overview of RTL-BenchMT

The RTL-BenchMT framework consists of three main processes, as illustrated in Figure 2. We first provide an overview of the three main processes: *(1) failure analysis process*, *(2) benchmark revision process*, and *(3) overfitting detection process*. Within the framework, the *manager agent* orchestrates the three core processes to support the two key applications. Specifically, *(1) The failure analysis process* is in charge of identifying flawed cases in benchmarks. In this process, `failure analysis agent` performs the core task: *identifying the flawed cases* by our curated `analysis reasoning template`. *(2) The benchmark revision process* revises the flawed cases based on the identification results. In this process, `description revision agent` will first propose a revision, and `description review agent` will validate the revised description with strict `semantic rules`. *(3) The overfitting detection process* detects overfitting cases through `description updating strategies`. In this process, the `description update agent` first generates description variations by modifying the format while preserving the original semantics and functionality. The LLMs' performance on the description variation will reveal instances of overfitting.

Agents of RTL-BenchMT. All agents in the *RTLBench-MT* framework follow a basic *iterative reasoning* paradigm. At each iteration i, the agent performs the three-step actions: `(1) generating thought, (2) taking action`, and `(3) obtaining observation`. Each agent has an `Action` list: 'COMPARE_CODES,' 'REASON,' 'CHECK_INSTRUCTION,' etc. At each iteration i, the agent chooses an action (denoted as $\mathcal{A}_i$) based on the thought (denoted as $\mathcal{T}_i$), then receives an observation (denoted as O_i) from the environment (`Env`). Given the observation from the previous step, O_{i-1}, the agent generates the thought $\mathcal{T}_i$ for the next step. Then the agent takes the action $\mathcal{A}_i$ for the next step based on the thought $\mathcal{T}_i$. After the action is taken, the new observation O_i will be returned. Such a process can be formulated as follows:

$$\mathcal{T}_i = \texttt{Agent}(O_{i-1});\ \mathcal{A}_i = \texttt{Agent}(\mathcal{T}_i);\ O_i = \texttt{Env}(\mathcal{A}_i) \quad (1)$$

Usually, the initial observation O_0 is the evaluation results. Take the *(1) failure analysis process* as an example, the initial observation O_0 for `analysis agent` is the evaluation results of the generated RTL code from LLMs. The *RTL-BenchMT* workflow will first execute a standardized evaluation across a diverse suite of LLMs and collect the most common failure cases. Such failure cases will be propagated as observation O_0 to the `failure analysis agent`. The agent will first generate thoughts $\mathcal{T}_1$ based on the evaluation results and decide the next actions $\mathcal{A}_1$ (e.g., 'CHECK_INSTRUCTION,' which reads the file of design description) and returns detailed contents of the design description $\mathcal{D}_{\text{design}}$ as a new observation O_1.

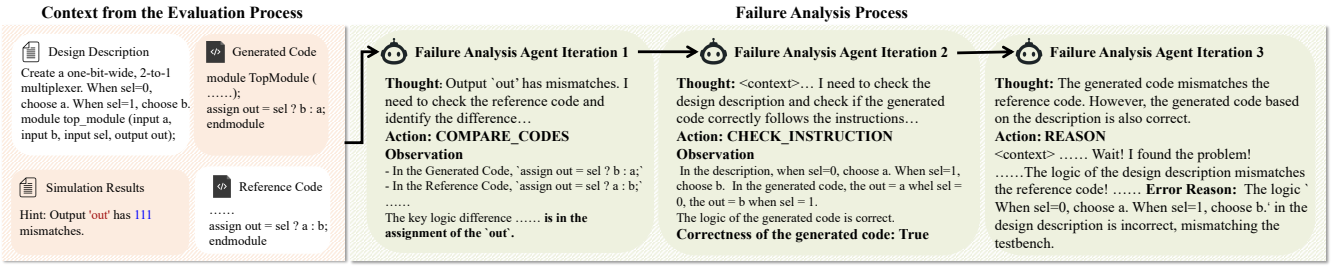

Figure 3: Example of automated flawed cases identification.

2.2 Application 1: Flaw Identification and Revision

The automated flawed case identification relies on effective failure case analysis, which is challenging and time-consuming. The RTL-BenchMT framework addresses the significant challenge by automating the failure analysis process. Figure 2 demonstrates the automated process, where the *failure analysis process* and *benchmark revision process* are executing the task. The **RTL-BenchMT** workflow will first execute a standardized evaluation across a diverse suite of LLMs and collect the most common failure cases. Such failure cases will be propagated to the failure analysis agent.

The **failure analysis agent** follows a three-step *thought-action-observation* reasoning paradigm. The analysis agent applies iterative observation and reasoning on the failure cases to identify the flaw within the design description. We curate a set of actions and an iterative reasoning paradigm for the analysis agent.

Algorithm 1: Analysis Reasoning Template

Iteration 1. Code Mismatch Analysis;
O_0 = `Simulation results`; `Action` $\mathcal{A}_1$ = `COMPARE_CODES`;
`Observation` O_1 = {`Generated Code`, `Reference Code`};
Iteration 2. Code Correctness Analysis;
`Thought` $\mathcal{T}_2$ = `Agent` (O_1); `Action` $\mathcal{A}_2$ = `CHECK_INSTRUCTION`;
`Observation` O_2 = Design Description $\mathcal{D}_{\text{design}}$;
Iteration 3. Description mismatch detection;
`Thought` $\mathcal{T}_3$ = Agent(O_2);
if *Code is correct in* $\mathcal{T}_3$ **then**
  `Action` $\mathcal{A}_3$ = `REASON`; `Context`= $\{O_1, O_2, O_3\}$;
  `Observation` O_3 = Agent(`Context`) = {`IS_FLAW`, $\mathcal{R}_{\texttt{flaw}}$};

Analysis reasoning template. Algorithm 1 illustrates the three-iteration template. In the first iteration (*iteration 1*), the agent analyzes the mismatches between the codes (observation O_1). Then, the agent checks the design description (observation O_2 in *iteration 2*) to verify if the generated code correctly implements the design specified in the description (if 'code is correct' in T_3). If the generated code is identified as correct, then the defects of LLMs are eliminated from the failure reason. Finally, the agent analyzes (Action A_3 = `REASON`) based on the previous observations (`Context`=$\{O_1, O_2, O_3\}$ in *iteration 3*). In this iteration, the agent will specifically focus on the design description and testbench to identify the mismatch. We provide an example in the following part to demonstrate the detailed results.

Example. Figure 3 illustrates the automated failure analysis process. At the input, the design description contains a mismatch in logic from the reference code in the testbench. At iteration 1, the failure analysis agent compares the generated code and the reference code. The agent identifies the mismatch between the generated code and the reference code. At iteration 2, then, the agent decides to thoroughly analyze the generated code based on the original description generation. In this iteration, the agent concludes that the generated code correctly implements the design based on the design description. Thus, the agent excludes the possibility of LLMs' ability defect. At iteration 3, this observation finally motivates the analysis agent to compare the design description and the reference code. Finally, the agent identifies the mismatching logic between the design description and the testbench.

Based on the key observation from the failure analysis, the **Benchmark Revision Process** updates the design description accordingly. The revision agent and the review agent collaborate to propose a revision. In this process, the *review agent* checks the revision with two strict rules. (1) The revision should not destroy the basic semantics. Under this requirement, the revision can slightly change the functionality to align with the testbench's requirements. (2) The revision should not directly leak the information from the testbench. For example, we have observed that the revision may directly leak code samples from the reference code as a shortcut for LLMs under test. This can also be viewed as a special case of revision destroying the semantics. Finally, the engineer can further validate the revision.

2.3 Application 2: Overfitting Detection and Updating

To detect the overfitting cases, we design *Process 3. overfitting detection*. In this process, *description update* agent creates more variance of design descriptions by rewriting the original description while keeping its semantics and functionality. A practical option is to shift the description styles. We pre-define a set of style templates for the *update agent* as a reference, e.g., 'Technical/Formal Style,' 'Educational/Tutorial Style,' 'Problem/Task Solving Style,' 'Specification/Documentation Style,' etc. Define the original description as `Desp`, the updated description as `Desp′`. The update process can be formulated as follows:

$$\mathcal{T}_{\text{Plan}} = \texttt{Agent}(\mathcal{D}_{\text{design}}, \{\texttt{Style}\});\ \mathcal{D}_{\text{variation}} = \texttt{Agent}(\mathcal{T}_{\text{Plan}}) \quad (2)$$

The *description update agent* will be provided a set of styles, denoted as $\{S_1, S_2, ...\}$. Then the agent will generate the `Plan` for updating. Finally, the *update agent* will update the design description based on the `Plan`.

2.4 Details of RTL-BenchMT

This subsection describes the **runtime environments** and **tool interfaces** that RTL-BenchMT exposes to LLM agents. From the agent's perspective, we distinguish between the environment where the agent itself runs and the environment where commercial EDA tools are executed.

Action	Interface usage & combination
COMPARE_CODES	1. view_file(generated_code.sv), 2. view_file(testbench.sv)
CHECK_INSTRUCTION	1. view_file(description.txt)
EVALUATION	1. connect(), 2. upload(design_folder) 3. compile(), 4. simulate() 5. download(results)
REVISE	1. edit_file(revised_description, description.txt)

Table 1: Illustration of commonly used agent `Actions` and their interface calls.

Environments. The LLM agents and orchestration scripts run in a lightweight host runtime, which we refer to as the *agent environment*. Commercial EDA tools are installed inside a Docker image running on a license server; we refer to this isolated runtime as the *EDA environment*. The agent never invokes EDA binaries directly. Instead, it issues high-level tool calls that internally start or attach to a Docker container, execute compilation and simulation commands inside the container, and collect log files or waveforms as outputs.

Interfaces. RTL-BenchMT exposes two groups of tools to the agent: *benchmark-related* tools and *EDA-related* tools. Benchmark-related tools form the **agent–benchmark interface**, which allows the agent to locate a test case, search for files, view their contents, and edit RTL, testbenches, or textual descriptions. EDA-related tools form the **agent–tool interface**, which mediates all communication with the Dockerized EDA environment. Typical operations include establishing an EDA session, uploading the current design directory, invoking compilation and simulation commands in Docker, and retrieving the resulting reports or waveforms.

These interfaces constitute the atomic operations in the agent `Action` set. Table 1 lists the most commonly used actions and their corresponding interface calls. For example, when the agent chooses the `EVALUATION` action, it sequentially invokes five interfaces: (1) connect to the EDA environment, (2) upload the design directory, (3) run compilation, (4) run simulation, and (5) download the simulation results for further analysis.

3 ANALYSIS ON IDENTIFIED FLAWED CASES

In this section, we provide a qualitative analysis of the identified flawed cases. For each example, we also provide suggested refinements of the cases. The flaws are mainly caused by ambiguity of descriptions, categorized into three types:

- *Syntax Ambiguity.* Design descriptions may accidentally include contents that lead to LLMs' syntax errors.
- *Functional Behavior Ambiguity.* Design description may contain unclear descriptions of the functional behavior, especially for the detailed operations.
- *Diagram Ambiguity*. The design description may contain diagrams. However, the manually written diagrams are prone to specifying incorrect or ambiguous logic.

3.1 Syntax ambiguity

We introduce three identified situations of syntax ambiguity: (1) *undefined module name*, (2) *unclear port type*, and (3) *syntax errors in code example.*

Undefined module name refers to the situation where the design description only specifies the functional logic while ignoring the name of the top module, which is required in the testbench.

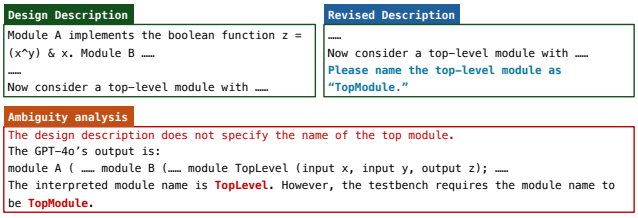

Figure 4: *Undefined module name.* The testbench requires the module name to be 'TopModule,' which is, however, not specified in the design description.

For example, in VerilogEval Human v2 [7], all the test cases within the benchmark require generating the design with the name of 'TopModule.' However, some specific test cases do not provide the module name in the design description. Figure 4 presents an example of an undefined module name. The GPT-4o interprets the module name as 'TopLevel,' with a correctly implemented logic and only a different name for the top module. To fix the ambiguity, we can mention in the description that the top-level module should be named as 'TopModule' rather than 'TopLevel.'

Unclear port type refers to the fact that the design description may not clearly specify the output port type (wire or register) of the module. Thus, the LLMs will fail the task if they interpret a different port type. For example, the LLM may try to assign the output port value in a sequential block because it interprets the output port as a register, which is required as a wire in the testbench.

Syntax errors in the code example are present in the description containing a code example as hints. However, the code example contains errors, and LLMs will usually directly copy most of it, including the syntax errors. Figure 5 illustrates the obvious syntax error present in the given template code example. The code example specifies a parameter configuration, but with syntax errors. We can resolve this ambiguity by correcting the code sample's syntax errors.

3.2 Functional behavior ambiguity

Functional behavior ambiguity includes both sequential and combinational logic. Such behavior ambiguity is usually related to the detailed operations, like bit-level operations or conditioned branches. We list three representative situations: (1) *register initialization*, (2) *trigger condition*, and (3) *missing implementation.*

Register initialization ambiguity exists in some designs with the register output port or intermediate registers. For example, in the VerilogEval [7, 13] benchmark, some tasks require the output port of a design to be a register. However, the testbenches require the output to be initialized to fixed values (for e.g., 0), which is not specified in the design description. As a result, LLMs that generate correct logic can still fail due to missing initial assignments in the testbench.

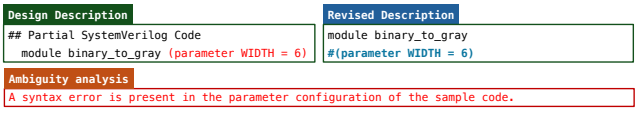

Figure 5: *Code sample with syntax error.* Task ID: binary_to_gray_0001 of *cid002* in CVDP benchmark. The code sample in the design description contains a syntactically incorrect parameter configuration that misleads LLMs into generating the same syntax error.

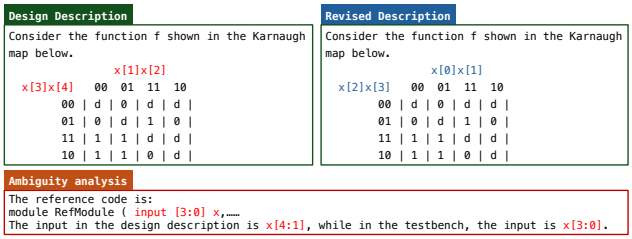

Figure 6: *Diagram ambiguity*. Task ID: HumanEval v2, 116 m2014_q3. The specified input is 'x[4:1]', while in the reference code, the input is 'x[3:0].'

Trigger condition ambiguity refers to the trigger condition of sequential blocks. Some design descriptions do not specify whether a sequential block should be clocked on the clock's posedge with a synchronous reset (reset sampled on the clock) or triggered asynchronously on the reset's posedge (asynchronous reset). LLMs typically produce correct internal logic but miss whether the 'reset' edge should be added to the trigger condition. Usually, LLMs have to choose a trigger condition by default randomly. Thus, the generated designs always fail due to this missing condition specification.

Missing implementation refers to the missing implementation of smaller modules. For example, some design descriptions require LLMs to generate a larger module from predefined smaller modules, which LLMs often interpret as already implemented. However, such pre-defined smaller modules are not provided in the testbench. LLMs are required to actually implement the smaller modules first.

3.3 Diagram ambiguity

Diagram ambiguity always exists in the design diagrams, which represent a classical specification format for RTL design. However, the manually written diagrams are prone to specifying incorrect or ambiguous logic. We present two representative diagrams for illustration: the KMap and FSM formats.

KMap diagram ambiguity refers to the unclear representation of a KMap. There are two kinds of ambiguity problems: 1) indication about the rows and columns (e.g., which input parts correspond to which columns or rows) is unclear or misleading, 2) the values do not match the testbench. For example, in Figure 6, the input in the KMap should be x[0:3], but the testbench requires the input to be x[1:4]. We can resolve this ambiguity by aligning the input x with the testbench.

FSM diagram ambiguity usually exists in the transfer condition of states. For example, the manually written FSM usually misses a few transition conditions, leading to an incomplete FSM specification. The effective way to fix the flaws is to identify the missing state conditions and add them back, following the styles of the original description.

4 QUANTATIVE ANALYSIS

4.1 Flawed Case Identification

In this section, we analyze the identified flawed cases across different benchmarks. Table 3 presents detailed statistics on flawed cases. Human engineers carefully verify all the identified cases. The table consists of two parts: 1) *cases that LLMs constantly fail* and 2) *identified flawed cases*. The number of part 1) obviously correlates with the number of identified flawed cases. Such an observation reveals that *many failure cases of LLMs are due to the case flaw rather than the LLMs' defects.*

It's noticeable that each benchmark contains around 10% of ambiguous test cases. Each benchmark reveals a different pattern of flaw distribution. For example, the flawed cases in VerilogEval [7] benchmarks are mainly related to functional behaviors and diagrams, with no syntax-related flaws. RTLLM [10] flawed cases are not related to diagrams. *cid002* and *cid003* tasks in the CVDP [14] benchmark contain new flawed cases related to the syntax. In the following part, we mainly discuss the detailed features of the three categories of flaws, together with a few examples. RTLLM [10] flawed cases are not related to diagrams. *cid002* and *cid003* tasks in the CVDP [14] benchmark contain new flawed cases related to the syntax.

Re-evaluation on Updated Benchmarks. We re-evaluate the performance of advanced LLMs (e.g., GPT-4o [1] and Claude-3.7) on the updated benchmark. Table 2 presents the updated performance. We can observe that Claude-3.7's performance shows minor variance, while GPT-4o [1] presents an obvious improvement. This observation infers that the performance of GPT-4o is slightly underestimated. With the updated benchmark, we are able to conduct a fairer and more realistic evaluation of different LLMs.

Table 2: Functional accuracy of different models on different benchmarks.

Type	Models	VerilogHuman v1		VerilogHuman v2		VerilogMachine v1		RTLLM v1.1		cid02 in CVDP		cid03 in CVDP	
		Syntax	Function	Syntax	Function	Syntax	Function	Syntax	Function	Syntax	Function	Syntax	Function
Original pass@1 functional accuracy													
Commercial model	GPT-3.5	73.7%	32.7%	53.2%	26.3%	25.0%	16.0%	80%	50%	-	-	-	-
	GPT-4o-mini	89.1%	51.3%	96.2%	51.9%	87.8%	59.6%	88%	56%	-	10.6%	-	18.1%
	GPT-4o	96.8%	67.3%	96.2%	68.8%	96.5%	74.1%	92%	68%	-	19.1%	-	35.1%
	Claude-3.7	98.1%	**80.1%**	98.1%	**76.3%**	92.3%	**76.9%**	90%	**70.0%**	-	**21.3%**	-	**39.0%**
Open model	QwQ-32B	53.2%	44.2%	61.5%	55.1%	74.4%	61.5%	28%	26%	-	8.5%	-	14.3%
	LlaMA-70B	78.8%	47.4%	80.1%	42.9%	78.8%	47.4%	54%	34%	-	10.6%	-	12.3%
	LlaMA-405B	75%	48.7%	91.7%	60.9%	81.4%	62.2%	67.3%	46.8%	-	16.0%	-	27.3%
Functional accuracy after **RTL-BenchMT** fixes flawed cases in benchmark													
Commercial model	GPT-4o	98.7%	71.1% (+3.8%)	98.1%	71.4% (+2.6%)	96.5%	77.6% (+3.5%)	92%	70% (+2.0%)	-	20.2% (+1.1%)	-	33.8% (-1.3%)
	Claude-3.7	99.3%	**79.8%** (-0.3%)	98.1%	**76.9%** (+0.6%)	92.3%	**78.2%** (+1.3%)	90%	**70%** (+0.0%)	-	**22.4%** (+1.1%)	-	**39.0%** (+0.0%)

Benchmarks	VerilogEval Machine v1		VerilogEval Human v1		VerilogEval Human v2		RTLLM v1.1		CVDP cid002		CVDP cid003	
	number	percentage	number	percentage	number	percentage	number	percentage	number	percentage	number	percentage
Cases that LLMs constantly fail												
Total	13/143	9.1%	21/156	13.5%	18/156	11.5%	8/50	16%	11/94	11.7%	28/77	36.4%
Identified flawed cases												
Total	11/143	7.7%	14/156	9.0%	8/156	5.1%	5/50	10.0%	4/94	4.3%	5/77	6.5%
Functional	7/143	6.3%	10/156	6.4%	5/156	3.2%	3/50	6.0%	2/94	2.1%	1/77	1.3%
Syntax	-/-	-	-/-	-	-/-	-	2/50	4.0%	1/94	1.1%	3/77	3.9%
Diagram	4/143	2.8%	4/156	2.6%	3/156	1.9%	-/-	-	1/94	1.1%	1/77	1.3%

Table 3: Statistics on benchmark ambiguity. The upper block reports, for each benchmark, the proportion of tasks that all evaluated LLMs consistently fail, and the lower block shows how many of these are manually confirmed as flawed and how they decompose into functional, syntax, and diagram ambiguities. Across benchmarks, roughly 5–10% of tasks are ambiguous, and a large fraction of consistently failing cases are actually flawed.

4.2 Overfitting Detection

In this section, we discuss the overfitting analysis results based on the performance after design description rewriting. For each original description, we generate four rewritten descriptions for evaluation. Figure 7 illustrates the overall performance comparison results. From the figure, we can see that most of the models show lower performance than original description. Among the models, CodeV [17] shows a significantly lower performance (9.6% decrease). The most advanced commercial models, GPT-4o [1] and DeepSeek [6], both show slight degradation on performance. Interestingly, the open-sourced model LlaMA [15] demonstrates an obvious performance improvement (up to 6.1%). Additionally, we collect the best and worst performance of LLMs on the four rewritten descriptions. DeepSeek [6] and CodeV [17], with more degradation on performance, also show larger performance variance than LlaMA [15] and GPT-4o [1].

Figure 8 illustrates the overfitting analysis matrix for each model. TP items means the evaluation results on both original and updated descriptions are passed. The summation of TP and FP reveals the overall performance on the updated descriptions. We consider FP a strong indicator of LLM generalizability, and TP a good indicator of LLM robustness. The figure further demonstrates that CodeV [17] suffers the most serious overfitting (FN) problem. Here, (FN) means that LLMs pass on the original description while failing the updated descriptions. We consider this item a strong indicator of overfitting. On the other hand, CodeV [17]s LlaMA [15] demonstrates a significantly higher number for FP items. Here, FP means that the LLMs fail on the original description while passing the updated descriptions. Among the fundation models, DeepSeek [6] shows higher overfitting (FN) problems over GPT-4o [1] and LlaMA [15].

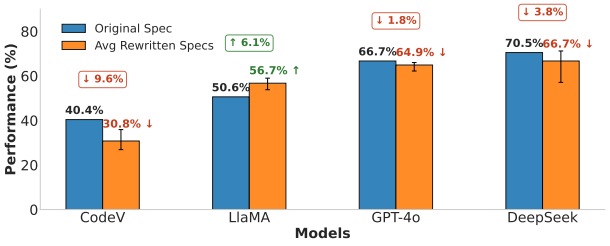

Figure 7: Performance evaluation based on the rewritten design descriptions (Rewritten Specs). CodeV, fine-tuned on the RTL generation tasks, shows more serious performance degradation, indicating more potential overfitting issues.

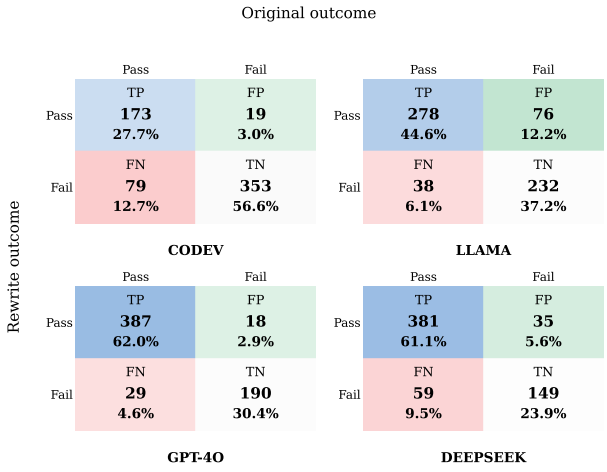

Figure 8: Overfitting analysis matrix. The most advanced open-sourced model, CodeV [17], exhibits the most severe overfitting (FN), while the open-sourced foundation model LLaMA demonstrates the greatest improvement (FP).

5 CONCLUSION

This work introduces RTL-BenchMT, an agentic framework for dynamically maintaining RTL generation benchmarks. RTL-BenchMT targets two core applications: (1) *automatically identifying and revising flawed cases*, and (2) *automatically detecting and updating overfitting cases*. With the assistance of RTL-BenchMT, we conduct an in-depth analysis of ambiguity and overfitting in existing benchmarks and show that a non-trivial fraction of LLM failures originates from benchmark flaws. Finally, we contribute a refined benchmark suite, together with the accompanying agentic maintenance framework, which we will open-source to the community.

6 ACKNOWLEDGEMENT

This work is supported by the Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, GRF 16216825. It was partially conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.

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] Mohammad Akyash, Kimia Azar, and Hadi Kamali. 2025. DecoRTL: A Runtime Decoding Framework for RTL Code Generation with LLMs. *arXiv preprint arXiv:2507.02226* (2025).

[3] Nurit Cohen-Inger, Yehonatan Elisha, Bracha Shapira, Lior Rokach, and Seffi Cohen. 2025. Forget What You Know about LLMs Evaluations–LLMs are Like a Chameleon. *arXiv preprint arXiv:2502.07445* (2025).

[4] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, et al. 2024. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *arXiv preprint arXiv:2407.16237* (2024).

[5] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2024. VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool. *arXiv preprint arXiv:2408.08927* (2024).

[6] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[7] 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.

[8] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE.

[9] Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. 2024. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[10] 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.

[11] Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. 2024. VerilogReader: LLM-Aided Hardware Test Generation. *arXiv preprint arXiv:2406.04373* (2024).

[12] 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).

[13] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2024. Revisiting VerilogEval: Newer LLMs, In-Context Learning, and Specification-to-RTL Tasks. *arXiv preprint arXiv:2408.11053* (2024).

[14] Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. 2025. Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification. *arXiv preprint arXiv:2506.14074* (2025).

[15] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[16] Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. 2024. RTLRewriter: Methodologies for Large Models aided RTL Code Optimization. *arXiv preprint arXiv:2409.11414* (2024).

[17] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, et al. 2024. Codev: Empowering llms for verilog generation through multi-level summarization. *arXiv preprint arXiv:2407.10424* (2024).

[18] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2024. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. *arXiv preprint arXiv:2412.07822* (2024).