AutoVeriFix +：基于轨迹感知的高正确性寄存器传输级生成因果修复与语义冗余剪枝

Yan Tan，*IEEE* 学生会员，Xiangchen Meng，*IEEE* 学生会员，Zijun Jiang，*IEEE* 学生会员，以及 Yangdi Lyu，*IEEE* 会员

arXiv:2603.11489v1 [cs.PL] 12 Mar 2026

摘要——**大语言模型（LLM）在生成 Python 和 C++ 等高级编程语言的软件代码方面展现出令人瞩目的能力。然而，由于高质量训练数据稀缺，将其应用于 Verilog 等硬件描述语言仍面临挑战。目前，使用大语言模型生成 Verilog 代码的方法往往侧重于语法正确性，因此生成的代码存在功能错误。为应对这些挑战，我们提出 AutoVeriFix+，一种新颖的三阶段框架，将高级语义推理与状态空间探索相结合，以提升功能正确性和设计效率。第一阶段使用大语言模型生成高级 Python 参考模型，用于定义预期的电路行为。第二阶段由另一个大语言模型生成初始 Verilog RTL 候选代码，并迭代修复语法错误。第三阶段，我们引入一个具体符号测试引擎，对深层时序逻辑进行测试并识别边界情况漏洞。借助周期精确的执行轨迹和内部寄存器快照，AutoVeriFix+ 为大语言模型提供解决复杂状态转换错误所需的因果上下文。此外，该框架还会生成覆盖率报告，以识别功能冗余分支，使大语言模型能够通过语义剪枝进行面积优化。实验结果表明，AutoVeriFix+在严格的基准测试中实现了超过 80% 的功能正确性，在 VerilogEval-machine 数据集上达到 90.2% 的 pass@10 分数。此外，通过轨迹感知优化，该方法在各项基准测试中平均消除了 25% 的冗余逻辑。**

索引词—**大语言模型生成的Verilog、功能正确性、自动化测试**

I. 引言

LARGE 语言模型（大语言模型）通过自动代码生成显著提升生产力，彻底改变了软件开发。通过将自然语言描述转化为高质量的软件代码，大语言模型使开发者能够专注于更高层次的设计与问题求解，而非日常的编码任务。这种生产力提升在很大程度上归功于大语言模型在海量开源仓库上的广泛训练，尤其是针对Python和C++ [1]、 [2]等广泛使用的语言。因此，大语言模型目前在软件领域不仅达到了很高的语法准确性，而且具备稳健的功能正确性。

在这些成果的基础上，研究人员开始探索将大语言模型用于生成硬件描述语言（HDL）。早期尝试在生成基本的寄存器传输级（RTL）组件方面已展现出潜力。然而，软件与硬件代码生成之间仍存在显著的性能差距 [9]。尽管硬件描述语言在语法上与软件语言相似，但它们要求精确地建模并发执行、时序约束以及硬件特有的惯用写法。此外，开源 Verilog 数据集相对匮乏，限制了监督微调在硬件描述语言任务上的效果。鉴于电子设计自动化（EDA）工具运行时间漫长且制造周期不可逆，硬件错误造成的代价呈指数级高于软件错误，因此，大语言模型生成的寄存器传输级代码在集成之前必须经过严格的功能验证。

本研究得到国家自然科学基金委员会资助，项目编号#62402412。（通信作者：*Yangdi Lyu*。）Yan Tan、Xiangchen Meng、Zijun Jiang和Yangdi Lyu就职于香港科技大学（广州）微电子学域，中国广州511453（电子邮箱：ytan910@connect.hkust-gz.edu.cn；xmeng027@connect.hkust-gz.edu.cn；zjiang438@connect.hkust-gz.edu.cn；yangdilyu@hkust-gz.edu.cn；）。

这种性能差异可能源于抽象层的根本差异。Python 等软件语言运行在功能抽象层级，与大语言模型的模式识别能力十分契合。相比之下，Verilog 等硬件描述语言需要显式指定低层电路行为，尤其是复杂的时序逻辑和非阻塞赋值。为解决这一问题，近期研究聚焦于硬件专用数据增强 [3]–[6], [10]。例如，RTLCoder [7] 利用自动化设计数据增强，使自然语言描述与 Verilog 语义相匹配。为进一步减少错误，研究者提出了自反思框架 [8], [11]–[13] ，用于识别语法错误并迭代改进生成的代码。然而，现有方法主要通过模式匹配和基于模板的方法进行语法修正，往往无法解决多分支逻辑或复杂有限状态机 (FSM) 中深层的功能差异。

为弥合这一差距，我们提出 AutoVeriFix+，这是一个三阶段框架，集成了大语言模型辅助自动化测试机制与轨迹感知优化，以确保寄存器传输级设计的功能完整性。

- **阶段 1：Python 辅助参考建模**。我们利用现有大语言模型在理解和生成 Python 代码方面的强大能力，根据自然语言规范生成高级参考模型。该模型充当功能黄金参考模型。尽管这些模型能够捕捉预期行为，但由于 MyHDL [14] 和 SODA [15]等高级综合（HLS）工具所支持的可综合子集存在限制，它们无法直接综合为硬件。

- **阶段 2：迭代式 RTL 优化**。在第二阶段，另一个大语言模型会生成初始的 Verilog RTL 实现，随后该实现将进入自动语法

验证循环，以解决即时出现的编译错误和代码检查器错误。确保语法无误后，我们会通过整合详细的调试信息来优化 RTL。我们不再依赖简单的输入/输出不匹配消息，而是利用后续阶段提供的因果反馈，准确指出出现故障的内部逻辑，从而确保最终代码既可编译，又在功能上符合规范。

- **第三阶段：跟踪感知优化与剪枝**。为突破黑盒测试的局限，我们引入了一个全新的第三阶段，利用具体符号测试系统地探索深层逻辑中的边界情况，并最大限度地提高分支覆盖率。通过自动对寄存器传输级进行插桩并解析其控制流图（CFG），我们将内部执行轨迹与 Python 参考模型对齐。这种白盒测试使我们能够检测出隐蔽的状态机错误并消除冗余逻辑，从而显著提升设计的功能可靠性和硬件效率。

我们的主要贡献总结如下。

1) 我们提出了一个框架，利用以软件为中心的大语言模型（Python）的高级推理能力，为尚不成熟的硬件领域（Verilog）提供功能预言机，从而实现黄金模型的自动生成。2) 我们引入了一种自动插桩机制，用于监控运行时内部状态。这种白盒方法能够发现传统黑盒测试无法检测到的深层功能错误。3) 我们开发了一种针对性反馈机制，向大语言模型提供周期精确跟踪和覆盖率报告，从而实现精确的错误修复并移除冗余代码。4) 实验结果表明，AutoVeriFix+ 显著优于最先进的方法，在严格的基准测试中实现了超过 80% 的功能正确性，同时优化了实现效率。

本文扩展了我们发表于 [16] 的初步研究。当前工作 AutoVeriFix+ 在多个关键方面拓展了原有的两阶段框架。具体而言，我们引入了阶段 3：跟踪感知优化与剪枝循环，通过监测仿真路径和内部状态转换，为修复错误提供因果上下文。此外，我们整合了 ConcolicTesting 引擎，以系统地探索边界情况，并对死代码剪枝机制进行了细致评估。最后，我们大幅扩展了实验部分，增加了更多实验和消融研究。

本文其余部分组织如下。第二节回顾相关工作并提供必要的背景。第三节介绍采用轨迹感知优化的 AutoVeriFix+ 框架架构。第四节给出实验结果，第五节对本文进行总结。

II. 背景与相关工作

A. 大语言模型的进展

近年来，大语言模型取得了重大突破，在语言理解与生成方面展现出卓越能力 [1], [2], [17]。当代大语言模型，如 ChatGPT [18]和 LLaMA [19]，拥有数十亿参数，并在来自不同领域的大规模数据集上接受训练。

大语言模型最广泛应用之一是自动代码生成。OpenAI 的 Codex [20] 和 DeepMind 的 AlphaCode [21] 等模型已展现出将自然语言描述转换为可执行代码的能力。这些模型基于大量编程数据集进行训练，能够执行代码补全、调试以及解决复杂编程难题等任务。通过自动化此类任务，大语言模型正在革新软件开发，提高生产力，并降低编程入门门槛。然而，尽管大语言模型在 Python、C++ 和 Java 等软件编程语言方面表现出色，将其能力扩展到 Verilog 等硬件描述语言仍然面临挑战 [17], [22], [23]。与软件不同，硬件设计要求对时序、并发和信号行为进行精确规定。此外，高质量且公开可用的寄存器传输级数据集有限，限制了通用大语言模型有效掌握硬件专属任务的能力 [9], [10], [22]。为弥合这一差距，近期研究主要沿着两个方向展开：通过数据中心方法提升数据质量，以及借助反馈驱动机制确保正确性。

B. 数据中心方法

为解决数据稀缺问题，研究人员专注于整理硬件专用数据集，并采用数据增强技术来提升模型性能。Thakur 等人 [24] 构建了一个无监督 Verilog 数据集，数据来源于 GitHub 代码库和 Verilog 教材，并用其对 CodeGen-16B [25] 进行微调。然而，由于该数据集中没有人工标注，在此数据集上微调的模型性能仍逊于 GPT-3.5 等商业工具。相比之下，VerilogEval [6] 通过仔细收集、筛选和完善来自开源硬件代码库的 Verilog 文件和设计片段，构建了一个有监督数据集。这些严谨的步骤确保其中包含具有代表性的寄存器传输级代码，使大语言模型能够更有效地学习硬件设计模式。在该有监督数据集上进行微调后，VerilogEval 成为首个性能达到 GPT-3.5 水平的非商业模型。

数据增强技术已被用于进一步提升数据集的多样性。RTLCoder [7] 提出了一种框架，可生成与 Verilog 代码语义相符的自然语言描述，从而创建上下文配对，丰富训练过程。类似地，OriGen [8]采用代码到代码的增强策略，生成经过扩充的数据集，其中包含原始设计和错误修正版

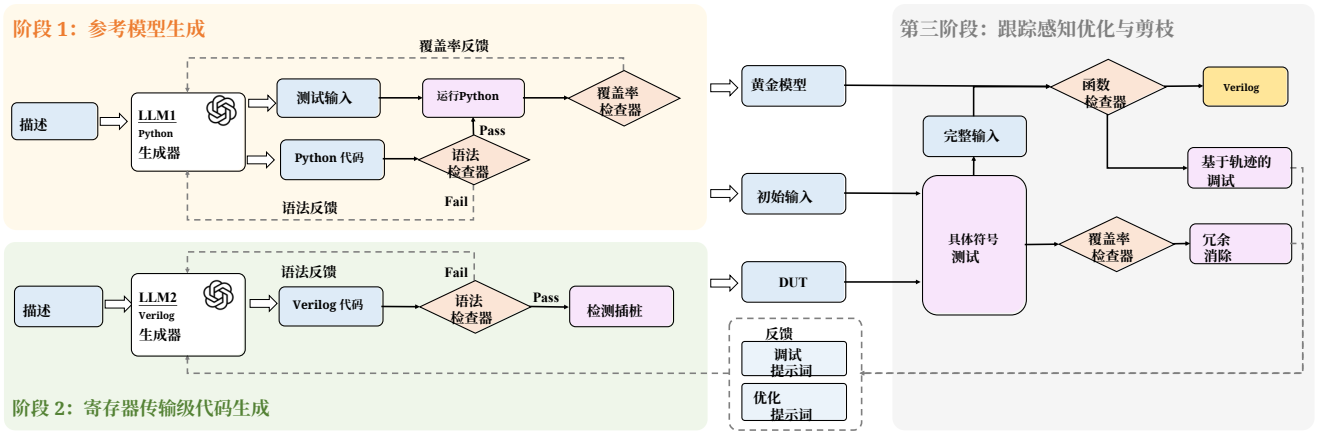


图 1：AutoVeriFix+ 框架的三阶段概览。阶段 1 构建高层级 Python 黄金模型和初始测试向量。阶段 2 生成初始 Verilog RTL，并迭代修复语法和功能错误。阶段 3 采用具体测试和轨迹感知分析，以支持深路径功能测试和冗余消除。

的寄存器传输级代码。HLSDebugger [26]，通过在包含 300,000 个样本的大规模数据集上训练，采用编码器—解码器结构，统一实现逻辑错误的定位、类型预测和自动修复。

尽管这些方法增强了大语言模型对 RTL 设计原则的理解，但仍不足以确保语法和功能正确性。具体而言，它们主要关注“生成”能力，而非满足硬件设计所要求的严格验证标准。

C. 反馈驱动机制

为提高生成代码的正确性，反馈驱动机制已被集成到大语言模型生成 RTL 代码的代码生成工作流中。最常见的方法是利用 EDA 工具日志修复语法错误。AutoChip [11] 和 OriGen [8] 率先将编译器错误消息用作提示词，促使模型进行自我纠正，使大语言模型能够迭代地解决语法错误和代码检查器警告。RTLFixer [12] 则进一步优化了这一流程：利用检索增强生成（RAG）检索与特定编译错误相关的调试指导。

除了简单的编译之外，一些框架还会引入辅助指标来指导模型。RTLCoder [7] 引入了一种代码质量评分反馈机制，根据语法正确性和与参考代码的相似度对生成的代码进行评估。随后，评分结果将作为反馈用于提升模型的生成能力。类似地，一些研究将物理实现反馈，特别是功耗、性能和面积（PPA）指标，整合进来，以引导大语言模型生成更具硬件效率的实现 [10], [27]。

尽管这些先前的研究为自动代码生成奠定了基础，但我们发现，这些方法生成的 Verilog 代码存在明显的功能错误，尤其是在多分支逻辑和复杂状态转换等常见硬件设计任务中。不同于早期仅依赖编译器语法反馈的方法，我们使用参考模型生成高质量的测试平台并通过动态仿真评估和改进代码的功能正确性。

III. 方法

A. 概述

为全面提升功能正确性和实现效率，我们提出了 AutoVeriFix+，这是一种三阶段反馈框架，旨在自动检测、纠正和优化大语言模型生成的 Verilog 代码。AutoVeriFix+ 的核心理念是利用大语言模型在高级语言（Python）方面的高功能准确性，监督更容易出错的硬件描述语言（Verilog）生成过程。如第四节 B所示，尽管大语言模型在 Python 中的功能正确性超过 95%，但由于高质量 RTL 训练数据稀缺，其 Verilog 输出的功能正确性往往低于 50%。理想情况下，将生成的 Python 代码用作近乎完美的参考模型，可以使大语言模型生成的 Verilog 代码达到与参考模型相当的功能正确性水平。如图 1所示，AutoVeriFix+ 将大语言模型与符号与具体相结合测试引擎集成于一个闭环框架中，以实现 Verilog 代码生成、验证和优化的自动化。

B. 阶段 *1*：*Python* 辅助参考模型 g

在第一阶段，Python 生成器（LLM1）将自然语言硬件规格说明转换为高级 Python 参考模型，以及一组相应的初始测试向量。此流程建立了一个功能黄金模型，作为后续所有验证周期的真实基准。与传统的高层次综合流程不同，后者将高级 Python 或 C++ 转换为 Verilog；我们的框架仅将 Python 代码用作功能预言机。

为确保该预言机的可靠性，生成的 Python 代码需经过双重验证：

- 语法与接口验证：生成的代码会经过语法检查器处理，以确保其可执行且符合接口要求。如果出现编译错误或

检测到接口不匹配，框架便会启动语法反馈循环来修复这些问题。

- 覆盖率分析：运行参考模型以评估逻辑分支覆盖率。如果覆盖率不足，反馈循环会触发 LLM1 扩展测试输入，为后续符号具体探索奠定稳健的基础。

*1) Python*生成与语法验证：首先向 LLM1 提供硬件设计的结构化功能描述。随后提示该模型生成两项内容：(1) 体现设计逻辑的功能性 Python 参考模型；(2) 一组用于验证的初始测试输入。

生成的 Python 代码会立即交由语法检查器处理。除了验证代码是否可执行，该检查器还会检查测试输入与模型端口定义之间的一致性。如果发现不匹配或语法错误，框架就会启动迭代式语法反馈循环。该循环会将具体的错误日志和诊断消息反馈给 LLM1，促使模型修正结构或逻辑违规，直至获得稳定且可执行的参考实现。

在图 2 所示的简单示例中，我们首先向 LLM1 提供状态机的结构化描述（如图 2 顶部所示）。然后，LLM1 生成一个用于描述该状态机行为的 Python 类（如图 2 中部所示），以及一组初始输入序列 [1,1,0]（如图 2 底部所示）。Python 类和测试输入生成后，会进行语法检查。如果检测到语法错误，框架会通过错误消息提供反馈，以便大语言模型根据需要调整其输出。

在图 2 所示的示例中，生成的参考模型有效地捕捉了硬件的功能行为。这一成功归功于大语言模型在海量 Python 数据集上接受了广泛训练，使其能够生成高度准确、易读的 Python 实现，并与提供的功能描述相符。

2）覆盖率驱动的测试输入细化：Python 代码通过语法验证后，框架会使用初始输入执行模型。覆盖率检查器会分析执行轨迹，以评估逻辑分支的覆盖程度。如果分支覆盖率低于预设阈值（例如 85%），检查器就会生成一条覆盖率消息，详细说明尚未覆盖的区域。这些反馈将引导 LLM1 生成更有针对性的测试向量。

需要注意的是，该阈值的定义较为宽松，主要用于优化阶段 3。目标是为符号与具体相结合测试引擎提供高质量的初始种子。虽然即使初始输入质量较低，具体符号引擎也有望实现较高覆盖率，但在阶段 1 提供高覆盖率的种子集，能够显著减少阶段 3 探索深层逻辑状态所需的计算时间和求解器迭代次数。

例如，如图 3 所示，初始覆盖率为 60%，还有 4 个未覆盖的分支。框架会识别这些具体缺口，促使大语言模型生成经过优化的向量

Prompt for Reference Model Generation

You are an expert in digital design and Python programming. The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include a positive edge triggered asynchronous reset that resets the FSM to state A.

// state | next state in=0, next state in=1 | output
// A | A, B | 0
// B | C, B | 0
// C | A, D | 0
// D | C, B | 1

Generate a Python class implementation based on the provided requirements and the corresponding test inputs.

Python Code Generation

```python
class MooreStateMachine:
    STATE_A = "A", STATE_B = "B", STATE_C = "C",STATE_D = "D"
    def __init__(self):
        self.state = self.STATE_A
        self.output = 0

    def reset(self):
        self.state = self.STATE_A
        self.output = 0

    def clock_edge(self, input_signal):
        if self.state == self.STATE_A:
            if input_signal == 0:
                self.state = self.STATE_A
            else:
                self.state = self.STATE_B
                self.output = 0 # Output for state A
        elif self.state == self.STATE_B:
            if input_signal == 0:
                self.state = self.STATE_C
            else:
                self.state = self.STATE_B
                self.output = 0 # Output for state B
        elif self.state == self.STATE_C:
            if input_signal == 0:
                self.state = self.STATE_A
            else:
                self.state = self.STATE_D
                self.output = 0 # Output for state C
        elif self.state == self.STATE_D:
            if input_signal == 0:
                self.state = self.STATE_C
            else:
                self.state = self.STATE_B
                self.output = 1 # Output for state D
```

Test Input Generation

```python
if __name__ == "__main__":
    fsm = MooreStateMachine()
    fsm.reset()
    print(f"After Reset: {fsm.state}, Output: {fsm.output}")
    # Simulate clock edges with input signals
    input_sequence = [1, 1, 0]
    for i, inp in enumerate(input_sequence):
        fsm.clock_edge(inp)
        print(f"Clock {i + 1}: Input={inp}, Output={fsm.output}")
```

图 2：我们的框架向大语言模型提供硬件设计的结构化功能描述，由大语言模型生成 Python 类和初始测试序列。

（例如 [1,1,0,1,0,0,1]）。这一迭代优化过程会持续进行，直至系统输出最终的黄金模型和高覆盖率输入集，从而确保后续的符号具体执行驱动的第三阶段能够高效运行。

C. 阶段 *2*：迭代式 *RTL* 优化

阶段 2 侧重于将功能规格说明转换为可综合的寄存器传输级（RTL）实现——

Provide feedback with line coverage and uncovered lines

Simulation Result:
After Reset: A, Output: 0
Clock 1: Input=1,Output=0
Clock 2: Input=1,Output=0
Clock 3: Input=0,Output=0

Coverage:
The coverage is 60%, with 12 missing lines and 4 partially covered branches.
Missing lines are line 13 (if input_signal == 0 in STATE_A), line 21 (else branch in STATE_B), and lines 25-35 (all branches in STATE_C and STATE_D).
Partially covered branches are line 12 (if input_signal == 0 in STATE_A), line 18 (if input_signal == 0 in STATE_B), and line 21 (else branch in STATE_B).

LLM refines the test inputs based on feedback to improve coverage

```
if __name__ == "__main__":
    fsm = MooreStateMachine()
    fsm.reset()
    print(f"After Reset: {fsm.state}, Output: {fsm.output}")
    # Simulate clock edges with input signals
    input_sequence = [1, 1, 0, 1, 0, 0, 1]
    for i, inp in enumerate(input_sequence):
        fsm.clock_edge(inp)
        print(f"Clock {i + 1}: Input={inp}, Output={fsm.output}")
```

图 3：大语言模型根据覆盖率反馈优化测试输入

mentation。该阶段在闭环中运行，处理初始生成、语法修复和自动化插桩，为阶段 3 中执行的深度功能分析准备设计。

现有基于大语言模型的硬件生成流程通常依赖黑盒功能验证，该验证只检查输入/输出端口对应关系。这种方法对于生产级硬件存在显著局限。首先，当内部状态转换不正确，但主要输出在有限测试集上恰好与预期值一致时，可能会出现不易察觉的错误。其次，大语言模型经常会臆造冗余逻辑或死代码；这些内容虽然能通过功能仿真，却会使设计的面积、功耗和时序指标变差。为应对这些挑战，阶段 2 会根据阶段 3 提供的反馈进行后续优化。

1）Verilog 生成与语法验证：Verilog生成器（LLM2）根据结构化硬件描述生成初始 Verilog 代码。除了阶段 1 提供的详细行为描述外，面向 LLM2 的提示词还包含全面的设计规范，例如模块定义和输入/输出端口配置。随后，LLM2 会生成完整的 Verilog实现，以满足指定的设计要求，如图 4 所示。这体现了模型将自然语言规格说明转换为结构化寄存器传输级设计的能力。

与阶段 1 类似，生成的 Verilog 会经过严格的语法检查，以确保能够正确编译。如果硬件编译器（例如 Icarus Verilog 或 Verilator）检测到结构违规，例如未声明信号、缺少分号或关键字位置错误，框架会自动生成语法调试提示（图 5）。该提示会将原始错误代码与编译器的诊断消息明确配对，其中包括错误类型、具体行号以及其他诊断元数据。通过将这些上下文反复反馈给 LLM2，框架促使模型分析失败原因并修正语法违规。该迭代修复周期会持续进行，直到设计成功编译，

Prompt for Verilog Generation

The following is the state transition table for a Moore state machine with one input, one output, and four states. Implement this state machine. Include a positive edge triggered asynchronous reset that resets the FSM to state A.

// state | next state in=0, next state in=1 | output
// A | A, B | 0
// B | C, B | 0
// C | A, D | 0
// D | C, B | 1

module top_module (input clk, input in, input areset, output out);

Verilog Code Generation

```
module top_module (input clk, input in, input areset, output out);
    typedef enum logic [1:0] {
        A = 2'b00,
        B = 2'b01,
        C = 2'b10,
        D = 2'b11
    } state_t;
    state_t current_state, next_state;
    always_comb begin
        case (current_state)
            A: next_state = (in) ? B : A;
            B: next_state = (in) ? B : C;
            C: next_state = (in) ? D : A;
            D: next_state = (in) ? B : C;
            default: next_state = A; // Default state
        endcase
    end
    always_ff @(posedge clk or posedge areset) begin
        if (areset)
            current_state <= A; // Reset to state A
        else
            current_state <= next_state;
    end
    always_comb begin
        case (current_state)
            A, B, C: out = 1'b0;
            D: out = 1'b1;
            default: out = 1'b0;
        endcase
    end
endmodule
```

图 4：Verilog 生成提示工程示例。

语法调试提示

你是一名代码专家。你的任务是根据编译器错误消息调试所提供代码中的语法错误。请修复错误并返回更正后的代码。

- [OriginalCode]<在此处插入 Python 或 Verilog 代码。>
- [编译器错误消息]<在此列出编译器错误消息。>

>

图 5：标准化语法调试提示模板

从而确保 Verilog 代码语法正确，可用于后续插桩。

2) RTL 代码插桩：为实现精确调试并追踪硬件执行逻辑，我们采用自动化插桩流程，将原始 Verilog 转换为待测设计（DUT），随后将其传递至阶段 3。此流程旨在捕获仿真期间的实际执行路径和瞬时寄存器状态，从而检测黑盒测试无法发现的深层功能偏差。

插桩工作流如下：

- 该框架将大语言模型生成的 Verilog 解析为

抽象语法树（AST），以构建控制流图（CFG）。

• CFG 中的每个分支都会被分配一个唯一标识符（*Bi*）。随后，框架会在每个基本块的末尾注入监控逻辑（例如显示语句）。

•最后，框架会识别所有寄存器，并插入显示语句，以便在每个时钟周期记录每个寄存器的状态。

清单 1：示例 1

```
模块 top(
    输入    wire          时钟,
    输入    wire          reset,
    输入    导线  [7: 0]in,
    输出寄存器              ou
                           t
);

    reg [7: 0] 计数器;

    始终块 @(上升沿 时钟) 开始
        if (复位)  开始
            计数器 <= 8'h00;
            out <= 1'b0;
            $display("B_1");
        end
        el se begin
            case语句(in)
                8'h00 : 开始
                    counter <=counter - 1;
                    $显示("B 2"); _
                end
                8'h02: begin
                    计数器 <=计数器 + 1;
                    $display("B_3");
                end
                8'hFF: begin
                    counter <= 8'h00;
                    $display("B_4");
                end
                default:开始
                    $display("B_5");
                end
            结束case语句
            if(counter == 8'h01) 开始
                out <= 1'b1; $display("B_6");
            end
            否则 begin
                输出 <= 1'b0; $display("B_7");
            end
        end
    end

    //显示寄存器
    始终块 @(上升沿 clock)begin
        $display("R counter = %d", counter);
        $display("R out = %d", out);
    end
endmodule
```

清单 1 展示了插入监测语句后的检测插桩设计示例。在仿真过程中，插入的代码会生成运行时执行轨迹和定期状态快照。通过将这条具体执行路径与 Python 参考模型的高级逻辑流程进行比较，该框架能够精确定位硬件逻辑偏离预期行为的周期和位置。

3) 优化循环：然而，语法正确并不意味着功能正确，也不保证实现效率。完成检测插桩后，被测设计进入阶段 3，进行严格的功能评估与优化。如框架概览（图 1）所示，阶段 2 与阶段 3 之间的交互构成了一个闭环迭代优化过程。当下游分析检测到功能差异或发现不可达逻辑时，就会生成高层语义反馈，包括基于跟踪的调试提示（图 8）和冗余消除提示（图 9）。正如第 III-D 节深入讨论的那样，这些信号为大语言模型提供了有关内部状态故障和实现效率低下问题的详细因果上下文。通过迭代处理这些反馈，LLM2 不断优化 Verilog 实现，逐步修复逻辑错误并剪除死代码，直到最终的寄存器传输级设计在功能上与黄金模型等效，同时也针对硬件实现进行了优化。

D. 第三阶段：跟踪感知优化与剪枝

虽然阶段 2 生成的 Verilog 代码在语法上有效，但这并不能保证功能正确性或硬件效率。大语言模型生成的硬件描述往往存在幻觉，即由于细微的时序问题或状态转换差异，逻辑偏离了规格说明。此外，大语言模型经常会生成冗余逻辑，这些逻辑虽然能通过基础仿真，却会导致过大的面积开销。

为应对这些挑战，我们采用基于协同符号执行的验证机制进行轨迹感知优化与剪枝。此阶段超越了传统的黑盒验证（仅监控输入/输出端口），通过系统地探索状态空间，并使 Verilog 代码的内部执行轨迹与 Python 参考模型保持一致。借助阶段 2 中插桩后的待测设计，我们的框架能够检测出内部硬件状态偏离黄金模型的确切时钟周期和路径轨迹。这为 LLM2 提供了细致的因果上下文，以便进行错误修复，而非泛泛的输入/输出不匹配消息。此外，通过识别功能上不可达的代码块，我们可以剪除冗余逻辑，在不影响功能的前提下优化硬件面积。

如图 1 右侧所示，阶段 3 接收 Python 参考模型、阶段 1 生成的初始高质量输入测试，以及阶段 2 生成的被测设计。尽管阶段 1 的输入向量提供了可靠的功能基线，但它们往往无法覆盖硬件实现中的深层时序逻辑或边界情况。因此，阶段 3 基于具体符号测试引擎构建，将初始输入扩展为覆盖率更高的测试集。随后，函数检查器在扩展输入集下将被测设计的输出与黄金模型的输出进行比较。检测到差异时，基于轨迹的调试模块会执行因果轨迹分析并生成相应的调试消息。与此同时，覆盖率检查器会评估逻辑分支的可达性，标记不可达代码，并生成覆盖率消息。之后，调试消息和覆盖率消息都会反馈给阶段 2 的大语言模型，形成闭环优化过程，逐步引导模型

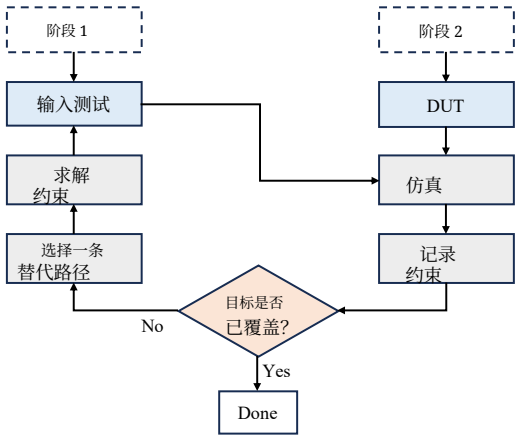


图 6：符号与具体相结合测试引擎的工作流程

优化 Verilog 代码，直至实现功能正确性，同时消除冗余逻辑。

1）具体符号测试：阶段 3 的核心引擎是一个具体符号测试框架（结合具体仿真与符号执行），改编自 [28]。该引擎通过系统地变异仿真路径中的分支，来触发深层逻辑，从而解决验证瓶颈；这些逻辑可能会被大规模随机测试忽略。虽然具体符号测试无法保证 100% 的功能正确性，但实验结果表明，我们的框架能够有效纠正大语言模型生成的 Verilog 代码中的大量错误。

图 6 所示的工作流程遵循路径探索策略以扩展仿真边界。该过程首先使用来自阶段 1 的高质量输入测试对被测设计进行仿真，使具体符号引擎能够比使用随机输入更快地实现覆盖率收敛。利用在阶段 2 注入的监控逻辑，引擎记录实时仿真路径和周期精确状态快照。如果仿真期间覆盖率得到提升，则将该输入向量加入输入集。否则，引擎沿当前路径启发式地选择一个分支节点进行变异。随后，它构建一组符号约束，表示将执行流强制进入所选分支所需的取反路径条件。这些约束被送入 SMT 求解器。如果它们可满足，求解器将返回一个新的输入向量，该向量保证驱动后续仿真迭代朝向所选分支。通过迭代更新约束并生成定向输入，该框架逐步深入深层逻辑状态。这一过程确保测试平台从仅仅检查高层逻辑演进为覆盖 Verilog实现中尽可能多的分支。

2）具体符号测试中的分支变异：为了直观展示状态空间遍历的底层机制，图 7 展示了基于清单 1 中所提供的寄存器传输级实现进行的一次分支变异迭代。

仿真与约束提取：该过程从阶段 1 生成的初始输入序列开始（例如，$\{in = 8'h00\}$，适用于全部三个周期），并以初始化 $\{reset = 1\}$ 开始。如跟踪记录所示，这个默认序列将执行限制在一条重复路径上，该路径经过分支 [B 1、B 2、B 7、B 2、B 7]。

分支变异与约束求解：为提高覆盖率并扩展输入集，具体符号引擎会识别尚未探索的分支，例如将分支 B 2 变异为分支 B 3。如图 7 所示，引擎会提取从周期 0 到目标汇合点的累积路径约束，并附加满足分支 B 3 所需的逻辑谓词。系统通过 SMT 求解器求解这些约束，生成新的定向输入向量 $\{reset = 0, in = 8'h02\}$。在随后的仿真中，这个变异后的向量成功触发了分支 B 3。此外，随着分支 B 3 中的计数器递增，分支 B 6 的前置条件随后也得到满足，从而立即提升了功能覆盖率。

这一个变异周期展示了该框架解决多周期序列问题的能力，这些序列对于到达深层逻辑边界至关重要。AutoVeriFix+ 利用这一能力生成一个全面的测试套件，即完整输入集，以最大限度地提高生成的寄存器传输级的功能可观测性。

3) 基于跟踪的功能调试：函数检查器使用完整输入集并行执行被测设计和 Python 黄金模型。在传统的差分测试范式中，系统只能检测到外部输出不匹配（$Output_{DUT} \neq Output_{Golden}$）。然而，这种黑盒反馈往往不足以诊断复杂的硬件逻辑错误。

为弥补这一可观测性差距，我们的框架采用基于跟踪的反馈机制。我们利用跟踪仿真路径并监控所有内部寄存器状态所生成的执行日志，为大语言模型提供深入的因果上下文。

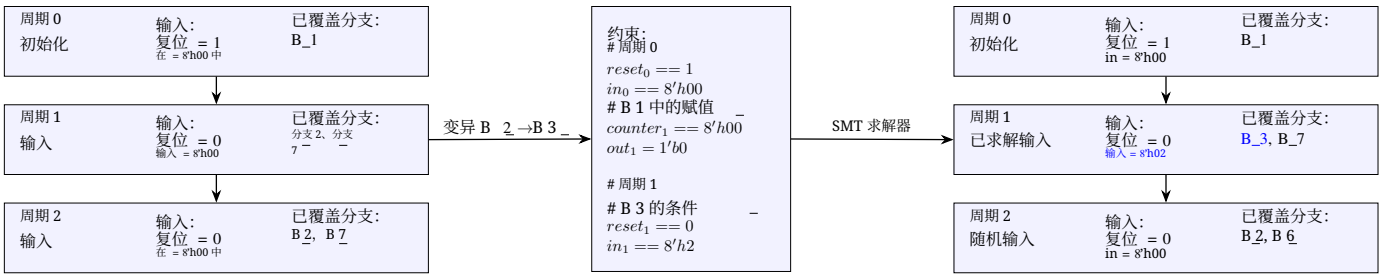


图 7：具体符号测试中分支变异的示例。具体符号引擎会生成变异输入，引导仿真遍历未覆盖分支。

基于执行轨迹的调试提示

你是一位 Verilog 验证工程师专家。你的目标是修复所提供代码中的功能错误，依据是执行轨迹分析。具体来说，重点关注寄存器状态与执行路径之间的因果关系。修复 Verilog 代码，使其符合预期行为。

- [原始代码]
  <请在此处插入 Verilog 代码。>
- [验证失败报告]
  <测试用例输入：>
  <预期行为：>
  <观测到的不匹配：>
- [跟踪反馈]
  以下跟踪信息显示了准确的执行路径（由 Exec: B_i 标记）以及失败时钟周期的寄存器状态（由 R 标记）。

图 8：用于修复功能错误的基于执行轨迹的调试提示。

具体而言，我们构建了一个“基于执行轨迹的调试提示”，由三个主要部分组成，如图 8 所示。

- 原始代码：当前正在测试的 Verilog 实现版本。
- 验证失败报告：一份完整报告，其中包括触发失败的具体测试用例、GoldenModel 的预期输出，以及从被测设计观察到的错误输出。
- 跟踪反馈：逐周期执行轨迹，每个周期由两部分组成。– 执行路径：每个周期触发的具体赋值标识符（B i）。– 内部状态快照：每个时钟沿的瞬时寄存器状态（R）。 –

这种结构化反馈促使大语言模型执行因果轨迹分析，检查内部寄存器值与由此产生的错误执行路径之间的关系。通过使模型能够逐周期跟踪信号传播和状态转换，该框架帮助大语言模型精确定位硬件状态偏离功能预期的具体时间点和结构位置。

大语言模型生成经过优化的 Verilog 候选版本后，更新后的代码会重新整合到具体符号引擎中，以重新生成完整输入集。这一步至关重要：针对某个逻辑分支的修复可能会破坏其他分支，或产生新的不可达状态。随后对设计重新进行仿真，并根据黄金模型重新评估输出。这个基于跟踪的功能调试迭代循环会持续进行，直到在完整输入测试套件下观察到的硬件行为与参考模型完全一致。

4) 冗余消除与优化：除了因果调试之外，阶段 3 还尝试通过覆盖率驱动优化来完善生成的寄存器传输级设计。Concolictesting 完成后，它根据求解器的结果将逻辑分支分为两类：可达分支和潜在不可达分支。为了优化硬件面积，我们设计了“冗余消除”提示词，如图 9 所示。大语言模型充当语义逻辑裁剪的优化专家。它利用可达性报告消除死

冗余消除提示

你是一名 Verilog 优化专家。你的目标是在不改变有效功能的前提下，尽可能缩小代码面积并消除冗余逻辑。

- [原始代码]
  <在此处插入 Verilog 代码。>
- [覆盖率报告]
  具体符号引擎已穷尽探索该设计。以下是状态每个逻辑分支的（以 B_i 标记）
  可达分支：<B_i >已用 <输入向量>覆盖
  不可达分支：<B_i> 未覆盖。

图 9：基于具体符号覆盖率报告移除死代码的冗余消除提示

代码，同时不改变可达分支的功能，从而确保最终设计既正确又高效。

虽然具体符号测试在路径探索方面显著优于随机模糊测试，但硬件状态空间固有的复杂性意味着，它有时可能因求解器超时或状态空间爆炸而无法到达某些有效的逻辑分支。因此，被判定为不可达的分支会被视为推测性剪枝的候选对象。值得注意的是，这些候选对象通常包括必要的防御性逻辑（例如有限状态机默认状态）；在正常运行情况下，这些逻辑本质上不可达，但对硬件安全性至关重要。我们不采用计算开销巨大的形式化验证，而是利用大语言模型对设计意图的语义理解来消除这些潜在冗余。如果某个代码块被判定为冗余，大语言模型会试探性地将其删除，并在下一轮中由框架针对黄金模型执行调试。这可以确保行为保持不变，避免误删任何被误判为死代码的逻辑。在实践中，高级规格说明对于帮助大语言模型在阶段 1 生成初始输入，以及在阶段 3 优化期间识别被误判为死代码的逻辑，都十分重要。

E. 总结

总而言之，我们提出的框架通过利用以软件为中心的高级推理来指导面向硬件的验证，为自动化硬件生成建立了一种新范式。通过引入软件到硬件的黄金模型参考，以及白盒、感知轨迹的反馈循环，我们有效解决了现有基于大语言模型的寄存器传输级生成的主要不足，即缺乏自适应功能反馈。因此，我们的方法显著降低了大语言模型生成的设计中出现硬件错误的风险，同时通过剪枝冗余对其进行优化，从而为可靠的硬件设计提供了一种自动化方案。

IV. 实验

A. 实验设置

我们使用 OpenAI 的 GPT-3.5 [18]和 GPT-4 [29] 作为代码生成的大语言模型工具，实现了 AutoVeriFix+ 。这些模型在处理基于 Python 的问题时表现出色，因此非常适合创建参考模型。

为评估 Verilog 代码生成的性能，我们选取了两个具有代表性的基准测试：VerilogEval [6]和 RTLLM [30]。这些数据集专门用于评估语言模型生成的 Verilog 代码的功能正确性和质量。VerilogEval 数据集源自 HDLBits 教学网站上的 Verilog 编程任务。这些任务分为两个子集：VerilogEval-Human，包含 156 道精心设计的手工题目；以及 VerilogEval-Machine，包含 143 道由 GPT 生成的题目。RTLLM 数据集 v1.1 共包含 29 个设计，而 RTLLM v2.0 则提供了 50 个多样化的 Verilog 任务，这些任务与现实世界中的寄存器传输级设计场景十分相似，难度各异，可用于评估 AutoVeriFix+及其他现有方法的性能。

B. Python 参考模型的正确性

为验证阶段 1 生成的参考模型的有效性，我们首先使用两项指标分析最终的 Python 代码：语法正确性和功能正确性。语法正确性评估生成的代码能否在无语法错误的情况下成功编译。功能正确性评估代码在执行时是否产生预期输出。

表 I 展示了 Python 参考模型的语法正确性与功能正确性。由于 GPT-3.5 的结果相似，我们已将其从表中省略。实验结果表明，为各类基准测试生成参考模型是有效的，在所有基准测试中均实现了近乎完美的语法正确性，以及超过 94% 的功能正确性。尤其是在 VerilogEval-machine 基准测试中，参考模型取得了高达 98.60% 的功能正确性。这些结果表明，大语言模型能够将高层硬件规格转换为可执行的 Python 预言机。该预言机为后续的差分测试和符号具体探索提供了坚实基础。

虽然大语言模型能够生成高质量的 Python 代码，但在没有反馈的情况下生成的初始测试输入可能会出现覆盖率不足的问题，导致边界行为未得到充分规范。为缓解这一问题，我们引入了一种覆盖率驱动的反馈机制，将 Python 模型的执行覆盖率反馈给大语言模型，以优化测试输入。通过强制执行这一 Python 覆盖率反馈闭环，生成的测试向量质量得到了显著提升。如图 10 所示，仅依赖基线 GPT-4 输入时，在 RTLLM v2.0 基准测试上的假阳性率（FPR）高达 28.40%。然而，应用 Python 覆盖率反馈后，FPR 显著降至 7.60%。这表明，阶段 1 为后续的差分测试提供了高度稳健且可信赖的基础。

C. Verilog 代码的功能正确性

遵循 VerilogEval [6], ，我们使用指标*pass@k* 来评估生成的 Verilog 代码的功能正确性。*pass@k* 表示随机从个候选答案中选取 [6], 个时，至少有一个答案通过功能评估的预期概率：

表 I：使用 GPT-4 对阶段 1 中生成的参考模型进行评估。

Verilog 数据集	**语法 正确性 (%)**	**功能 正确性 (%)**
VerilogEval-human	99.35	96.15
VerilogEval-machine	100.00	98.60
RTLLM v1.1	100.00	96.55
RTLLM v2.0	98.00	94.00

随机从 n 个候选答案中选取 k 个时，至少有一个答案通过功能评估：

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

其中， n 表示给定问题的生成样本总数， c表示正确样本数， k表示选取的代码样本数。我们根据原始设置设定 $n = 10$。

我们将我们的方法与多类模型进行了比较：

- 闭源商业大语言模型：GPT-3.5 [18], 、GPT-4 [29]和 Claude3-Sonnet [35]。
- 通用开源模型：CodeLlama [32], 、Code- Qwen [33], 和 DeepSeek-Coder [34]。
- 最先进的特定于 Verilog 的模型：VerilogEval [6],、CodeGen-6B MEV-LLM [31], 、BetterV-CodeQwen [10],、RTLCoder [7] 和 OriGen [8]。

表 II 展示了各基准测试中的功能正确性结果。实验结果表明，AutoVeriFix+不仅显著优于 GPT-4 和 Claude3 等领先的商业大语言模型，也优于 OriGen 和 RTLCoder 等领域特定模型。即使在我们的框架中使用 GPT-3.5，我们的表现也超过了大多数模型。在我们的框架中使用 GPT-4 时，我们的方法表现更佳，在所有基准测试中始终优于其他所有模型。

1）RTLLM v2.0 上的性能：在 RTLLM v2.0 基准测试中，AutoVeriFix+表现出色，使用 GPT-3.5 和 GPT-4 时分别取得 71.9 和 85.4 的分数，显著优于所有基线模型。例如，与原始 GPT-3.5 相比，使用 GPT-3.5 的 AutoVeriFix+将 RTLLM v2.0 上的分数（pass@5）提高了 98.6%（从 36.2 提高到 71.9）。这凸显了基于阶段的细化过程在解决功能错误方面的有效性。

2) VerilogEval 上的评估：在 VerilogEval-human 基准测试中，使用 GPT-4 的 AutoVeriFix+ 取得了 87.2 的 pass@10 分数，优于商业模型 GPT-4 Turbo（72.4）和 Claude3-Sonnet 模型（60.3）。AutoVeriFix+取得的最高分是在 VerilogEval-machine 基准测试中，pass@10 分数为 91.6，超过了 OriGen（85.7）和 RTLCoder（81.8）等领域特定模型。如表 I 所示，该分数也相当接近参考模型的理想表现。

表 II：VerilogEval 基准测试上的功能正确性比较 [6] 以及 RTLLM 基准测试 [30]。每列排名 1st、 2nd和 3rd 最靠前的分数分别以红色、蓝色和绿色突出显示。

类别	模型	VerilogEval-Human (%)			VerilogEval-machine (%)			RTLLM v1.1 (%)	RTLLM v2.0 (%)
		pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@5	pass@5
特定于 Verilog 模型	VerilogEval [6]	28.8	45.9	52.3	46.2	67.3	73.7	-	-
	CodeGen-6B MEV-LLM [31]	42.9	48.0	54.4	57.3	61.5	66.4	-	-
	BetterV-CodeQwen [10]	46.1	53.7	58.2	68.1	79.4	84.5	-	-
	RTLCoder [7]	41.6	50.1	53.4	61.2	76.5	81.8	48.3	-
	OriGen [8]	54.4	60.1	64.2	74.1	82.4	85.7	65.5	-
开源模型	CodeLlama-7B-Instruct [32]	18.2	22.7	24.3	43.1	47.1	47.7	34.5	33.1
	CodeQwen1.5-7B-Chat [33]	22.4	41.1	46.2	45.1	70.2	77.6	37.9	36.4
	DeepSeek-Coder-7B-Instruct-v1.5 [34]	31.7	42.8	46.8	55.7	73.9	77.6	37.9	36.4
商业大语言模型	Claude3-Sonnet [35]	46.1	56	60.3	58.4	71.8	74.8	58.6	54.4
	GPT-3.5 [18]	35.6	48.8	52.6	49.4	72.7	77.6	44.8	36.2
	GPT-4 [29]	43.5	55.8	58.9	60	70.6	73.5	65.5	58.7
	GPT-4 Turbo [29]	54.2	68.5	72.4	58.6	71.9	76.2	65.5	63.4
Ours	AutoVeriFix+ 与 GPT-3.5	58.5	71.8	73.7	69.5	77.6	79.7	75.9	71.9
	AutoVeriFix+ 与 GPT4	78.1	84.4	87.2	83.7	89.2	91.6	90.2	85.4

总体而言，这些结果表明，使用 GPT-4 的 AutoVeriFix+ 不仅优于商业通用大语言模型，在生成具备功能正确性的 Verilog 代码方面也胜过领域特定模型。

D. 消融实验：反馈机制的重要性

为了评估 AutoVeriFix+ 中每种反馈机制的独立贡献，我们在 RTLLM v2.0 和 VerilogEval 基准测试上进行了全面的消融实验。表 III 和表 IV 总结了实验结果，展示了语法反馈、输入/输出功能反馈和基于轨迹的调试如何逐步提升代码生成质量。

1) 语法反馈评估：我们首先通过比较 GPT-4 的零样本基线与仅加入语法反馈的版本，评估第 2 阶段语法反馈循环的影响。如表 III 所示，加入自动化编译器反馈显著提升了生成的寄存器传输级实现的语法正确性，使 pass@1 得分从 58.6% 提高到 77.8%。

然而，语法修复对功能正确性的影响因数据集复杂度而异。在较简单的 VerilogEval 基准测试中（表 IV），修复语法错误会使功能通过率大幅提升（例如，VerilogEval-Human 上的 pass@1 得分从 43.5% 提高到 68.1%）。这表明，对于较简单的问题，大语言模型逻辑通常是正确的，语法错误才是主要障碍。相反，在复杂的 RTLLM v2.0 数据集上，仅进行语法修复几乎不会提升功能表现（从 51.6% 提高到 52.4%）。这说明，对于复杂的硬件逻辑，编译是必要但不充分的条件，

表 III：语法反馈与功能反馈对提升 RTLLMv2.0 上 pass@k 分数的影响。

模型	语法（%）			函数（%）		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
GPT-4	58.6	80.3	84.8	51.6	58.7	68.9
GPT-4 + 语法反馈-back	77.8	89.6	90.0	52.4	64.5	70.7
GPT-4 + I/O反馈	89.8	92.0	94.0	74.6	83.5	88.0
GPT-4 + 轨迹感知调试	92.9	94.0	96.0	79.0	85.4	88.6

表 IV：语法反馈和功能反馈对提升 VerilogEval-Human 上 pass@k 分数的作用。

模型	语法（%）			函数（%）		
	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
GPT-4	78.5	87.2	89.1	43.5	55.8	58.9
GPT-4 + 语法反馈-back	91.5	96.7	97.2	68.1	70.2	75.8
GPT-4 + I/O反馈	98.0	99.6	99.9	77.2	82.7	84.6
GPT-4 + 轨迹感知调试	98.6	99.6	99.9	78.1	84.4	87.2

正确性；底层逻辑仍存在缺陷，需要语义指导。

2) 基于输入/输出的功能反馈评估：阶段 3 中差分测试与高覆盖率完整输入测试集的结合，对功能正确性的提升最为显著。为分离差分刺激相对于因果轨迹数据的具体影响，我们开展了一项受控实验。我们构建了一个精简版

的“基于轨迹的调试”提示词（图 8），省略了跟踪反馈部分（内部状态快照和执行路径）。这个精简版提示词代表一种黑盒差分测试范式，仅向大语言模型提供特定的失败测试向量、预期的黄金行为以及观察到的输出不匹配。

实验结果表明，即使缺乏内部可见性，具体符号引擎生成的高质量测试输入也能显著提升大语言模型的调试能力。在 RTLLM v2.0 上，加入基于 I/O 的反馈后，pass@1 功能得分从 52.4% 提升至 74.6%。这证实了针对黄金参考实现进行高覆盖率差分测试，对于解决常见的组合逻辑和时序逻辑错误非常有效。在 VerilogEval-Human 上也观察到类似的提升（从 68.1% 提高至 77.2%），进一步验证了基于 Python 的参考模型在为各种硬件设计提供准确验证预言机方面的稳健性和实用性。

3）基于轨迹的调试反馈的评估：轨迹感知调试能够处理仅凭黑盒输入输出反馈无法有效诊断的复杂功能错误。轨迹感知调试在具有挑战性的 RTLLM v2.0 基准测试上效果尤为显著。高覆盖率 I/O 反馈虽已建立起 74.6% 的 pass@1 准确率基线，但加入细粒度执行轨迹后，性能进一步提升至 79.0%，绝对增幅为 4.4%。这一结果表明，对于状态密集型设计，仅提供通用的“输出不匹配”信息不足以进行诊断。相比之下，周期级精确地呈现内部状态和执行路径，使大语言模型能够进行因果推理，并修正深层的时序逻辑和时序相关逻辑错误。

E. 验证完整性与假阳性分析

自动化寄存器传输级生成面临的一项重大挑战是假阳性的出现，即某个实现被标记为“正确”，但内部仍存在功能缺陷。不同于不提供置信度度量的传统生成方法，AutoVeriFix+ 实现了透明的验证循环。通过在最终寄存器传输级代码旁输出差分仿真结果，该框架提供了验证凭证，突出显示残留差异，供人在回路中

审查。

1) 寄存器传输级结果分类：为了评估我们框架的可靠性，我们对所有基准测试套件中生成的寄存器传输级代码进行了人工检查。尽管 Python参考实现并非百分之百正确，但人工检查揭示了一个一致的趋势：所有未通过差分测试的生成 Verilog 代码都是错误的。因此，我们将生成的 Verilog 代码分为三类：

- 未通过差分测试的设计。
- 通过差分测试但功能上不正确的设计（假阳性）。
- 功能正确的设计。

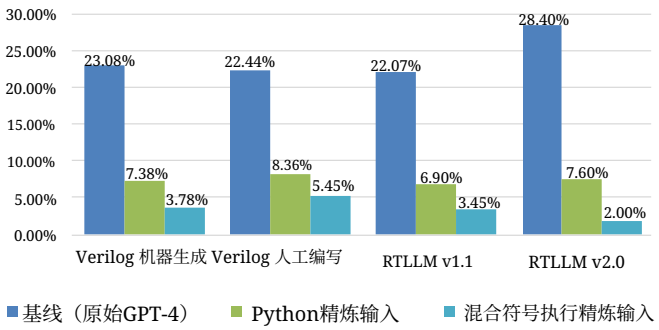


图 10：通过差分测试但功能上不正确的设计所占比例。

2) 覆盖率驱动精炼对假阳性率（*FPR*）的影响：即使参考模型质量很高，且输入经过阶段 1 的精炼，测试平台仍可能不完整，从而使功能不正确的寄存器传输级设计通过验证。为量化这一风险，我们评估假阳性率（FPR），其定义为通过测试平台验证但功能不正确的设计所占的比例。

$$FPR = 1 - \frac{\#\ \text{designs that are functionally correct}}{\#\ \text{designs that pass the differential testing}} \quad (2)$$

较低的假阳性率表明，具体符号引擎生成的 FullInput测试集具有很高的可靠性，能够有效过滤无效设计，并确保大多数通过验证的设计保持高质量。

我们在四个基准测试中评估了三种不同配置下的假阳性率：（1）**基线（原始GPT-4）**，使用阶段 1 中未经优化的原始测试平台；（2）**Python 优化** 配置，通过反馈优化阶段 1 测试平台的完备性；（3）**具体符号优化（AutoVeriFix+）**配置，在阶段 3 引入具体符号引擎，以最大限度提高寄存器传输级代码覆盖率。如图 10 所示，结果证实，纳入这些逐步完善的反馈机制可显著降低假阳性率，从而有效过滤潜在的功能错误。

在所有基准测试中，基线的假阳性率都显著偏高，在 RTLLM v2.0 上达到峰值 28.40%。这表明，通过朴素测试平台验证的设计中，近三分之一实际上存在潜在的功能错误，因此基线输出不适合直接采用。相比之下，集成具体符号测试能够成功拦截这些隐蔽错误，将 RTLLM v2.0 上的假阳性率大幅降至仅 2.00%，并且在其他基准测试中也始终保持较低水平。假阳性率逐步降低，验证了 AutoVeriFix+作为严格筛选器的作用，并证实我们由具体符号测试驱动的机制能够确保经验证的设计在功能上稳健且高度可靠。

F. 具体符号测试评估

如表 V 所示，在阶段 3 中集成具体符号测试后，所有评估基准测试的 Verilog 分支覆盖率均得到稳定且显著的提升

。基于 Python 参考模型（阶段 1）生成的输入向量提供了可靠的基线，平均覆盖率约为 88%；但由于随机或基于启发式的测试生成存在局限，它们无法达到完全覆盖。阶段 3 中的具体符号测试有效弥补了这一差距。例如，在 VerilogEval-Human 基准测试中，覆盖率从 85.22% 飙升至 95.83%。此外，在 RTLLM v1.1 基准测试中，具体符号引擎实现了近乎完美的覆盖率，达到 99.63%。

表 V：评估使用 GPT-4 在阶段 1 生成的输入向量在阶段 3 中通过具体符号测试实现的 Verilog 分支覆盖率。

Verilog 数据集	基线覆盖率（%）	Python 优化覆盖率 (%)	具体符号执行优化覆盖率 (%)
VerilogEval-machine	84.54	87.31	92.37
VerilogEval-human	79.66	85.22	95.83
RTLLM v1.1	79.96	90.91	99.63
RTLLM v2.0	81.89	89.74	98.34

这种近乎穷尽式的覆盖率直接解释了上一节讨论的高可靠性和低假阳性率。通过将逻辑覆盖率提升至接近 100 %，具体符号测试能够揭示潜在缺陷，而这些缺陷在基准测试平台中原本会被忽略。RTLLM v2.0 的假阳性率从 28.40% 大幅降至 2.00%，这直接归功于具体符号引擎能够系统地遍历初始测试平台遗漏的复杂状态转换。较高的 Verilog 覆盖率也为冗余消除任务提供了所需的数学置信度。例如，在 RTLLM v1.1 中，覆盖率达到 99.63% 这一事实意味着，剩余 0.37% 的代码很可能是不可达的死代码或冗余的防御性逻辑。

G. 冗余消除评估

为量化冗余消除对物理实现的影响，我们使用 Yosys 综合生成的寄存器传输级设计，并比较阶段 3 前后的平均门数量。表 VI 展示了结果，并将其分为总单元数（包括寄存器）和组合逻辑单元。

对于包含多种逻辑问题的 VerilogEval 数据集，我们观察到门数量显著减少：机器生成集减少了 19.2%，人工整理集减少了 26.6%。这表明，大语言模型最初生成的代码通常包含多余的逻辑分支（例如冗余的 default case 或无法到达的 else 代码块），这些分支会增加电路面积，却不会带来任何功能上的贡献。对于
在包含复杂且真实的 RTL 设计场景的 RTLLM v2.0 数据集中，其总门数量优化了 10.5%，组合逻辑优化了 4.5%。

H. 定性案例研究

为展示阶段 3 在处理复杂实现细节方面的有效性，我们以一个控制

表 V：使用 Yosys 综合对各基准测试进行综合后，门数量的总平均值和组合逻辑平均值。

Verilog 数据集	优化前		优化后	
	总计	Comb	总计	Comb
VerilogEval-machine	147.84	117.93	119.42	91.65
VerilogEval-human	149.46	117.68	109.69	91.18
RTLLM v1.1	314.28	268.16	246.87	214.35
RTLLM v2.0	360.67	310.62	322.76	296.53

基于执行轨迹的调试提示

你是一名经验丰富的 Verilog 验证工程师。你的目标是根据执行跟踪分析修复所提供代码中的功能错误。具体来说，请重点关注寄存器状态与执行路径之间的因果关系。请修复 Verilog 代码，使其符合预期行为。

- [Original Code]
  < 如清单 1 所示：示例 1 >
- [验证失败报告]
  < 测试用例输入：时钟周期 1：复位=0，输入=8'h02 >
  < 预期行为：输出=1>
  < 观测到的不匹配：输出=0>
- [跟踪反馈]
  以下跟踪展示了失败时钟周期的确切执行路径（以 Exec: B_i 标记），以及寄存器状态（以 R 标记）。
  周期 1 跟踪：Exec: B_3; R 计数器 = 0; Exec: B_7; R 输出 = 0

图 11：调试示例

逻辑模块为例进行分析，如清单 2 所示。规范要求模块在收到特定输入（8'h02）时递增内部计数器，并在计数器达到 1 时立即置位输出 out。最初的大语言模型生成的代码（清单 2）未能满足这一时序要求。代码在更新计数器的同一个时钟块内检查 counter 的值，以确定 out 的状态。由于非阻塞赋值直到时间步结束时才会更新寄存器值，因此逻辑表达式 if (counter == 8'h01) 判断的是旧值（0），而不是新值（1）。因此，输出 out 比预期晚一个时钟周期置位。

如果我们仅反馈仿真报告中的不匹配信息：“预期输出=1，实际输出=0”，大语言模型往往会产生幻觉，错误地修改复位逻辑或极性，未能解决根本原因。正如图 11 所示，我们的框架会捕获故障周期中的执行轨迹和寄存器状态，并向大语言模型提供详细反馈。

接收阶段 3 反馈后生成的优化代码展现出显著改进。首先，在跟踪反馈的指导下，大语言模型重构了设计，将组合逻辑（下一个计数器）与时序逻辑分离。通过检查下一个计数器，这种做法明确解决了时序竞争，确保输出信号与更新后的状态保持一致。借助覆盖率报告，大语言模型成功移除了冗余的默认分支，使设计更加简洁且面积效率更高。

清单 2：优化后的 Verilog

```
模块 top (
    输入 wire 时钟,
    输入 wire 复位,
    输入 导线 [7:0] in,
    输出寄存器 out
);
    reg [7:0] 计数器;
    reg [7:0] 下一个计数器; // [修正]引入中间状态
    // 1. 组合逻辑 用于 下一个状态
    始终块 @(*) 开始
        case (输入)
            8'h00: next_counter = counter - 1;
            8'h02: 下一个计数器 = 计数器 + 1;
            8'hFF: next_counter = 8'h00;
            // [优化]'默认' case语句已移除
            基于UNSAT证明
        endcase
    end
    // 2. 时序逻辑更新
    始终块 @(上升沿 时钟) begin
        如果 (复位) begin
            计数器 <= 8'h00;
            out <= 1'b1;
        end
        else begin
            计数器 <= 下一计数器;
            // [修复]检查'next_counter'(the新value)而不是'counter'。
            // 这可确保 '输出' 反映 该 状态
            更新后。
            如果（下一个计数器 == 8'h01）
                输出 <= 1'b1;
            否则
                输出 <= 1'b0;
        end
    end
结束模块
```

V. 结论

本文提出了一种用于寄存器传输级代码生成的三阶段框架，该框架利用大语言模型在 Python 代码生成方面的优势来创建基于 Python 的参考模型。在第一阶段，框架将 Python 参考模型确立为功能标准，提供指导测试平台创建的高层实现。框架还引入了反馈机制，使大语言模型能够通过分析覆盖率指标来提高测试平台覆盖率。在第二阶段，大语言模型利用编译器反馈迭代处理语法错误，并通过识别 Verilog 实现与测试平台之间的不一致来纠正功能错误。这些迭代改进确保生成的 Verilog RTL 保持较高的功能正确性。实验结果表明，我们的方法在生成准确的 Verilog 代码方面优于现有的通用大语言模型和硬件专用模型。

参考文献

[1] A. Mastropaolo、L. Pascarella、E. Guglielmi、M. Ciniselli、S. Scalabrino、R. Oliveto 和 G. Bavota，《代码生成技术的鲁棒性：一项关于 GitHub Copilot 的实证研究》，载于 *2023 IEEE/ACM 第 45 届软件工程国际会议（ICSE）*，2023 年，第 2149–2160 页。

[2] E. Nijkamp、H. Hayashi、C. Xiong、S. Savarese和Y. Zhou，“Codegen2：在编程语言与自然语言上训练大语言模型的经验教训”，arXiv预印本 arXiv:2305.02309，2023。[3] S. Thakur、B. Ahmad、Z. Fan、H. Pearce、B. Tan、R. Karri、B. Dolan-Gavitt和S. Garg，“面向自动化Verilog RTL代码生成的大语言模型基准测试”，载于2023年欧洲设计、自动化与测试会议暨展览（DATE）。IEEE，2023，第1–6页。[4] E. Dehaerne、B. Dey、S. Halder和S. De Gendt，“面向设计与验证自动化的Verilog自动补全深度学习框架”，arXiv预印本 arXiv:2304.13840，2023。[5] S. Liu、Y. Lu、W. Fang、M. Li和Z. Xie，“OpenLLM-RTL：面向大语言模型辅助设计RTL生成的开放数据集与基准测试（特邀）”，载于2024年IEEE/ACM计算机辅助设计国际会议（ICCAD）。ACM，2024。[6] M. Liu、N. Pinckney、B. Khailany和H. Ren，“VerilogEval：评估大语言模型的Verilog代码生成”，载于2023年IEEE/ACM计算机辅助设计国际会议（ICCAD），2023，第1–8页。[7] S. Liu、W. Fang、Y. Lu、Q. Zhang、H. Zhang和Z. Xie，“RTLCoder：借助我们的开源数据集与轻量级解决方案，在设计RTL生成方面超越GPT-3.5”，载于2024年IEEE大语言模型辅助设计研讨会（LAD）。IEEE，2024，第1–5页。[8] F. Cui、C. Yin、K. Zhou、Y. Xiao、G. Sun、Q. Xu、Q. Guo、D. Song、D. Lin、X. Zhang和Y. E. Liang，“OriGen：通过代码到代码增强与自我反思提升RTL代码生成”，载于2024年ACM/IEEE计算机辅助设计国际会议（ICCAD），2024，第1–9页。[9] M. Chen、J. Tworek、H. Jun、Q. Yuan、H. P. D. O. Pinto、J. Kaplan、H. Edwards、Y. Burda、N. Joseph、G. Brockman等，“评估在代码上训练的大语言模型”，arXiv预印本 arXiv:2107 03374，2021。[10] Z. Pei、H.-L. Zhen、M. Yuan、Y. Huang和B. Yu，“BetterV：基于判别式引导的受控Verilog生成”，arXiv预印本 arXiv:2402.03375，2024。[11] S. Thakur、J. Blocklove、H. Pearce、B. Tan、S. Garg和R. Karri，“AutoChip：利用大语言模型反馈实现HDL生成自动化”，arXiv预印本 arXiv:2311.04887，2023。[12] Y. Tsai、M. Liu和H. Ren，“RTLFixer：利用大语言模型自动修复RTL语法错误”，载于第61届ACM/IEEE设计自动化会议论文集，2024，第1–6页。[13] C.-T. Ho、H. Ren和B. Khailany，“VerilogCoder：基于图的规划与抽象语法树（AST）波形跟踪工具的自主Verilog编码智能体”，arXiv预印本 arXiv:2408.08927，2024。[14] J. Decaluwe，“MyHDL：一种基于Python的硬件描述语言”，2003，访问日期：2026。 [在线]。可获取：http://www.myhdl.org [15] Y. Chi和J. Cong，“利用计算复用实现模板加速器”，载于2020年第57届ACM/IEEE设计自动化会议（DAC）。IEEE，2020，第1–6页。[16] Y. Tan、X. Meng、Z. Jiang和Y. Lyu，“AutoVeriFix：自动纠正大语言模型生成的 Verilog 代码中的错误并提升功能正确性”，载于2026年第31届亚洲及南太平洋设计自动化会议（ASP-DAC），2026，第526–532页。[17] Q. Sun、Z. Chen、F. Xu、K. Cheng、C. Ma、Z. Yin、J. Wang、C. Han、R. Zhu、S. Yuan等，“神经代码智能综述：范式、进展与展望”，arXiv预印本 arXiv:2403.14734，2024。[18] OpenAI，“GPT-3.5-turbo”，2023，访问日期：2026。 [在线]。可获取：https://platform.openai.com/docs/models/gpt-3 5 - H. Touvron、T. Lavril、[19] G. Izacard、X. Martinet、M.-A. Lachaux、T. Lacroix、B. Rozi`ere、N. Goyal、E. Hambro、F. Azhar等，“LLaMA：开放且高效的基础语言模型”，arXiv预印本 arXiv:2302.13971，2023。[20] OpenAI，“OpenAI Codex”，2023，https://openai.com。[21] Y. Li、D. Choi、J. Chung、N. Kushman、J. Schrittwieser、R. Leblond、T. Eccles、J. Keeling、F. Gimeno、A. Dal Lago等，“基于AlphaCode的竞赛级代码生成”，《Science》，第378卷，第6624期，第1092–1097页，2022。[22] J. Blocklove、S. Garg、R. Karri和H. Pearce，“Chip-chat：对话式硬件设计中的挑战与机遇”，载于2023年ACM/IEEE第五届面向CAD的机器学习研讨会（MLCAD）。IEEE，2023，第1–6页。[23] J. Gai、H. Chen、Z. Wang、H. Zhou、W. Zhao、N. Lane和H. Fan，“探索代码语言模型以实现基于HLS的自动化硬件生成：基准测试、基础设施与分析”，载于第30届亚洲及南太平洋设计自动化会议论文集，2025，第988–994页。

[24] S. Thakur、B. Ahmad、H. Pearce、B. Tan、B. Dolan-Gavitt、R. Karri和S. Garg，“Verigen：面向Verilog代码生成的大语言模型”，*ACM*电子系统设计自动化汇刊，第29卷，第3期，第1–31页，2024。[25] E. Nijkamp、B. Pang、H. Hayashi、L. Tu、H. Wang、Y. Zhou、S. Savarese和C. Xiong，“Codegen：支持多轮程序合成的开放代码大语言模型”，*arXiv*预印本 *arXiv:2203.13474*，2022。[26] J. Wang、S. Liu、Y. Lu和Z. Xie，“HLSDebugger：利用大语言模型解决方案识别并纠正HLS代码中的逻辑错误”，*arXiv*预印本 *arXiv:2507.21485*，2025。[27] K. Chang、Y. Wang、H. Ren、M. Wang、S. Liang、Y. Han、H. Li和X. Li，“Chipgpt：我们距离自然语言硬件设计还有多远”，*arXiv*预印本 *arXiv:2305.14019*，2023。[28] Y. Lyu和P. Mishra，“RTL模型的可扩展符号具体混合测试”，*IEEE*计算机汇刊，第70卷，第7期，第979–991页，2021。[29] OpenAI，“GPT-4技术报告”，2023，访问日期：2026。 [在线]。可获取：https://openai.com/research/gpt-4 [30] Y. Lu、S. Liu、Q. Zhang和Z. Xie，“RTLLM：面向利用大语言模型进行设计RTL生成的开源基准”，载于*2024*年第*29*届亚洲及南太平洋设计自动化会议（*ASP-DAC*）。IEEE，2024，第722–727页。[31] B. Nadimi和H. Zheng，“面向Verilog代码生成的多专家大语言模型架构”，载于*2024*年*IEEE*大语言模型辅助设计研讨会（*LAD*）。IEEE，2024，第1–5页。[32] B. Roziere、J. Gehring、F. Gloeckle、S. Sootla、I. Gat、X. E. Tan、Y. Adi、J. Liu、R. Sauvestre、T. Remez 等，“Code Llama：面向代码的开放基础模型”，*arXiv*预印本 *arXiv:2308.12950*，2023。[33] J. Bai、S. Bai、Y. Chu、Z. Cui、K. Dang、X. Deng、Y. Fan、W. Ge、Y. Han、F. Huang 等，“通义千问技术报告”，*arXiv*预印本 *arXiv:2309.16609*，2023。[34] D. Guo、Q. Zhu、D. Yang、Z. Xie、K. Dong、W. Zhang、G. Chen、X. Bi、Y. Wu、Y. Li 等，“DeepSeek-Coder：当大语言模型遇见编程——代码智能的兴起”，*arXiv*预印本 *arXiv:2401.14196*，2024。[35] Anthropic，“介绍下一代Claude”，https://www. anthropic.com/，2024。