

基于大语言模型的高效程序验证方法

康 克, 尹良泽[†], 董 威

国防科技大学 计算机学院, 湖南 长沙 410003

收稿日期: 2025-11-14 [†] 通信联系人 E-mail: yinliangze@163.com

基金项目: 国家挑战项目(TZ2025002); 国家自然科学基金(U2341212)

第一作者: 康 克, 男, 硕士生, 现从事高可信软件方面的研究。E-mail: kk503390960@163.com

摘 要: 随着软件系统规模与复杂度的增长, 传统程序验证在效率与可靠性方面面临严峻挑战。形式化验证虽能提供数学级别的正确性保障, 但在处理大规模程序时常遭遇状态空间爆炸等技术瓶颈。为缓解上述限制, 提出了两种基于大语言模型的高效程序验证方法: 数据规模优化与反例生成。数据规模优化方法通过大语言模型分析程序语义逻辑与数据规模的关联, 在保持程序语义逻辑不变的前提下压缩验证所需数据规模, 并结合数据流切片确保大语言模型优化结果的可靠性; 反例生成方法则通过大语言模型自动构造潜在反例, 并通过测试验证其有效性。在此基础上, 进一步构建了一个融合大语言模型与传统验证工具的并程序验证框架, 通过多线程机制同时执行数据规模优化、反例生成与传统验证工具流程, 并优先采纳最先完成流程的验证结果, 实现效率与可靠性的平衡。在 SV-COMP 基准集上与传统工具进行的对比实验结果表明, 本文方法与框架均显著提升了验证任务覆盖能力与资源利用效率, 同时保持较高验证准确率, 为智能化程序验证提供了有效解决方案。

关 键 词: 程序验证; 形式化方法; 大语言模型

中图分类号: TP311.5

文献标志码: A

文章编号: 1671-8836(2026)04-0447-14

Efficient Program Verification Methods Based on Large Language Models

KANG Ke, YIN Liangze[†], DONG Wei

College of Computer Science and Technology, National University of Defense Technology, Changsha 410003, Hunan, China

Abstract: As software systems grow in scale and complexity, traditional program verification faces severe challenges in efficiency and reliability. Although formal verification provides rigorous correctness guarantees, it often suffers from technical bottlenecks such as state-space explosion when handling large-scale programs. To address this limitation, this paper proposes two efficient LLM-based program methods: data-scale optimization and counterexample generation. The former analyzes the relationship between program semantics and data scale, reduces verification scale while preserving semantic logic, and uses data-flow slicing to ensure reliability. The latter constructs potential counterexamples with an LLM and validates them through testing. Furthermore, this paper builds a parallel verification framework integrating LLMs and traditional verification tools, executing the two methods and the baseline tool concurrently and adopting the earliest validated result. Experiments on the SV-COMP benchmark show that the proposed methods and framework improve task coverage and resource efficiency while maintaining high verification accuracy, providing an effective solution for intelligent program verification.

Key words: program verification; formal methods; large language models

0 引 言

随着计算机技术的发展与广泛应用, 软件系统

已从单纯的信息处理工具演变为航空航天、自动驾驶、金融科技、医疗设备等安全攸关领域的核心基础设施^[1]。软件规模的指数级增长与功能逻辑的高

引用格式: 康克, 尹良泽, 董威. 基于大语言模型的高效程序验证方法[J]. 武汉大学学报(理学版), 2026, 72(4): 447-460. DOI: 10. 14188/j. 1671-8836. 2025. 0217.

KANG Ke, YIN Liangze, DONG Wei. Efficient Program Verification Methods Based on Large Language Models [J]. J Wuhan Univ (Nat Sci Ed), 2026, 72(4): 447-460. DOI: 10. 14188/j. 1671-8836. 2025. 0217(Ch).

度复杂化,使得其正确性与安全性直接关系到人身财产、社会运行乃至国家战略的稳定与发展^[2]。

程序验证作为形式化方法的重要分支,通过对程序语义进行数学建模与逻辑推理,为软件正确性提供严格保证^[3]。但在面对包含大规模数组、深层循环和复杂控制结构的程序时,验证过程往往受到状态空间爆炸问题的制约,导致时间与内存开销迅速增长,难以兼顾验证效率、资源消耗与覆盖范围^[4-6]。即便是工业界广泛采用的有界模型检验(Bounded Model Checking, BMC)技术,也需在验证深度与资源消耗之间进行权衡,从而影响性能^[7]。

近年来,大语言模型(Large Language Model, LLM)在代码理解与语义建模任务中展现出较强能力。借助大规模代码语料预训练,LLM能够较好地捕捉程序结构模式、变量依赖关系和控制流特征,可用于辅助识别语义冗余、分析潜在错误路径以及生成验证相关候选信息^[8]。然而,LLM本身并不能直接提供形式化正确性论证,其推理过程也缺乏可解释性,输出结果也会受到概率生成和训练分布的影响,从而产生结构幻觉或语义偏差,难以直接作为可信的验证依据^[9-12]。因此,如何将LLM的语义分析能力与形式化验证工具进行有效结合,构建兼具效率与可靠性的程序验证方法,成为值得研究的重要问题^[13]。

针对上述问题,本文提出了两种基于LLM的高效程序验证方法,分别从验证规模优化与缺陷反例搜索两个角度提升程序验证效率与适用性。其中,数据规模优化方法主要适用于包含数组规模、循环边界等可缩减规模参数,且验证属性对数据规模不敏感的程序;反例生成方法主要适用于可达性验证和存在性缺陷搜索场景,尤其适合通过具体输入触发验证属性失败的验证任务。在此基础上,本文进一步构建了一个融合LLM与传统验证工具的验证框架,通过并行执行机制协同集成多种验证流程,以提高整体验证性能。

1 相关理论与工作

本文方法建立在传统程序验证工具之上,并通过引入LLM进行语义辅助分析与候选生成。因此,本节主要从程序验证基础、程序中间表示以及大语言模型三个方面介绍与本文方法相关的理论基础,为后续方法设计提供支撑。其中,程序验证基础部分重点介绍有界模型检验,程序中间表示部分说明其在程序分析中的作用,大语言模型部分讨论其在程序分析与验证辅助中的应用现状。

1.1 程序验证基础

在程序验证过程中,程序通常被抽象为状态转换系统,通过遍历状态空间或路径集合判断是否存在违反规范的可达执行路径^[14]。对于包含循环、分支和数组操作的程序,验证条件往往需要随着路径展开不断累积,导致状态表示与约束规模持续增长,并进一步增加求解过程的计算复杂度^[15-16]。尽管研究者提出了偏序约简、抽象解释与对称性压缩等优化技术,但在涉及复杂循环展开和大规模数据对象的验证任务中,状态空间爆炸问题仍未得到较好解决^[17]。

BMC作为工程实践中广泛采用的验证技术,通过对程序执行路径进行有限步展开,将路径可达性问题编码为布尔可满足性(SAT)或可满足性模理论(SMT)问题^[7,18-19]。该方法能够在有限步范围内有效发现违反断言的可达路径,但其验证效率在很大程度上受路径展开规模与约束表达复杂度影响。当循环展开层数增加、分支路径增多或数据对象规模扩大时,公式规模会快速增长,从而带来更高的时间与内存开销。因此,在保证程序语义正确性的前提下控制状态扩展规模,是提升BMC类方法验证效率的重要途径。

1.2 程序中间表示

程序中间表示(Intermediate Representation, IR)是现代程序分析与验证系统的核心组成部分,其作用在于桥接源代码与形式化工具之间的结构鸿沟,为静态分析、路径抽取、验证建模等任务提供统一的语义抽象基础。IR的设计目标是保留程序的语义信息与结构特征,同时消除语言依赖与语法冗余,使后续分析可以在跨语言、跨平台的条件下进行^[20-22]。

主流IR包括静态单赋值形式(SSA)^[23]、控制流图(CFG)^[24]、数据流图(DFG)^[25]与程序依赖图(PDG)^[26]等。其中,SSA主要通过变量唯一赋值机制显式表示数据定义与使用关系;CFG和DFG分别从控制转移与数据传播角度描述程序结构;PDG则在此基础上综合刻画控制依赖与数据依赖,能够更直接地表示语句之间对程序行为和验证属性的影响关系。由于本文后续需要围绕断言相关依赖关系进行数据流切片与行为一致性分析,因此重点采用PDG作为支撑结构。

基于上述IR,尤其是PDG所表示的控制依赖与数据依赖结构,可实现高效的数据流分析与程序切片,提取与验证目标相关的最小影响子程序,从而在保持语义完整的同时压缩待验证状态空间,提升模型检验与SMT求解的效率。尤其在LLM生成优化方案后,需要对语义一致性进行验证时,IR及其分析

能力为差异检测与一致性判定提供了结构化支持。

1.3 大语言模型

随着 Transformer 架构的发展与大规模预训练语料的积累, LLM 在自然语言处理与代码建模领域取得突破性进展, 逐步成为程序分析、缺陷检测与验证辅助任务的重要工具^[27-28]。其能够从源代码中捕捉程序结构、变量依赖和控制流特征, 为程序验证中的属性构造、路径分析与候选信息生成提供语义支持^[28]。

在程序验证领域, LLM 的应用大致可归纳为 3 个方向: 一是自动构建验证属性, 如将自然语言需求转化为形式化表达^[29]; 二是辅助生成验证相关结构, 如预测循环不变式、前置条件与逻辑断言等^[29-32]; 三是参与验证过程优化, 如定位潜在缺陷路径、生成测试用例或反例、提供路径约束简化建议等^[30, 33]。相关研究如 AutoSpec^[34]、Lemur^[31] 与 iRank^[32] 已展示了 LLM 在任务分解、条件构造与路径排序方面的有效性, 表明其在降低验证门槛与提升自动化程度方面具有应用潜力。

但现有多数研究仍将 LLM 定位为验证辅助工具, 而非独立验证主体。LLM 给出的推理过程缺乏可解释性, 其生成结果通常需借助模型检验、定理证明等机制进行严格校验。基于此, 如何在发挥 LLM 语义分析与候选生成优势的同时, 通过外部验证机制约束

其输出可靠性, 仍是当前相关研究中的关键问题。

2 本文方法

2.1 数据规模优化

在程序验证中, 数据规模往往会影响路径展开范围与约束求解复杂度, 尤其在包含数组与循环结构的程序中, 这一影响更为明显。对于基于 BMC 的验证方法, 输入规模增大通常会带来更高的验证开销。但进一步分析可发现, 部分验证属性在逻辑上并不依赖具体的数据规模。若其验证结论在不同规模下保持一致, 则可在不改变程序语义逻辑的前提下适当缩减数据规模, 以降低验证复杂度。这种语义层面的规模独立性构成了本文数据规模优化方法的基本出发点。

因此, 本文引入 LLM 作为核心推理引擎, 提出一种基于 LLM 的数据规模优化方法。该方法通过 LLM 对程序变量、控制结构与验证属性之间的依赖关系进行推理分析, 识别语义上与属性无关的数据规模, 并在保证验证结论等价的前提下将其缩减至尽可能小的可行数据规模。整体流程如图 1 所示, 其中, 语义解析、语义独立性判定与优化生成阶段由 LLM 驱动完成, 相关提示词模板见表 1。

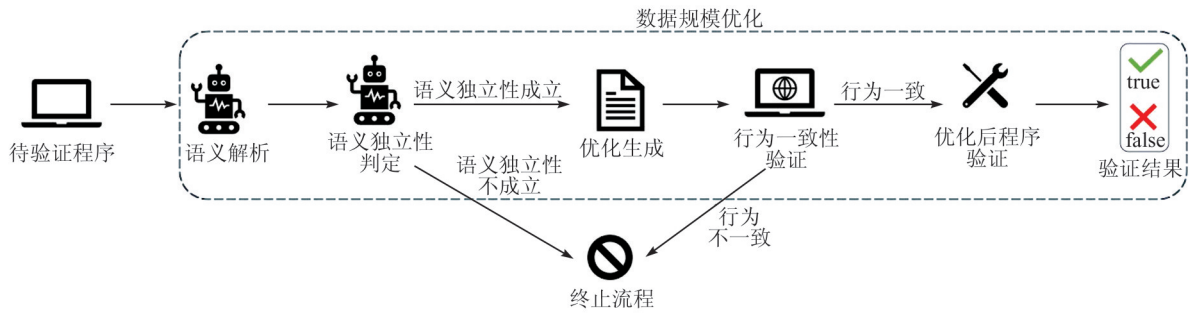


图1 数据规模优化流程

Fig. 1 Process of data scale optimization

2.1.1 语义解析

在语义解析阶段, LLM 会通过思维链的方式对整个程序的行为、结构进行分析, 逐步梳理程序中的控制流关系、关键变量的数据依赖以及与验证属性相关的核心语义逻辑, 从而形成对程序整体执行行为的综合语义理解。该分析结果并不直接作为验证结论使用, 而是作为后续流程的语义基础, 提供必要的上下文信息。

2.1.2 语义独立性判定与优化生成

在语义独立性分析阶段, 设程序为 P , 验证属性为 x , 数据规模变量为 N , 程序 P 在数据规模为 N 时的执行行为表示为 $\text{exec}(P, N)$, $\text{exec}(P, N) \models x$ 表示

程序在规模 N 下满足属性 x 。

若对于任意两个输入规模 $N_1, N_2 \in N$, 总有:

$$\text{exec}(P, N_1) \models x \Leftrightarrow \text{exec}(P, N_2) \models x$$

则称程序 P 在验证属性 x 上对规模变量 N 语义独立, 程序 P 在验证属性 x 上的判定结果与具体数据规模无关。

在此阶段, LLM 基于前述语义解析结果, 判断数据规模变量是否会影响验证属性相关的执行过程与判定结果。在判定语义独立性成立后, LLM 将进行优化生成, 自动构造优化版本程序 P' , 并将原始规模变量 N 缩减为满足验证等价的尽可能的小值 N' 。若语义独立性不成立, 则终止数据规模优化流程。

表 1 数据规模优化方法相关提示词

Table 1 Prompts for data scale optimization

阶段	提示词	输出示例
语义解析	你是一位程序语义分析专家,请对给定的程序按照以下顺序进行分析:概括程序整体结构,识别关键模块与调用关系;梳理程序中的控制流路径,包括顺序、分支与循环;分析关键变量的数据依赖关系,追踪其赋值、传递与使用过程;然后,提炼程序的核心语义逻辑,特别关注与验证属性相关的逻辑片段;综合以上分析,构建程序的行为分析,总结其整体执行语义与潜在的数据依赖。请逐步思考。 [待验证程序代码]	[对程序的逐步分析过程]
语义独立性判断	现在,我需要对该程序进行验证,但可能由于数据规模较大,验证工作非常困难。请分析程序的语义逻辑是否与数据规模变量的大小直接相关。简而言之,数据规模的大小是否会影响程序验证属性的相关运行过程或结果?请逐步思考。	{ 分析过程:[简单分析过程] 判断结果:[“独立”,“不独立”] }
优化生成	既然认为程序的语义逻辑与数据规模无关,也就是说,如果控制数据规模的变量被适当缩小,不会影响整个程序验证属性的正确性,那么请提供控制数据规模的变量名称以及它缩小到尽可能最大程度后的结果。每个变量只能提供一个具体值。	{ 规模变量:[“N:3”,“SIZE:10”] }

注:N和SIZE仅为示例,表示程序中控制数据规模的变量名;表中冒号后的数值表示经优化后建议采用的数据规模取值。

2.1.3 行为一致性验证

在行为一致性验证阶段,若经检测数据规模优化前后两程序的行为保持一致,则进入优化后程序验证阶段;若两程序的行为未能保持一致,则终止数据规模优化流程。

设 P 为源程序(数据规模为 N), P' 为规模优化后程序(数据规模为 N'), V 表示程序的输入空间, $v \in V$ 为任意一个具体输入实例。记 $\text{exec}(P, v)$ 为程序 P 在输入 v 下的执行行为, $\text{exec}(P, v) \models x$ 表示程序在输入 v 下满足属性 x 。

若对于任意输入 $v \in V$, 均有:

$$\text{exec}(P, v) \models x \Leftrightarrow \text{exec}(P', v) \models x$$

则称程序 P 与 P' 在属性 x 上行为一致。

上述定义要求程序在不同数据规模下对验证属性给出一致的判定结果,同时保证整体执行语义不发生改变。具体到程序行为而言,当数据规模发生变化时,程序的控制流结构、可达路径以及语句执行顺序应保持不变。数据规模优化仅允许影响与规模相关的数值取值,而不得引入新的控制分支、改变原有路径选择或重排语句执行序列。通过对执行轨迹一致性的约束,可以保证数据规模缩减前后的程序在逻辑语义层面保持一致。

为提高该行为一致性验证的自动化程度,本文设计了一种基于数据流切片的等价性分析方法。在验证过程中,通过 Frama-C 的 PDG 与 Slicing 相关插件^[35]提取程序 P 与 P' 关于验证属性 x 的切片,分别记为 $S(P, x)$ 与 $S(P', x)$ 。其中,程序切片表示

所有可能影响属性 x 判定结果的语句集合。若这两个切片在结构上满足三项约束:数据依赖关系一致、控制路径一致与语义执行顺序一致,则认为切片结构等价,记作: $S(P, x) \equiv S(P', x)$ 。PDG 会构建程序中的两种依赖关系:若语句 s_1 的变量定义可能影响语句 s_2 的变量使用,则在二者之间建立数据依赖边;若语句 s_1 的控制条件决定语句 s_2 是否执行,则建立控制依赖边。通过追踪断言语句的所有前驱节点, Frama-C 能自动生成影响断言语句相关变量的语句集合,即该断言的切片。

为便于刻画两个切片结构之间的相似程度,进一步引入切片结构相似度函数:

$$\text{Sim}(S(P, x), S(P', x)) = \frac{|E_P \cap E_{P'}|}{|E_P \cup E_{P'}|}$$

其中, E_P 与 $E_{P'}$ 分别为两切片的依赖边集合。该相似度函数基于集合交并比进行计算,用以衡量两个程序在属性 x 相关依赖结构上的重合程度。若 $\text{Sim} = 1$, 则表示切片完全等价;若 $0 < \text{Sim} < 1$, 则说明切片存在依赖差异。在实际验证过程中,当切片结构相似度为 1 时,即程序 P 与程序 P' 的切片内容完全相同时,才认为两个切片在结构上具有相似性。

在上述切片结构相似度分析的基础上,可以进一步探讨结构等价与行为一致性之间的理论关系。直观而言,若两个程序在验证属性 x 相关的依赖结构上完全一致,则其对断言结果的影响路径应保持不变。基于此,本文提出如下定理,用以说明切片结构等价对行为一致性的充分性。

定理 1(切片一致性蕴含行为一致性) 设 P 为源程序, P' 为优化后程序, V 表示程序输入空间, $v \in V$ 为任意输入示例, x 为验证属性, $S(P, x)$ 与 $S(P', x)$ 分别为 P 与 P' 关于属性 x 的程序切片, $\text{exec}(P, v)$ 为程序 P 在输入 v 下的执行行为, $\text{exec}(P, v) \models x$ 表示程序在输入 v 下满足属性 x 。

若程序 P 与 P' 在属性 x 的程序切片等价, 即 $S(P, x) \equiv S(P', x)$, 则:

$$S(P, x) \equiv S(P', x) \Rightarrow \forall v \in V, \text{exec}(P, v) \models x \Leftrightarrow \text{exec}(P', v) \models x$$

即程序 P 与 P' 在属性 x 上行为一致。

证 根据程序切片的定义, $S(P, x)$ 包含所有可能通过数据依赖或控制依赖影响属性 x 判定结果的语句。若 $S(P, x) \equiv S(P', x)$, 两程序在影响属性 x 的语句集合及其依赖结构完全一致。

由于属性 x 的判定结果完全由该切片内语句决定, 且切片结构一致, 则对于任意输入, $v \in V$ 两程序对属性 x 的判定结果相同, 即:

$$\text{exec}(P, v) \models x \Leftrightarrow \text{exec}(P', v) \models x$$

证毕。

2.1.4 优化后程序验证

在通过语义和行为一致性验证后, 优化后程序 P' 将被输入至验证工具执行验证, 将程序 P' 的验证

结果作为源程序 P 的最终验证结果。在优化程序 P' 的验证过程中, 因为状态空间已被显著压缩, 模型求解效率显著提升, 验证时间与资源消耗得到大幅度优化。

2.2 反例生成

前述数据规模优化方法在一定程度上缓解了程序验证中的状态空间爆炸问题, 但其适用前提是程序中存在可压缩的数据规模变量(如数组长度或循环展开深度)。对于不满足该前提或语义上无法进行规模缩减的程序, 仅依赖数据规模优化仍不足以高效确认程序是否存在对应缺陷。因此, 本文进一步引入基于 LLM 的反例生成方法, 以补充后续验证框架在一般场景下的适用能力。

基于 LLM 的反例生成方法的核心目标在于通过 LLM 辅助构造触发断言失败的具体输入, 判定程序在给定验证属性 x 下是否存在对应错误。相较于传统形式化验证需要遍历或编码大量路径空间, 该方法通过使用 LLM 快速得到可能违反属性的执行路径, 并尝试构造满足该路径约束的输入实例, 从而以存在性证明的方式快速确认缺陷。方法总体流程如图 2 所示, 其中, 缺陷分析、缺陷路径识别与反例生成阶段由 LLM 参与完成, 相关提示词模板如表 2 所示。

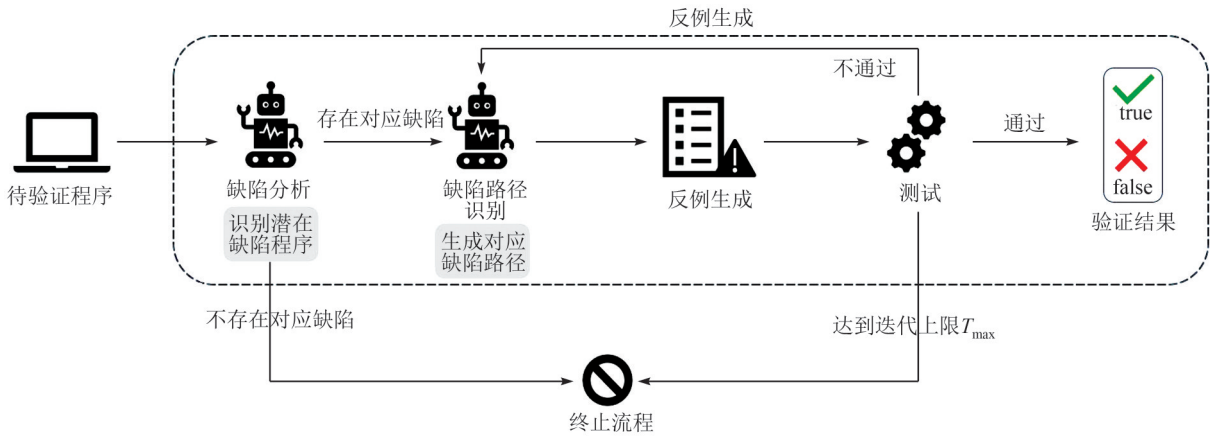


图 2 反例生成流程

Fig. 2 Process of counterexample generation

2.2.1 缺陷分析

在缺陷分析阶段, LLM 同样首先对程序的控制流结构、数据依赖关系以及验证属性相关逻辑进行语义解析。在此基础上, 本阶段进一步面向断言失败风险进行分析, 判断目标属性是否可能被违背。若判断存在违背风险, 即存在对应缺陷, 则进入缺陷路径识别阶段; 若判断不存在违背风险, 即不存在对应缺陷, 则终止反例生成流程。该分析仅作为候选缺陷路径识别与反例构造的辅助依据, 而最终

缺陷判定仍需通过测试机制完成。

2.2.2 缺陷路径识别

在缺陷路径识别阶段, LLM 结合前述分析结果, 从断言语句的逻辑条件出发进行逆向推导, 识别可能导致属性 x 失效的关键执行路径, 并抽取相应路径约束集合。该阶段的目标在于缩小候选反例的搜索空间, 提高后续反例生成的针对性与有效性。

2.2.3 反例生成

设程序为 P , V 表示程序的输入空间, 验证属性为

表 2 反例生成方法相关提示词

Table 2 Prompts for counterexample generation

阶段	提示词	输出
缺陷分析	你是一位缺陷分析专家,请对给定的程序按照以下要求进行分析:明确验证属性,梳理控制流与数据流,接着推断违反属性的潜在执行路径与状态条件,然后构建具体的缺陷触发场景,最后总结缺陷的根本原因。请逐步思考。[待验证程序代码]	{ 分析过程: 是否含有对应缺陷:[“是”,“否”] }
	请基于前面的分析内容,确认需重点关注的相关验证属性,然后结合程序控制流结构,系统识别最可能违反上述属性的具体执行路径:对路径需清晰描述其起止点与具体信息,以及触发该路径所需的控制条件与数据约束。	{
缺陷路径识别	伪反例反馈:经测试,刚才提供的缺陷路径并不能使断言无效,这与之前的目标相悖。因为对于这个程序,您的分析结果表明存在一组赋值可以使断言无效,而且我也相信您之前对程序的分析是正确的。请基于您之前对整个程序的分析,重新进行相关缺陷路径分析。本次回复不能与之前的任何回复相同。	路径描述: }
反例生成	请锁定程序中的输入变量,并根据前面的缺陷路径描述,构建一组能够触发相关缺陷的输入值。	{ 变量:[“n:3”,“j:5”,“k:2”] }

注: n 、 j 和 k 仅为输出示例,表示待验证程序中的输入变量名;冒号后的数值表示 LLM 生成的候选反例取值。

x 。程序在输入 $v \in V$ 下的执行行为表示为 $\text{exec}(P, v)$, $\text{exec}(P, v) \not\models x$ 为程序 P 在输入 v 下不满足属性 x 。

若存在输入 $v_{\text{ce}} \in V$ 满足:

$$\text{exec}(P, v_{\text{ce}}) \not\models x$$

则称 v_{ce} 为属性 x 的有效反例。

有效反例的关键特征在于其能够在程序真实运行中直接导致断言失败,从而证实程序在所定义语义属性 x 上存在实际的逻辑缺陷。因此,一旦构造出有效反例,即可直接否定属性 x 的全称成立性。

在反例生成阶段,LLM 根据缺陷路径识别阶段识别出的路径约束条件,构造满足该约束的输入 $v_{\text{ce,LLM}}$ 作为候选反例。候选反例仅作为潜在缺陷的假设性输入描述,其是否为有效反例必须通过后续测试来验证其真实性。

2.2.4 测试

在测试阶段,系统将候选反例嵌入待验证程序执行,并检测其是否能够触发验证属性 x 失败。若满足 $\text{exec}(P, v_{\text{ce,LLM}}) \not\models x$,则认为该候选反例通过测试,确认其为有效反例,输出验证结果;若未触发验证属性失败,则认为该候选反例不通过测试,视为无效反例。系统会将失败测试信息反馈给 LLM,调整路径约束并重新生成候选输入进行测试,形成迭代搜索机制。设最大迭代次数为 T_{max} (在实验中 T_{max} 被设置为 15),当迭代次数达到 T_{max} 仍未生成有效反例时,终止反例生成流程。

2.3 基于大语言模型的高效程序验证框架

上述两种方法在验证流程中处于不同技术层

级:前者属于规模压缩增强策略,后者属于候选解构造策略。为充分发挥两者优势,本文进一步构建统一并行验证框架,对两种方法与传统验证工具进行协同调度,实现效率与可靠性的统一。

图 3 给出了本文提出的基于 LLM 的程序验证框架总体流程。对于给定待验证程序,系统采用并行执行的策略,分别运行数据规模优化模块、反例生成模块以及传统验证工具模块。3 个模块相互独立运行,由调度机制优先采纳率先完成模块的验证结果。

在数据规模优化模块中,LLM 首先对程序语义逻辑与数据规模之间的依赖关系进行分析,并判断两者是否存在关联。若存在关联,则放弃该模块;若不存在关联,LLM 生成规模优化后的程序版本。随后,基于 Frama-C 的数据流切片反向分析,对原程序与优化程序进行行为一致性验证。若一致性不成立,则放弃该模块;若一致性成立,则将优化后的程序提交至程序验证工具执行,并由验证工具输出最终验证结果。

在反例生成模块中,LLM 首先进行程序的缺陷分析,并判断目标安全属性是否可能存在缺陷。若判断不存在缺陷,则放弃该模块。若判断存在缺陷,LLM 进行反例生成,生成候选反例并将其封装为测试用例执行。随后,系统判断测试用例是否触发缺陷:若成功触发缺陷,则认为该候选反例通过测试,并输出验证结果;若未触发缺陷且尚未达到最大迭代次数,则反馈测试信息并重新生成候选反例;若达到迭代上限仍未获得有效反例,则放弃该模块。

对于传统验证工具模块,系统会直接调用程序

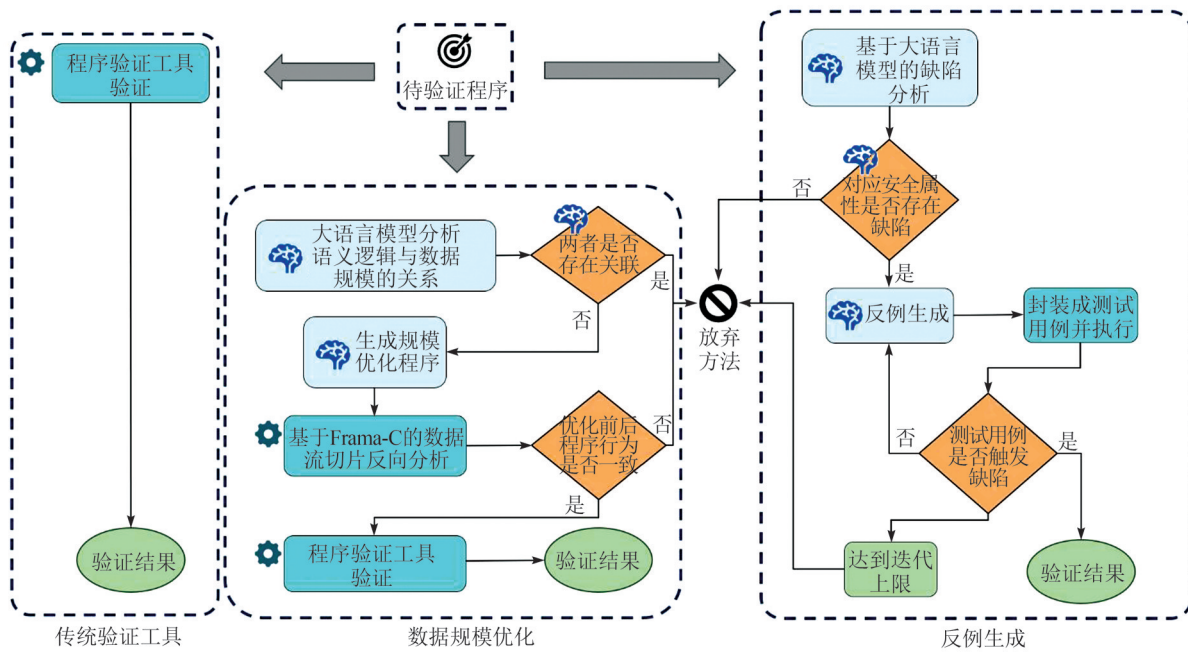


图3 基于大语言模型的高效率程序验证框架

Fig. 3 Efficient program verification framework based on LLM

验证工具对待验证程序进行验证并得到结果。

3 实验评估

3.1 实验设置

3.1.1 数据集及任务

本文选用 SV-COMP 2024 标准基准集 SV-Benchmarks^[36]作为实验数据来源。本文所使用的验证任务属于该基准集中的 ReachSafety 类别,其中具体任务类型为 unreachable-call(不可达调用),其目标是判断程序执行过程中是否存在可达的断言失败路径。基于该任务类型,本文进一步构建了一个固定评测子集,共包含 3 043 个 C 程序验证任务。

本文实验使用两个层次的数据集:一是包含 3 043 个程序的完整评测集,用于反例生成方法与并行验证框架的性能评估;二是从完整评测集中进一步筛选得到的 703 个数组或循环程序子集,用于评估数据规模优化方法的专项效果。这 703 个数组或循环程序子集的构建依据为程序是否包含数组或循环结构,因为数据规模优化方法主要面向此类受输入规模影响更显著的验证场景。

3.1.2 对比方法

为评估本文所提出并行验证框架的有效性,本文选择传统形式化验证方法和 GPT-4o 直接推理方法作为对比。具体包括:

CBMC^[19]:传统有界模型检验工具,直接对原始程序进行验证,作为形式化验证基线。CBMC 基于

有界模型检验思想,将程序在给定展开界限下的执行路径编码为约束求解问题,并调用后端求解器完成可满足性判定,在 C 程序验证领域具有较强代表性。

GPT-4o^[37]:仅使用 LLM 对程序进行语义分析并直接给出验证结论,不结合形式化验证工具,用于评估纯 LLM 推理方法的性能特点及其局限性。

3.1.3 实验环境

实验中,框架默认采用 GPT-4o 作为 LLM 组件,用于完成程序语义分析、候选优化方案生成、反例构造等任务;在后续鲁棒性分析中,进一步将底层模型替换为 DeepSeek-V3^[38],以考察框架在不同语言模型支持下的性能稳定性。

LLM 生成过程中涉及温度参数设置。温度参数主要用于控制模型输出的随机性与多样性:较低温度下,模型更倾向于选择高概率输出,生成结果相对稳定;较高温度下,模型输出的多样性增强,更有利于探索不同候选推理路径或输入构造方案。考虑到本文框架中数据规模优化和反例生成分支均需要一定程度的候选方案探索能力,除特别说明外,本文实验中语言模型温度参数默认设置为 0.8。

所有实验均在 Windows Subsystem for Linux (WSL2) 环境下进行,操作系统为基于 64 位 x86 架构的 Linux 内核(版本 5.10.16.3)。硬件配置包括 AMD Ryzen 7 7840H 处理器,系统总可用内存为 27 GB。

为保证实验公平性与可比性,每个验证任务均设置统一资源约束:最大运行时间为 900 s,最大内

存使用限制为 12 GB。所有实验均在相同硬件环境与资源限制下运行。本文框架的时间统计区间为：从开始处理单个验证任务起,到首个有效验证结果返回止;内存统计为产生该有效结果的完成路径所对应的峰值内存开销,并包含框架调度、候选生成、测试执行或验证工具调用等必要本地开销,而不是将三个并行分支的内存消耗简单累加。对于未率先产生有效结果的分支,其运行结果不作为该任务最终统计依据。CBMC 对比实验则统计其作为独

立验证工具直接运行原始任务时的总耗时与峰值内存。

3.2 框架实验结果与分析

3.2.1 总体性能分析

为验证本文所提出框架在整体任务求解能力方面的效果,本节对其在统一资源约束下的总体性能进行分析,重点比较框架与传统验证工具 CBMC 在任务解决能力、平均验证时间以及平均内存消耗等方面的差异。实验结果如表 3 所示。

表 3 实验框架与传统验证工具的总体验证结果

Table 3 Overall verification results of the experimental framework and traditional verification tool

方法	任务总数	解决数量	LLM 处理平均时间/s	总平均时间/s	平均内存/MB
本文框架	3 043	1 103	22.14	23.76	18.40
CBMC	3 043	297	—	133.65	6 411.67

注：“—”表示该方法不涉及对应模块或该指标不适用,因此未进行统计。

从任务解决能力来看,本文框架共成功解决 1 103 个任务,约为 CBMC(297 个)的 3.7 倍,这表明在相同时间与内存资源约束下,框架能够显著扩展 CBMC 在复杂程序场景中的有效任务覆盖范围。这是因为框架并非仅依赖单一路径展开或约束求解,而是通过并行集成数据规模优化、反例生成以及传统工具验证多个分支,使不同验证流程能够在适合的程序类型上优先发挥作用,从而提高整体任务完成率。

在时间效率层面,框架的总平均验证时间为 23.76 s,显著低于 CBMC 的 133.65 s。该差异一方面来源于 LLM 在语义层面对潜在错误路径进行定向推理,避免对大规模状态空间进行盲目遍历;另一方面得益于并行调度机制,在多个模块同时运行的情况下优先采纳最快返回的有效结果,从而整体缩短等待时间。

在内存开销方面,框架的平均内存占用为 18.40 MB,而 CBMC 达到 6 411.67 MB。该差异反映了两类方法在搜索策略上的本质不同:传统程序验证需构建大规模约束编码与状态空间;在框架中,反例生成通过生成反例及测试验证的方式直接验证可达性,数据规模优化通过压缩验证所需的数据规模,从而避免高成本的全局探索,大幅度减少了整体验证的计算消耗。

进一步将总平均时间拆分可知,框架中 LLM 处理的平均耗时为 22.14 s,占总平均时间的 93.18%,这说明框架的主要时间开销集中在与 LLM 的交互与推理环节。尽管 LLM 成为框架主要耗时来源,但其带来的任务解决能力提升与内存大幅下降,使得整体验证流程在工程可用性上更具

优势。

综上,实验结果表明,本文框架在统一资源约束下相较于传统验证工具 CBMC 表现出更强的任务解决能力,同时具有更低的平均验证时间和显著更低的内存开销。

3.2.2 正确性与可靠性分析

为分析本文框架在提升任务解决能力的同时是否保持较高的结果可靠性,本节进一步统计并比较不同方法在验证正确性方面的表现。具体地,本文将所提出框架与 GPT-4o 直接推理方法及 CBMC 基线进行对比,分析 LLM 引入后可能带来的误判情况,以及外部校验机制对误判的抑制作用。针对该实验,本文统计了不同方法在 3 043 个基准任务上的 TP、TN、FP、FN 与 T/M 分布情况,其含义如下:

TP:实际存在缺陷的程序,被方法正确判定为存在缺陷;

FN:实际存在缺陷的程序,被方法错误判定为不存在缺陷(即漏判);

FP:实际无缺陷的程序,被方法错误判定为存在缺陷(即误判);

TN:实际无缺陷的程序,被方法正确判定为不存在缺陷;

T/M:受时间与内存资源限制,方法无法完成对该程序的评估,不能输出判定结果。

准确率:仅基于已完成验证并给出明确判定结果的任务计算,不包含 T/M 任务。

实验结果如表 4 所示。从表中可以看出,仅使用 GPT-4o 进行语义推理时,其准确率为 68.09%,并伴随较高的 FP 与 FN 数量。这一结果表明,在缺

表4 不同方法的总体准确率比较

Table 4 Overall accuracy comparison of different methods

方法	TP	TN	FP	FN	T/M	准确率/%
框架	653	427	5	18	1 940	97.91
GPT-4o直接推理	1 278	794	243	728	0	68.09
CBMC	180	116	0	1	2 746	99.66

乏形式化校验与可执行验证的情况下,虽然LLM表现出了极高的推理效率(零T/M),但其推理结果仍会受到语义误解、幻觉现象等影响,从而导致错误判断。因此,仅依赖自然语言推理难以保证其结论严谨性。

相比之下,CBMC的准确率达到99.66%,零误判且仅1个缺陷漏检,成功识别180个缺陷程序、正确判定116个安全程序,体现了传统形式化验证工具在结论可靠性方面的优势。然而,受状态空间爆炸等问题约束,其T/M任务数量高达2 746个,说明在严格资源约束下,大量程序无法在限定时间或内存内完成验证。

本文框架在准确率达到97.91%的同时,将T/M任务数量降低至1 940个,实现了较高准确率与显著覆盖能力提升之间的平衡。一方面,框架通过反例测试验证以及语义一致性校验的机制对LLM输出进行检验,从而有效抑制了LLM推理可能带来的误判倾向,使FP与FN均维持在较低水平;另一方面,当LLM分支在复杂程序上无法稳定收敛时,CBMC兜底分支仍可提供形式化保证,从而整体保持接近传统工具的高可信度。

对于框架实验结果中的FP与FN,需要结合具体方法的误差来源进一步分析。一方面,部分被统计为FP的程序在基准集中被标注为无缺陷,即其不可达调用(unreach-call)任务的验证结果应为真,但在反例生成方法中,这些程序被检出存在缺陷,并成功生成了对应反例,且相关反例经测试后均能够真实触发断言失败。因此,这部分结果在统计上表现为FP,但其产生原因并不能简单归结为方法本身的无效推断。另一方面,在数据规模优化中为了保证优化前后程序的行为一致性,方法使用了Frama-C的PDG插件辅助进行,但本身静态分析工具存在短板,易产生误报^[39],尽管在针对数据规模优化方法的专项实验(3.3.1节)中,经过行为一致性验证后的程序均能够得到正确验证结果,但在更大规模、更多样化的程序上,仍可能存在个别边界情况,影响整体判定结果的准确性。

综上,实验结果表明,本文框架在显著提升任

务覆盖能力的同时,仍保持了接近传统形式化验证工具的较高准确率,体现出LLM语义推理与工具校验相结合在可靠性与效率之间的有效平衡。

3.2.3 方法贡献与适用性分析

为进一步分析框架整体性能提升的来源,本节对各验证分支在成功任务中的贡献情况进行了统计分析。图4展示了框架在全部3 043个基准程序中成功解决的1 103个任务的策略分布情况。

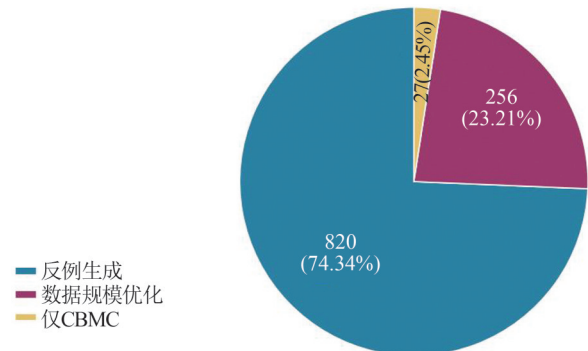


图4 框架方法分布图

Fig. 4 Strategy distribution of solved tasks in the verification framework

从图4中可以看出,反例生成策略在已解决任务中占据主导地位,数据规模优化策略起到重要补充作用,而CBMC分支主要承担兜底验证角色。

反例生成方法贡献最大的主要原因在于unreach-call任务本质上属于可达性判断问题。对于大量程序而言,若存在违反断言的路径,构造一个具体输入并通过测试验证即可完成任务,而无需进行完整的符号状态空间遍历。LLM在语义层面分析程序控制流与数据依赖后,能够快速定位潜在错误路径,并生成候选输入作为反例。这种方式在大量缺陷程序上具备较高效率与较强适用性,因此成为框架中的主要覆盖来源。

数据规模优化方法则主要覆盖数组操作密集或深层循环嵌套的程序。在这类程序中,传统模型检测工具往往因状态空间指数级增长而难以在资源限制内完成验证。通过分析数据规模变量与验证属性之间的语义独立性关系,框架能够在保持行为一致性的前提下缩减问题规模,从而将原本不可达的验证任务转化为可计算实例。因此,该方法在状态空间爆炸型程序中发挥了关键补充作用。

另外,实验仅有27个任务最后经CBMC验证返回结果。这一现象并不意味着CBMC在验证能力上不足,而是由于本文框架采用并行调度机制,并优先采纳更快返回的有效结果。在多数情况下,LLM分支已在CBMC完成之前给出可靠结论;

CBMC 主要承担在 LLM 分支无法收敛或结果不稳定时的兜底与保障角色。

综上,实验结果表明,本文框架的性能提升并非来源于单一方法,而是多方法与验证工具协同作用的结果,不同分支在不同类型程序上发挥了互补优势。

3.2.4 鲁棒性分析

为分析本文框架在不同条件下的稳定性,本节从跨模型泛化性与温度参数敏感性两个维度,对所提出框架的鲁棒性进行了进一步实验分析。实验结果如表 5 所示。

1) 跨模型泛化性分析

为验证本文框架对底层 LLM 组件的依赖程度,本文以 GPT-4o 作为默认 LLM 配置,并在保持数据规模优化、反例生成、并行调度机制以及传统验证工具模块均不变的条件下,将框架中的 LLM 组件替换为 DeepSeek-V3 进行对比实验。需要说明的是,该实验比较的是同一并行验证框架在不同 LLM 组件支持下的性能差异,而非不同验证架构之间的差异。

如表 5 所示,以 DeepSeek-V3 为底层 LLM 的框架在 3 043 个任务中成功解决 1 025 个任务,准确率达到 96.29%,平均耗时与内存占用与以 GPT-4o 为底层 LLM 的默认框架保持在同一数量级。尽管

在任务完成数量与准确率上略低于默认框架,但整体性能仍显著优于仅使用 CBMC 进行验证,并维持较高的稳定性。

这一结果表明,框架的性能提升主要来源于数据规模优化、反例生成与工具兜底的系统化设计,而非依赖某一特定 LLM 的偶然优势。不同模型之间的性能差异更多体现在语义推理细节与生成策略偏好上,但在并行校验与形式化工具约束机制下,整体输出结果能够保持一致性与稳定性。

2) 温度参数敏感性

对于温度参数敏感性分析,本文选取温度 $T \in [0.2, 1.0]$ 的典型区间进行扩展测试^[28,37,40]。

如表 5 所示,在不同温度设置下,框架的解决任务数量在 1 100 附近小幅波动,准确率始终稳定在 97% 以上,平均耗时与平均内存占用这些核心性能指标的波动幅度均处于可忽略范围,说明框架对 LLM 的温度参数变化具有不敏感性,即便参数在合理区间内调整,仍能保持稳定的验证性能。

该结果表明,在参数范围内,框架对 LLM 输出随机性的敏感度较低,其关键原因在于后续的形式化约束与一致性校验机制能够抑制推理阶段的噪声传播,体现出较强的参数鲁棒性。

表 5 跨模型对比与温度参数敏感性分析

Table 5 Cross-model comparison and temperature parameter sensitivity analysis				
方法	解决任务数量	平均时间/s	平均内存/MB	准确率/%
框架(DeepSeek-V3)	1 025	28.14	28.13	96.29
框架(GPT-4o)($T=0.2$)	1 056	24.74	20.27	98.14
框架(GPT-4o)($T=0.5$)	1 067	25.10	22.84	97.69
框架(GPT-4o)($T=0.8$)	1 103	23.76	18.40	97.91
框架(GPT-4o)($T=1.0$)	1 090	24.38	19.51	97.26

3.3 核心方法独立实验

前文从整体层面对本文框架的性能进行了评估。由于数据规模优化方法与反例生成方法分别面向不同类型的验证问题,且适用场景与作用机制存在差异,仅依据框架整体结果难以充分反映两种方法各自的实际作用。因此,本节进一步对这两个核心方法开展独立实验,以分别验证其在对应任务场景下的有效性。

3.3.1 数据规模优化

如表 6 所示,经基于 LLM 的语义分析判定,其中 287 个程序具备数据规模优化潜力,即同时满足:数据规模变量为固定值;对于验证属性,程序语义逻辑与数据规模之间不存在依赖关系。为避免 LLM 误判而错误缩减程序规模,本文进一步对上述 287 个程序开展缩减前后的行为一致性验证。结果显示,其中 267 个程序的缩减版本未改变源程序在

表 6 实验框架与传统验证工具在数据规模优化中的比较

Table 6 Comparison between the experimental framework and verification tools in data scale optimization							
方法	任务总数	优化潜力任务数	优化后行为一致任务数	解决数量	LLM 处理平均时间/s	平均时间/s	平均内存/MB
数据规模优化	703	287	267	267	19.03	20.56	18.40
CBMC	703	—	—	36	—	196.53	7 244.12

注:“—”表示未进行统计。

断言属性上的语义行为,排除了20个因语义关联性误判而产生的无效优化方案。

在效率层面,数据规模优化显著降低了传统工具的计算负担。在该子集中,CBMC仅能在资源约束内完成36个任务,而数据规模优化可成功解决更多数组规模较大或循环嵌套较深的程序,并显著降低平均验证时间与内存开销。这一结果表明,该方法可以有效缓解状态空间爆炸问题,同时保证验证结论的可靠性。

需要指出的是,20个被过滤的无效优化案例主要源于数据规模变量在隐式条件分支或循环边界中参与逻辑判断,导致其与验证属性存在间接依赖关系。这说明,在复杂依赖结构下,单纯依赖LLM的语义抽象仍可能产生误判,而后续引入一致性验

证机制对于保证整体安全性至关重要。

综上,数据规模优化方法在数组与循环场景下能够有效缩减验证规模,并通过一致性验证机制保证语义等价,从而在状态空间膨胀型程序中发挥关键作用。

3.3.2 反例生成

从表7中可以看出,反例生成方法共完成931个程序的验证,其中895个程序的验证结论与真实属性一致,准确率达到96.13%。在资源开销方面,其平均验证时间为23.55 s,平均内存占用为10.41 MB,显著低于传统CBMC的时间与内存消耗。这一结果说明,基于语义推理的反例搜索在大量可达性任务中能够以较低资源成本完成验证。

表7 实验框架与传统验证工具在反例生成中的比较

Table 7 Comparison between the framework and verification tools in counterexample generation

方法	任务总数	解决数量	正确数量	LLM处理平均时间/s	平均时间/s	平均内存/MB
反例生成	3 043	931	895	22.14	23.55	10.41
CBMC	3 043	271	270	—	104.57	4 484.49

注:“—”表示该方法不涉及对应模块或该指标不适用,因此未进行统计。

进一步分析36个没有正确解决的失败案例发现,反例生成方法的误差主要集中在两类场景。第一类是实际无缺陷程序在缺陷分析阶段被误判为存在缺陷。在此情况下,LLM可能过度抽象或忽略某些约束条件,导致生成的候选反例无法在实际执行中触发断言失败。第二类是程序实际存在缺陷但反例搜索未能成功构造有效输入,通常与路径条件复杂、循环依赖链较长或涉及精细算术语义有关。另外,实验在数据集中发现3个标记为无缺陷(即其unreach-call任务验证结果应为真)的程序,但在反例生成方法下,这些程序被检出存在缺陷且生成了对应有效反例。图5展示了前述三个程序中的两个案例。为便于呈现,这两个程序已做简化处理,但保留了原始程序的核心要素。图5(a)的程序首先初始化大小为100 000的数组 a ,随后通过非确定性整数 x 控制的循环决定是否提前终止循环。在每次迭代中,若 x 等于0,则将当前 j 的值赋给 $a[i]$ 并更新 j 和 k 的值(j 增加 i , k 减少 i)。循环结束后,程序验证数组中第二个元素到第 i 个元素的值是否均大于等于 k 的终值,若验证失败则触发错误。

该过程旨在检验循环内数组元素赋值与后续条件检查的正确性。若仅采用形式化方法分析该程序行为,可推导出在第 i 次循环迭代时:

$$a[i] = \sum_{m=0}^{i-1} m = \frac{(i-1) \times i}{2}$$

```

1. int N = 100000;
2. int main()
3. {
4.   int j;
5.   int i = 0;
6.   int k = 0;
7.   int a[N];
8.   for(i=0; i<N; i++){
9.     int x = rand();
10.    if(x) break;
11.    a[i] = j;
12.    j = j + i;
13.    k = k - i;
14.  }
15.  for(i=1; i<k; i++)
16.    assert(a[i] >= k);
17.  return 0;
18. }

```

(a) 整数溢出误报示例

```

1. int main()
2. {
3.   long double x = 1.0/7.0;
4.   long long res = 0;
5.   for(int i = 1; x != 0; i++)
6.   {
7.     res += ((int)(x * 10) % 10) * (i * 10);
8.     x = (x * 10) - (int) x * 10;
9.   }
10.  }
11.  }
12.  assert(res == 67050);
13.  return 0;
14. }
15.

```

(b) 浮点数运算验证示例

图5 SV-COMP 无缺陷程序中识别出缺陷的情况

Fig. 5 Defect identification in defect-free programs of SV-COMP

$$\text{且 } k = -\sum_{m=0}^{i-1} m = -\frac{(i-1) \times i}{2}$$

若不考虑溢出,数组 a 所有元素均为正数而 k 为负值,因此断言 $\text{assert}(a[i] \geq k)$ 恒成立。但实际程序执行时,变量 j 和 k 均可能发生溢出,导致 k 的终值为 $-704\,982\,704$,而数组 a 部分元素实际小于 $-704\,982\,704$ 。这意味着在100 000次迭代中,部分情况下会出现 $a[i] < k$,从而触发缺陷。这对安全攸关程序构成严重隐患,可能导致程序行为异常。

对于图5(b)不含随机赋值的程序,本文方法则采用测试的方法来验证程序。测试结果显示 res 的终值为56 430,不满足验证属性 $\text{assert}(\text{res} = 67\,050)$ 。此外还存在第三个类似程序,经框架的

反例生成与测试后,同样发现实际结果属性不成立。

4 局限性

本文提出的基于 LLM 的程序验证方法与框架在实验评估中显著提升了验证效率,在一定程度上缓解了传统程序验证方法在大规模与高复杂度上遇到的状态空间爆炸的瓶颈问题,但该方法在实际应用中仍然存在若干局限性,接下来将进行分析。

4.1 幻觉的不确定性

尽管 LLM 在程序理解与逻辑推理方面展现出较强能力,但其在程序验证任务中仍不可避免地会产生不同形式的幻觉。相较于自然语言生成任务,程序验证对推理的精确性与一致性要求更高,模型生成的中间推理结果一旦出现偏差,便可能直接影响验证结论的可靠性。

从表现形式上看,LLM 在程序验证场景中的幻觉主要可分为结构幻觉与语义幻觉两类。结构幻觉指模型在分析程序结构或依赖关系时,生成了不符合实际程序结构的中间结果,例如推断出程序依赖图中并不存在的控制路径或数据依赖关系,从而导致无效程序切片或错误分析结论。此类幻觉通常源于模型对程序结构的理解依赖统计相关性而非显式语义约束,在复杂控制流或嵌套结构下尤为明显。

语义幻觉则主要体现在模型对程序逻辑含义的误解上,例如对循环不变式、边界条件或断言语义的错误推断。模型可能生成形式上合理但在语义上并不成立的推理结论,从而错误判断程序的安全性。其根本原因在于 LLM 缺乏可验证的形式语义,其推理过程无法保证与程序的真实执行行为一致。

针对上述幻觉问题,本文通过引入基于 Frama-C 的数据流切片反向分析与测试机制对 LLM 输出进行系统性校验,在一定程度上缓解了幻觉对验证结果的影响。然而,该策略主要侧重于事后过滤与回退,而非从根本上消除幻觉的产生,因此在极端复杂程序场景下仍可能带来额外的分析开销。

4.2 程序规模与结构复杂度的限制

本文方法的设计目标主要面向中等规模至较大规模的程序验证任务,在该规模范围内,通过 LLM 进行深入的程序语义分析,并结合基于 Frama-C 的数据流切片反向分析与测试机制,可以在保证验证结果高正确率的同时,有效提升验证效率。然而,当程序规模进一步增大,或其控制流结构、循环嵌套层次以及数据依赖关系高度复杂时,

LLM 的分析过程本身可能成为性能瓶颈,从而导致验证流程超时。

实验结果中出现的部分超时任务,正是由于程序结构复杂度超出了当前分析策略所能有效覆盖的范围。所以本文方法在面对结构极端复杂或语义高度交织的程序时,仍需依赖更精细的程序抽象、分层分析或复杂度控制机制,以进一步提升适用性。

4.3 计算成本与工程部署挑战

在计算成本方面,本文方法采用了一种先分析、后生成的 LLM 调用策略。该策略通过在验证流程初始阶段对程序进行较为充分的语义与结构分析,为后续各验证子任务提供稳定且一致的上下文基础,从而减少重复推理。然而,该初始分析阶段通常伴随较高的计算开销与推理时间,使整体性能在一定程度上受限于 LLM 的调用效率。

在实际工业场景中,LLM 的调用成本与推理延迟可能对持续集成环境和大规模自动化验证流程带来额外负担。模型调用的经济成本与输入和输出的 token 数量呈正相关,其中输出 token 数量可通过策略设计进行有效控制。例如,本文在验证流程中要求 LLM 对程序进行较为详细的初始分析,而在后续数据规模优化或反例生成中,仅需 LLM 输出简洁的结论性结果,从而显著降低后续调用的生成开销。然而,由于输入 token 中必须包含待验证程序本身,当程序规模较大或结构复杂时,输入 token 数量仍可能成为整体成本的主要来源。

此外,工业级软件系统往往具有多语言混合开发、并发执行以及复杂系统级交互等特征,这些因素将显著增加程序行为建模与语义分析的难度,对 LLM 的推理能力以及形式化验证工具的处理能力提出更高要求。如何在保证验证准确性的同时进一步降低模型调用成本,并提升方法在复杂工程场景下的适应性与可扩展性,仍是本文方法在工程部署层面需要重点关注的挑战。

5 结 语

本文针对传统程序验证在大规模程序中易受状态空间爆炸影响的问题,提出了数据规模优化与反例生成两种基于 LLM 的验证增强方法,并构建了融合 LLM 与传统验证工具的并行验证框架。实验结果表明,在统一资源限制下,该框架能够显著提高验证任务的解决数量,降低平均验证时间与内存开销,同时保持较高的验证准确率。该结果说明,LLM 适合作为程序验证中的语义分析与候选生

成工具,而不宜直接作为独立验证器;通过引入数据流切片、测试执行和传统验证工具兜底机制,可以在提升效率的同时增强验证结论的可靠性。

与已有主要依赖形式化工具或纯LLM推理的方法相比,本文方法的特点在于将LLM的启发式推理能力与外部验证机制结合,形成从候选生成到结果校验的闭环流程。该思路为智能化程序验证提供了一种兼顾效率与可信度的实现路径,对处理数组、循环以及可达性验证任务具有一定实用价值。

但本文方法仍主要适用于以可达性为核心的安全性验证任务,对no-overflow、termination以及并发正确性等任务的支持仍然有限。未来可进一步探索LLM与多类验证后端、程序抽象技术及反馈学习机制的结合,以提升框架在复杂工程场景中的适用性与鲁棒性。

参考文献:

- [1] SELIGMAN E, SCHUBERT T, KUMAR M V A. Formal Verification: An Essential Toolkit for Modern VLSI Design [M]. San Diego: Academic Press, 2015.
- [2] TIHANYI N, CHARALAMBOUS Y, JAIN R, *et al.* A new era in software security: Towards self-healing software *via* large language models and formal verification [C]//2025 IEEE/ACM International Conference on Automation of Software Test (AST). New York: IEEE Press, 2025: 136-147. DOI: 10.1109/AST66626.2025.00020.
- [3] TER BEEK M H, CHAPMAN R, CLEVELAND R, *et al.* Formal methods in industry [J]. *Formal Aspects of Computing*, 2024, **37**(1): 1-38. DOI:10.1145/3689374.
- [4] BASILE D, TER BEEK M H, FERRARI A, *et al.* Exploring the ERTMS/ETCS full moving block specification: An experience with formal methods[J]. *International Journal on Software Tools for Technology Transfer*, 2022, **24**(3): 351-370. DOI:10.1007/s10009-022-00653-3.
- [5] RAJABLI N, FLAMMINI F, NARDONE R, *et al.* Software verification and validation of safe autonomous cars: A systematic literature review [J]. *IEEE Access*, 2021, **9**: 4797-4819. DOI: 10.1109/ACCESS.2020.3048047.
- [6] CONTI M, DONADEL D, TURRIN F. A survey on industrial control system testbeds and datasets for security research[J]. *IEEE Communications Surveys & Tutorials*, 2021, **23**(4): 2248-2294. DOI:10.1109/COMST.2021.3094360.
- [7] CLARKE E, BIERE A, RAIMI R, *et al.* Bounded model checking using satisfiability solving[J]. *Formal Methods in System Design*, 2001, **19**(1): 7-34. DOI: 10.1023/A:1011276507260.
- [8] LYU M R, RAY B, ROYCHOUDHURY A, *et al.* Automatic programming: Large language models and beyond [J]. *ACM Transactions on Software Engineering and Methodology*, 2025, **34**(5): 1-33. DOI: 10.1145/3708519.
- [9] SEVENHUISEN M. Effectiveness of large language models to generate formally verified C code[D]. Stockholm: KTH Royal Institute of Technology, 2024.
- [10] LIU J W, XIA C S, WANG Y Y, *et al.* Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation [EB/OL]. 2023; arXiv: 2305.01210. <https://arxiv.org/abs/2305.01210>.
- [11] LIU Y, TANTITHAMTHAVORN C, LIU Y H, *et al.* On the reliability and explainability of language models for program generation [J]. *ACM Transactions on Software Engineering and Methodology*, 2024, **33**(5): 1-26. DOI:10.1145/3641540.
- [12] JI Z W, LEE N, FRIESKE R, *et al.* Survey of hallucination in natural language generation[J]. *ACM Computing Surveys*, 2023, **55**(12): 1-38. DOI:10.1145/3571730.
- [13] WALLIN T, WIDHOLM K. AI assisted validation and verification of software: Enhancing system test verification processes through large language model classification[D]. Umeå: Umeå University, 2025.
- [14] NEJATI F, GHANI A A A, YAP N K, *et al.* Handling state space explosion in component-based software verification: A review [J]. *IEEE Access*, 2021, **9**: 77526-77544. DOI:10.1109/ACCESS.2021.3081742.
- [15] BENEŠ N, BRIM L, HUVAR O, *et al.* Symbolic model checking of hybrid ctl on coloured kripke structures[C]//Automated Technology for Verification and Analysis. Cham: Springer, 2025: 212-233. DOI:10.1007/978-3-031-78750-8_11.
- [16] BERGER P. Applying software model checking: Experiences and advancements [D]. Aachen: RWTH Aachen University, 2024.
- [17] XU T. Bounded model checking and passive learning of LTL formulas [R]. Le Kremlin-Bicêtre: Laboratoire de Recherche de l'EPITA, 2025.
- [18] BIERE A, CLARKE E, RAIMI R, *et al.* Verifying safety properties of a PowerPC — microprocessor using symbolic model checking without BDDs [C]//Computer Aided Verification. Berlin: Springer, 1999: 60-71. DOI:10.1007/3-540-48683-6_8.
- [19] KROENING D, TAUTSCHNIG M. CBMC-C bounded

- model checker [C]//Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2014: 389–391. DOI:10.1007/978-3-642-54862-8_26.
- [20] BEYENE M S. Source-code summarization of java methods using control-flow graphs [D]. Saskatoon: University of Saskatchewan, 2021.
- [21] ZHANG B W, CHEN W, CHIU H C, *et al.* Unveiling the power of intermediate representations for static analysis: A survey [EB/OL]. 2024: arXiv: 2405.12841. <https://arxiv.org/abs/2405.12841>.
- [22] MALÍK V. Static analysis of C programs [D]. Brno: Brno University of Technology, Faculty of Information Technology, 2024.
- [23] GRAHAM S L, CYTRON R, FERRANTE J, *et al.* Efficiently computing static single assignment form and the control dependence graph [J]. *ACM Transactions on Programming Languages and Systems*, 1991, **13** (4) : 451–490. DOI:10.1145/115372.115320.
- [24] ALLEN F E. Control flow analysis [C]//Proceedings of a Symposium on Compiler Optimization. New York: ACM, 1970: 1–19. DOI:10.1145/800028.808479.
- [25] DENNIS J B. First version of a data flow procedure language [C]//Programming Symposium. Berlin: Springer, 1974: 362–376. DOI:10.1007/3-540-06859-7_145.
- [26] FERRANTE J, OTTENSTEIN K J, WARREN J D. The program dependence graph and its use in optimization [C]//International Symposium on Programming. Berlin: Springer, 1984: 125–132. DOI: 10.1007/3-540-12925-1_33.
- [27] RAE J W, BORGEAUD S, CAI T, *et al.* Scaling language models: Methods, analysis & insights from training gopher [EB/OL]. 2021: arXiv: 2112.11446. <https://arxiv.org/abs/2112.11446>.
- [28] BROWN T B, MANN B, RYDER N, *et al.* Language models are few-shot learners [EB/OL]. [2020-12-26]. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf.
- [29] MA L Z, LIU S Q, LI Y, *et al.* SpecGen: Automated generation of formal program specifications *via* large language models [C]//2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). New York: IEEE Press, 2025: 16–28. DOI:10.1109/ICSE55347.2025.00129.
- [30] MISU M R H, LOPES C V, MA I, *et al.* Towards AI-assisted synthesis of verified dafny methods [J]. *Proceedings of the ACM on Software Engineering*, 2024, **1** (FSE) : 812–835. DOI:10.1145/3643763.
- [31] WU H Z, BARRETT C, NARODYTSKA N. Lemur: Integrating large language models in automated program verification [EB/OL]. 2023: arXiv: 2310.04870. <https://arxiv.org/abs/2310.04870>.
- [32] CHAKRABORTY S, LAHIRI S K, FAKHOURY S, *et al.* Ranking LLM-generated loop invariants for program verification [EB/OL]. 2023: arXiv: 2310.09342. <https://arxiv.org/abs/2310.09342>.
- [33] WANG J, NI T, LEE W B, *et al.* A contemporary survey of large language model assisted program analysis [J]. *Transactions on Artificial Intelligence*, 2025, **1** (1) : 105–129. DOI:10.53941/tai.2025.100006.
- [34] WEN C, CAO J L, SU J, *et al.* Enchanting program specification synthesis by large language models using static analysis and program verification [C]//Computer Aided Verification. Cham: Springer, 2024: 302–328. DOI: 10.1007/978-3-031-65630-9_16.
- [35] KIRCHNER F, KOSMATOV N, PREVOSTO V, *et al.* Frama-C: A software analysis perspective [J]. *Formal Aspects of Computing*, 2015, **27** (3) : 573–609. DOI: 10.1007/s00165-014-0326-7.
- [36] BEYER D. State of the art in software verification and witness validation: SV-COMP 2024 [C]//Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2024: 299–329. DOI: 10.1007/978-3-031-57256-2_15.
- [37] OPENAI. GPT-4o system card [EB/OL]. 2024: arXiv: 2410.21276. <https://arxiv.org/abs/2410.21276>.
- [38] DEEPSEEK-AI, LIU A X, FENG B, *et al.* DeepSeek-V3 technical report [EB/OL]. 2024: arXiv: 2412.19437. <https://arxiv.org/abs/2412.19437>.
- [39] POST H, SINZ C, KAISER A, *et al.* Reducing false positives by combining abstract interpretation and bounded model checking [C]//2008 23rd IEEE/ACM International Conference on Automated Software Engineering. New York: IEEE Press, 2008: 188–197. DOI:10.1109/ASE.2008.29.
- [40] LI Y J, CHOI D, CHUNG J, *et al.* Competition-level code generation with AlphaCode [J]. *Science*, 2022, **378** (6624) : 1092–1097. DOI:10.1126/science.abq1158.

□