arXiv:2512.15816v1 [cs.PL] 17 Dec 2025

一种通过最弱前置条件推理进行循环不变式生成的神经符号方法

DaraghKing[1] ✉计算机科学与统计学系，都柏林圣三一学院，都柏林，爱尔兰

Vasileios Koutavas ✉
Lero, 爱尔兰软件研究中心
计算机科学与统计学系，都柏林圣三一学院，都柏林，爱尔兰

Laura Kovács ✉
维也纳工业大学信息学学院，维也纳，奥地利

摘要

循环不变式生成仍然是自动化程序验证中的关键瓶颈。近期研究开始探索在这一领域使用大语言模型（LLM），但这些方法往往缺乏可靠且结构化的方法论，对现有程序验证理论的借鉴也很少。本文提出了 NeuroInv，一种用于循环不变式生成的神经符号方法。NeuroInv 包含两个关键模块：（1）神经推理模块，利用大语言模型和霍尔逻辑，通过反向链式最弱前置条件推理来推导和完善候选不变式；（2）验证引导的符号模块，利用来自 OpenJML 的反例迭代修复不变式。我们在一个包含 150 个 Java 程序的综合基准测试上评估了 NeuroInv，这些程序涵盖单循环和多重（顺序）循环、多个数组、随机分支以及含噪代码片段。NeuroInv 达到了 99.5% 的成功率，显著优于其他受评估的方法。此外，我们还引入了一个包含 10 个更大型多循环程序的困难基准测试（每个程序平均包含 7 个循环）；NeuroInv 在这一设置下的表现表明，它能够扩展到更复杂的验证场景。

2012 年 ACM 学科分类 软件及其工程 → 形式化软件验证；计算理论 → 程序验证；计算方法 → 人工智能

关键词与短语 循环不变式、程序验证、霍尔逻辑、神经符号人工智能、大语言模型

1 引言

循环不变式生成是自动化程序验证中的一项核心挑战。在霍尔逻辑中， [12],循环不变式是用于证明循环计算正确性的归纳断言；循环不变式必须满足以下条件：(a) 在循环开始前成立，(b) 在每次循环迭代中保持成立，以及 (c) 在退出时蕴含循环后置条件（有关霍尔逻辑和循环不变式的更多细节，请参见第 1.1 节）。尽管概念上很简单，寻找合适的循环不变式却是一个不可判定问题，而且众所周知，启发式方法难以解决这一问题。传统静态技术——包括抽象解释 [4],逻辑溯因 [1], 符号执行 [21], 和递推求解 [13]——在狭窄领域中表现良好，但往往难以推广到结构更复杂的程序，例如包含多个循环的程序。诸如 [7]之类的动态方法可以从程序执行中推断循环不变式，但其可靠性从根本上受执行轨迹的数量和质量限制。

大语言模型（LLM）的出现，促使近期研究开始探索它们能否生成程序性质，例如最弱前置条件 [17, 16], 后置条件——

[1] 通讯作者

以及 [6], 甚至循环不变式 [23, 14]。尽管这些早期研究展现出了一定潜力，但现有的基于大语言模型的循环不变式生成方法通常依赖临时提示，且未能与程序验证理论进行系统性对齐。此外，由于这些方法缺乏有原则的机制来引导循环不变式的细化，生成的不变式往往较为脆弱（在语法和语义层面皆是如此）——对于简单的整数循环，这些不变式或许足够，但对于涉及数组、多重循环，或含有噪声及语义无关代码的程序，则不可靠。

为解决基于大语言模型的循环不变式推断中的这些局限，我们提出 NeuroInv，这是一种用于循环不变式生成的神经符号 [10] 框架，它将大语言模型与最弱前置条件（WP）推理 [5]以及反例引导的符号验证相结合。NeuroInv 由两个互补模块构成。神经模块将输入程序划分为无循环部分和含循环部分，随后利用大语言模型完成三项主要任务：(1) 计算无循环代码段的最弱前置条件；(2) 生成候选循环不变式；(3) 根据霍尔逻辑（第 1.1 节）定义的保持义务和退出义务检查并细化候选循环不变式——当上述义务未能满足时，该模块会尝试有针对性地细化候选循环不变式。神经模块采用的这一过程形成了一套结构化且由理论指导的推理流程，与临时提示方法 [14]形成对比。

符号模块通过使用 OpenJML 验证神经模块推导出的不变式，在验证失败时提取反例，并尝试通过反馈引导的过程修复不变式，从而强化上述方法。这种双层细化与修复方法使 NeuroInv 能够可靠地生成正确的循环不变式。

我们在一个包含 150 个 Java 程序的综合基准测试上评估了 NeuroInv，涵盖单个循环和多个（顺序执行的）循环、多个数组、随机分支以及含噪代码片段。NeuroInv 的成功率达到 99.5%，显著优于临时提示、纯神经消融实验，以及最先进的、基于大语言模型的 Java 规格生成器 [19]。对一个由 10 个规模较大的多循环程序组成的困难基准测试进行的评估进一步表明，NeuroInv 能够有效扩展至更复杂的程序，而神经模块在处理更长、更复杂的推理链方面发挥着关键作用。总的来说，这些结果表明，将系统化的最弱前置条件推理与验证反馈相结合，能够显著增强大语言模型辅助不变式生成的稳健性和可靠性。

本文其余部分的结构如下。第 1.1 节概述了理解 NeuroInv 所需的霍尔逻辑 [9, 12], 、最弱前置条件 [5], 和循环不变式背景知识。第2节介绍 NeuroInv 背后的方法，第 3 节则概述我们的实验评估及所得结果。第 4 节讨论有效性威胁和未来工作。第 5 节介绍相关工作，第 6 节作出总结。

1.1 霍尔逻辑、最弱前置条件与循环不变式

霍尔逻辑为推理命令式程序的正确性提供了一个形式化系统。这种逻辑的核心单元是霍尔三元组（定义 1）。霍尔三元组从根本上依赖于前置条件和后置条件的概念：后置条件规定程序终止时程序状态应满足的性质，而前置条件规定初始程序状态应具备的性质，以确保程序按预期运行（即满足程序的后置条件）。根据是否证明终止性，该逻辑分为完全正确性版本和部分正确性版本。本文聚焦于部分正确性。

▶ **定义 1**（霍尔三元组（部分正确性））。 霍尔三元组 $\{P\}\ C\ \{Q\}$ 断言：如果前置条件 P 在执行命令 C之前成立，并且 C 终止，那么

后置条件 Q在执行后成立。

对霍尔三元组进行推理可以实现组合式程序验证，但要有效地做到这一点，必须仔细考量前置条件的概念。考虑霍尔三元组 $\{P\}\ C\ \{Q\}$，如果 P 过于严格（或强），那么它可能会排除一些初始状态，而这些状态实际上在 C 执行并终止后会满足 Q 。 [5] 已有研究证明，对于程序 C 和后置条件 Q，存在唯一的最大前置条件，称为最弱前置条件 （WP）。

▶ **定义 2 (**最弱前置条件**)**。 最弱前置条件 $WP(C, Q)$ 是最一般的条件：如果在执行命令 C 之前该条件成立，就能保证后置条件 Q在执行后成立（假设程序会终止）。

最弱前置条件演算 [5] 为霍尔三元组提供了一种证明技术：要证明 $\{P\}\ C\ \{Q\}$，必须计算 WP(C, Q)，并证明 $P \Rightarrow$ WP(C, Q)。这意味着，任何满足 P 的初始状态也满足 WP(C, Q)；执行 C后，最终状态将满足 Q。

▶**定义 3** （初始化）。

$$P \Rightarrow \text{最弱前置条件}(C, Q) \tag{1}$$

关键在于，弱前置条件演算提供了组合规则，用于计算最弱前置条件。该演算的一条核心规则指出，顺序命令的最弱前置条件以反向方式组合：对于命令序列 $C_1; C_2$， C_2 的最弱前置条件（相对于总体后置条件 Q）充当 C_1 的直接后置条件（定义 4）。（其他规则则用于组合分支命令和赋值命令的最弱前置条件。）

▶**定义 4** （序列的最弱前置条件）。

$$WP(C_1; C_2, Q) = WP(C_1, WP(C_2, Q)) \tag{2}$$

然而，对于循环，计算最弱前置条件是不可判定的，因此弱前置条件演算依赖于外部提供的循环不变式。一般而言，循环不变式是指在每次循环迭代之前和之后都成立的断言。然而，在弱前置条件演算的语境中，足够的循环不变式还必须满足以下两个条件：

▶**定义 5** （保持）。 如果不变式 I 成立且循环条件 B 为真，那么执行循环体 C 后必须仍保持该不变式。等价地：

$$I \wedge B \Rightarrow WP(C, I) \tag{3}$$

▶**定义 6** （退出）。 当不变式 I 成立且循环退出时（即 B 为假），必须满足循环的后置条件 Q 。

$$I \wedge \neg B \Rightarrow Q \tag{4}$$

这些条件与公式（1）和最弱前置条件的组合规则相结合，确保弱前置条件演算能够在霍尔逻辑框架内正确构建验证证明。在第 2.1 节中，定义5和定义6中的两个条件在循环不变式细化过程中发挥着重要作用。此外，定义4构成了神经模块推理的基础。

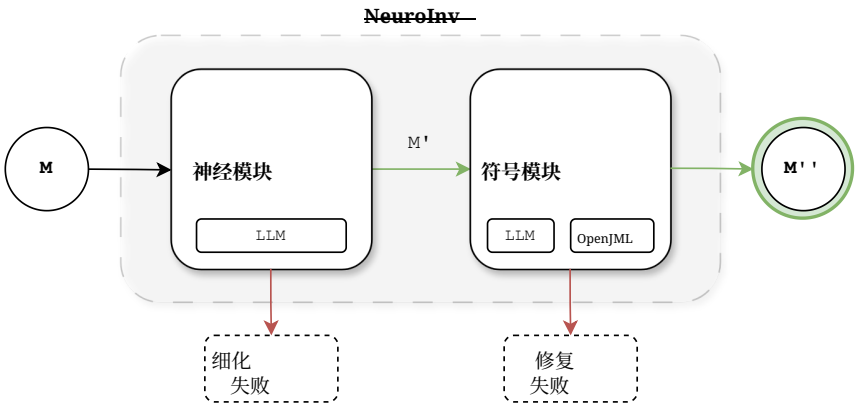

图1 NeuroInv 的高级架构； M 是输入程序， M' 是包含神经推导循环不变式的输入程序，而 M'' 表示带有正确循环不变式（以 JML 注解形式表示）的 M 。

算法1 神经符号循环不变量生成

要求： 带有 JML 前置条件 Pre 和后置条件 Post 的 Java 方法 M **确保：** 带有循环不变式的已验证方法 M' 1：**模块 1：神经推理（算法2）**2：不变式 $\leftarrow$ NeuralReasoningModule(M，Pre，Post) 3：4：**模块 2：符号验证（算法 3）** 5： $M' \leftarrow$ SymbolicVerificationModule(M，不变式，Pre，Post) 6：**返回** M'

2 方法

我们的方法 NeuroInv 旨在生成充分的循环不变式，以根据给定的前置条件和后置条件验证 Java 方法的正确性。该系统采用双模块神经符号架构，将最弱前置条件（WP）推理与符号验证相结合。

神经模块： 给定一个带有 JML 前置条件和后置条件的 Java 方法，神经模块会尝试通过反向链式最弱前置条件推理，推导出高质量的候选循环不变式。该模块利用大语言模型完成三项主要任务：(1) 计算无循环代码段的最弱前置条件；(2) 生成候选循环不变式；(3) 根据定义5和定义6中规定的逻辑蕴含，检查并完善候选循环不变式。

符号模块： 对于神经模块推导出的候选循环不变式，符号模块会将其转换为 JML 注解，并尝试通过 OpenJML 进行程序验证。如果验证失败，该模块会从 OpenJML 的输出中提取反例信息，并尝试利用这些信息修复候选循环不变式。此过程会持续迭代，直到验证成功或达到最大修复上限。

我们神经符号设计的核心洞见是双层细化与修复：神经模块采用的细化 过程旨在确保以符合霍尔逻辑和弱前置条件演算的方式生成候选循环不变式。符号模块的修复 过程则利用验证反馈，根据神经模块未能发现的错误来修复循环不变式。

图1展示了 NeuroInv 的高层架构。算法1描述了

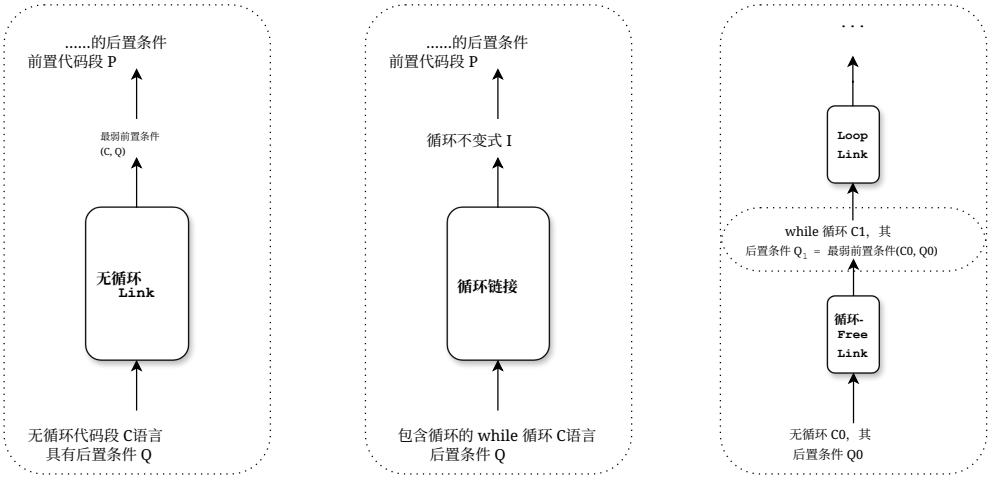

图2 最左侧的两个实体展示了神经模块在反向链接最弱前置条件推理中使用的链接类型。右侧的实体展示了“无循环”链接与“循环”链接之间的链，对于一个假设的程序段，该程序段由一个 `while` 循环和其后的无循环代码组成。

整体方法，而算法2和算法3分别展示了神经模块和符号模块的细节。此外，附录A包含 NeuroInv 的提示词。

2.1 神经模块

神经模块通过反向链式最弱前置条件推理生成候选循环不变式。此过程依赖大语言模型完成三项核心任务：(1) 计算无循环代码段的最弱前置条件；(2) 生成候选循环不变式；(3) 根据定义5和定义6的逻辑蕴含检查并改进候选循环不变式。此过程遵循下文的算法2。

2.1.1 程序分段

在尝试任何推理之前，神经模块会先将输入程序拆分为循环段和无循环段。每个代码段都用起始和结束标记注释标记，例如：

`// code N open` … `// code N close`：第 N 个无循环代码段。`// while N open` … `// while N close`：第 N 个 `while` 循环。这种分段方式便于系统化处理：标记注释使所需代码段能够通过程序化方式提取。

2.1.2 反向链接 WPReasoning

从程序给定的后置条件出发，神经模块沿着分段程序（第 2.1.1 节）向后推理，将最弱前置条件计算与循环不变式推导串联起来。

从概念上讲，前述链由两种链接类型组成，每种链接类型都与特定类型的代码段和一组基于大语言模型的计算相关联：1. **无循环链接：**

算法2 神经模块：通过反向链式最弱前置条件推理进行循环不变式生成 要求：方法 M、前置条件 Pre、后置条件 Post 确保：候选循环不变式；如果达到细化上限，则返回错误 1：阶段1：程序分段 2：代码段, 循环 ← SegmentProgram(M) 3： 4：阶段2：反向链式最弱前置条件推理 5：Post ← ExtractPostcondition(M) ▷ 来自 JML 后置条件 6：currentPost ← Post 7：invariants ← {} 8：如果 ∃ 所有循环之后的最终代码段 则 9：currentPost ← LLM-ComputeWP(finalSegment, currentPost) 10：结束如果 11：对于 loopIndex ← |loops| 从下至1执行 ▷ 按逆序处理循环 12：currentLoop ←loops[loopIndex]13：loopCondition ← condition(currentLoop) 14：loopBody ← body(currentLoop) 15：loopPost ←currentPost 16：invariant ← LLM-GenerateInvariant(currentLoop, loopPost) 17：validated ←false 18：对于 attempt ← 1 至 MAX_REFINEMENT 执行 19：weakestPre ← LLM-ComputeWP(loopBody, invariant) 20：preservation ← LLM-CheckImplication(invariant ∧ loopCondition ⇒ weakestPre) 21：exit ← LLM-CheckImplication(invariant ∧ ¬loopCondition ⇒ loopPost) 22：如果 preservation ∧ exit 则 23：validated ← true 24：跳出 ▷ 两个蕴含均成立 25：否则 26：invariant ← LLM-RefineInvariant(invariant, preservation, exit) 27：结束如果 28：结束循环 29：如果 ¬validated 则 30：抛出错误 ▷已达到细化上限 31：结束如果 32：invariants[currentLoop] ← invariant 33：currentPost ←invariant ▷ 不变式成为前一代码段的后置条件 34：如果 ∃ 当前循环之前存在代码段 则 35：precedingSegment ←codeSegments[loopIndex]36：currentPost ← LLM-ComputeWP(precedingSegment, currentPost) 37：结束如果 38：结束循环 39： 40：前置条件合理性检查 41：entailment ← LLM-CheckImplication(Pre ⇒ currentPost) ▷ 非阻塞 42：返回 invariants

---

- **代码段类型：**无循环。**计算：**给定后置条件Q，计算无循环代码段的最弱前置条件。
-

2. 循环链接：代码段类型： `while`循环。计算（序列）： a. 根据循环后置条件 Q生成候选循环不变式 I。b. 将循环不变式 I 作为后置条件，计算循环体 S 的最弱前置条件，即计算 $WP(S, I)$。c. 验证 I 是否满足定义 5。d. 验证 I 是否满足定义 6。e. 如果步骤 c 或 d 失败，则细化 I；若发生这种情况，序列将从步骤 b重新开始，并使用细化后的循环不变式 I。

算法 3 符号模块：基于反例引导修复的验证

要求： 方法 M，候选不变式 invs，前置条件 Pre，后置条件 Post
确保： 已验证方法 M' 或错误
阶段 3：最终化与验证
$M' \leftarrow$ LLM-转换为 JML(M，invs) ▷ 转换为 JML 语法
对repairAttempt $\leftarrow 1$ **进行**MAX_修复 **直到**
  ⟨已验证，反例⟩ $\leftarrow$ OpenJML-Verify(M')
  如果 已验证**则**
    返回 M' ▷ 已成功验证
  结束 if
  不变式 $\leftarrow$ LLM-修复不变式(不变式，反例)
  $M' \leftarrow$ LLM-转换为 JML(M，不变式)
结束 for
返回错误 ▷ 在修复上限内无法验证

图 2 展示了两种链接类型；这两种链接的最终目标都是为前置代码段（如果存在）生成后置条件。通过将这些链接串联起来，神经模块可以从程序给定的后置条件出发，反向推导循环不变式。在此过程结束时，神经模块会检查推导出的最后一个最弱前置条件是否满足定义 3。

2.2 符号模块

符号模块在概念上较为简单，可概括为以下步骤：
1. 将神经模块生成的循环不变式转换为相应的 JML 注解。**2.** 尝试使用 OpenJML 验证带注解的程序。**3.** 如果步骤 2 失败，则从 OpenJML 中提取反例信息，并利用这些信息修复循环不变式。随后从步骤 2 开始，使用修复后的循环不变式重复该过程，直到 (a) 验证成功，或 (b) 达到最大修复上限。

算法 3 对该过程进行了详细说明。

3 评估

3.1 实验设置

以下各节概述了实验评估的基本组成部分。第 3.1.1 节介绍基准测试套件。第 3.1.2 节介绍所评估的方法。第3.1.3节介绍实现细节。最后，第 3.1.4 节详细说明所使用的评估指标。

3.1.1 基准测试套件

我们在评估中使用的基准测试套件分为两个子集：(1) 标准基准测试和 (2) 困难基准测试。这些子集中的程序具有共同属性，每个程序都具备：(a) 使用 JML 前置条件和后置条件进行注释的单个 Java 方法，以及 (b) 至少一个 `while` 循环。 [2]

[2] 标准基准测试和困难基准测试都将存档于 Zenodo。

表 1 标准基准测试的组成和关键特征。

类别	数量	数组	多重循环	含噪	随机分支	∀-不变式	∃-不变式
1.	62	✗	✗	✗	✗	✗	✗
2.	35	✓	✗	✗	✗	✓	✗
3.	21	✓	✓	✗	✗	✓	✗
4.	15	✓†	✗	✓	✗	✓†	✗
5.	10	✓	✗	✗	✓	✓	✗
6.	5	✓	✗	✗	✗	✗	✓
7.	2	✓	✗	✗	✗	✓	✗
总计	150						

†只有一部分含噪程序基于数组，需要全称量化不变式（即源自类别 2 的不变式）。

标准基准测试。 标准基准测试包含 150 个程序，按以下高层类别进行划分（汇总见表 1）：
1. 62 个程序是将源自 Code2Inv [26]数据集的程序翻译为 Java 后得到的单循环程序；选择这些 62 个程序，是因为它们的循环守卫中不包含非确定性。**2.** 35 个程序是源自 [16] 工作的数组操作单循环程序；这些 35 个程序可能涉及多个数组，并且需要使用全称量化的循环不变式。**3.** 21个程序是通过对上述第 2 点所述程序的一个子集进行保语义的多重循环分解得到的。**4.** 15 个程序是通过对上述第 1 点和第 2 点所述程序的一个子集进行保语义的“含噪”转换得到的。这些程序包含无用代码、无关代码或不可达代码，同时（可能）使用了具有误导性的变量名。**5.** 10 个程序是将源自 [17] 的程序翻译为 Java 后得到的单循环程序；这些程序各自在循环内包含随机分支。**6.** 5 个程序是源自 [16] 工作的数组操作单循环程序；这些 5 个程序可能涉及多个数组，并且需要使用存在量化的循环不变式。**7.** 2 个程序不属于上述类别，其中一个程序用于反转给定数组，另一个程序用于检查两个数组是否相等。

困难基准测试。 困难基准测试包含 10 个程序，专门用于对所评估的方法（第 3.1.2 节）及其可扩展性进行压力测试。该基准测试的特征见第 3.3 节。

3.1.2 评估的方法

我们在实验中评估了四种方法：临时提示、NeuroInv（第2节）、NeuroInv* 和 SpecGen [19]。

临时提示： 该方法直接要求大语言模型为给定程序生成正确的循环不变式；过程中不提供任何带标注的示例，因此可视为一种零样本提示方法 [25]。所使用的提示词见附录 A 的清单 1。

NeuroInv： 这是我们提出的用于循环不变式生成的神经符号方法，详见第2节。

NeuroInv*：这种方法仅由第 2.1 节概述的神经模块组成。我们将这一“仅神经模块”变体作为消融实验，以便更好地了解神经模块和符号模块各自的贡献。请注意，NeuroInv 包含 NeuroInv*，因此两种方法在后续章节中会有一些相同的结果。

SpecGen：SpecGen [19]是一种较新的基于大语言模型的方法，可自动生成完整的程序规格（包括循环不变式、前置条件和后置条件）。因此，SpecGen 的目标比单独生成循环不变式更为广泛——它还可能修改给定的 JML 前置条件和后置条件，因为它最初是为未指定程序而设计的（实验中，我们发现标准基准测试中约 68% 的程序都出现了这种情况）。在我们的设置中，对于第 3.2 节报告的统计数据，我们将此类情况视为失败，因为此时我们无法再保证经验证的前置条件和后置条件与原始条件等价。尽管如此，我们仍将 SpecGen 纳入评估，因为它是与我们的方法最为接近的相关方法：SpecGen 面向 Java，生成基于 JML 的规格说明，并且能够生成循环不变式——据我们所知，目前尚无更适合 Java 的基于大语言模型的循环不变式生成器。此外，研究表明，SpecGen 在规格生成方面优于其他相关方法，例如 AutoSpec [27], Daikon [7], 和 Houdini [8]。

3.1.3 实现细节

我们的实验采用以下配置。

- **实验协议：** 我们完成5 次独立实验运行，采用第 3.1.2 节所述的方法，并分别对标准基准测试和困难基准测试进行评估。**模型：** 我们使用 `o4-mini` [22] 作为每种受评估方法（包括 SpecGen）的底层模型。在撰写本文时， `o4-mini` 是 OpenAI 推理模型中性能最高且价格最实惠的模型。**OpenJML 和求解器：** 我们使用 `openjml 21-0.11` ，并将 `Z3 version 4.13.4 - 64 bit`用作其底层求解器。**细化与修复参数：** 1. `MAX_REFINEMENT`：指神经模块针对给定程序中的每个循环可进行的最大细化尝试次数。我们设置为 `MAX_REFINEMENT`=5。2. `MAX_REPAIR`：指符号模块对每个程序可进行的最大修复尝试次数。我们设置为 `MAX_REPAIR`= 5

3.1.4 评估指标

以下指标构成了我们分析的基础。

- **成功率：** 这表示生成的循环不变式能够成功验证的程序所占的百分比。表 2 中的“平均值”成功率表示在标准基准测试中对所有程序进行 5 轮实验运行所得到的成功率均值（即总计 750 次程序评估）。**标准差：** 我们用它来衡量多轮实验运行中“成功率”结果的稳定性。**Pass@k：** 该指标量化了对于给定问题 [2]，至少有一个解（在 k个解中）正确的概率。我们按程序使用此指标，以观察

表 2 在 5 次实验运行中获得的总体标准基准测试结果。

方法	平均值成功率 (%)	标准差 (%)	平均值细化迭代次数	平均修复迭代次数	Avg. Pass@3	Avg. Pass@5
临时的	66.4	3.11	-	-	0.87	0.91
SpecGen†	28.5	2.47	-	-	-	-
NeuroInv*	84.7	2.31	0.59	-	0.95	0.97
NeuroInv	99.5	0.87	0.59	0.24	1	1

†有关 SpecGen 的注意事项，请参见第 3.1.2 节。

增加 k 对至少一个为程序生成的k 个循环不变式（或在多重循环情况下，循环不变式集合）能够成功验证的概率有何影响。表 2 中所示的“平均值” pass@k 值表示标准基准测试中 pass@3 和 pass@5 值的均值。

- **细化迭代次数：** 这表示神经模块针对每个程序细化候选循环不变式的平均尝试次数，该数值基于一组程序计算得出，例如标准或困难基准测试。表 2 中的“平均值”细化迭代次数表示在标准基准测试上进行 5 次实验运行所观察到的细化迭代次数的平均值。

- **修复迭代次数：** 这表示符号模块针对每个程序修复循环不变式的平均尝试次数，该数值基于一组程序计算得出，例如标准或困难基准测试。表 2 中的“平均值”修复迭代次数表示在标准基准测试上进行 5 次实验运行所观察到的修复迭代次数的平均值。

3.2 主要结果

3.2.1 总体性能

表 2 和图 3 展示了在标准基准测试上取得的总体结果。

从观察到的平均成功率来看，NeuroInv 明显优于其他方法；其成功率高达 99.5%，远超 SpecGen（28.5%）和临时提示（66.4%）。尽管 NeuroInv*（84.7%）的表现优于 SpecGen 和临时提示，但仍不及完整的 NeuroInv 方法。这表明，加入符号模块能够生成质量更高的循环不变式；也就是说，OpenJML 提供的反例反馈实际上有助于修复候选循环不变式。值得注意的是，SpecGen 表现相对较差，这可能是因为它旨在生成完整的程序规格，而非专门生成循环不变式。NeuroInv 与 SpecGen 之间的性能差距表明，在给定前置条件和后置条件的情况下，采用一概而论的方法处理程序规格，无法生成复杂的循环不变式。

从结果稳定性来看，NeuroInv 的成功率标准差最低。结合其更高的平均成功率，这两项指标表明 NeuroInv 能够持续生成正确的循环不变式。NeuroInv 的平均 Pass@3 得分为 1，进一步支持了这一结论（这意味着，在任意 3 次实验运行中，对于任何给定的标准基准测试程序，NeuroInv 至少有一次能够找到正确的循环不变式）。尽管 NeuroInv* 的稳定性不如 NeuroInv，但其标准差仍低于临时提示和 SpecGen；这表明

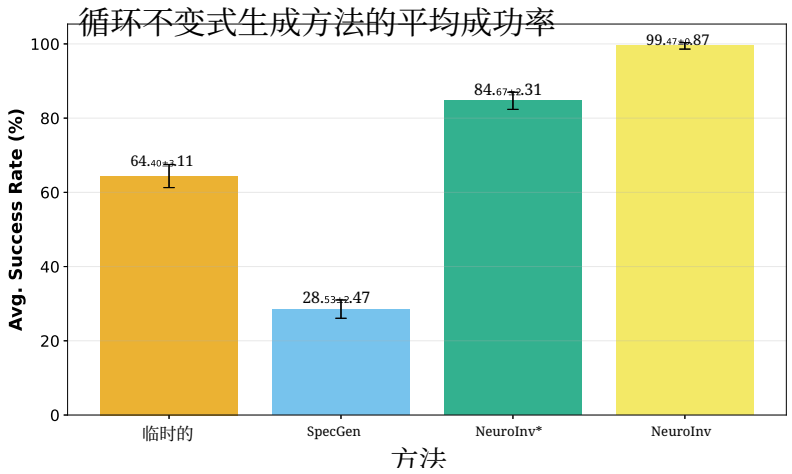

图3 在 5 次实验运行中得到的平均成功率（带标准差误差线）。

神经模块采用的系统化方法能够持续生成正确的循环不变式。

要点： 在我们的基准测试中，NeuroInv 能够稳定地生成正确率最高的循环不变式（99.5%）；SpecGen 表现最差（28.5%），甚至不如一种专门用于循环不变式生成的临时提示方法（66.4 %）。

3.2.2 细化结果

本节介绍 NeuroInv 的神经模块（即 NeuroInv*）在标准基准测试中得出的循环不变式细化结果。

请回忆，当大语言模型判定候选不变式不满足定义 5（即蕴含 1 失败）或定义 6（即蕴含 2 失败）中的至少一项时，神经模块就会进行循环不变式细化。请注意，蕴含 1 失败会排除记录蕴含 2 失败的可能性——这是当前实现的副作用。

图 4 展示了需要（以及不需要）进行循环不变式精化的程序比例；这些图表的稳定性表明，需要进行循环不变式精化的程序数量相对稳定。考虑到“细化总数”图（即虚线），这种稳定性多少有些出人意料；该统计量的最小值和最大值分别为 77 和 102，相比之下显得更加波动。综合来看，这些观察结果表明，需要精化的程序数量与所需精化总数之间并无强相关关系。

图 5 展示了由蕴含 1 和蕴含 2 失败所引发的精化比例；每个实验轮次中，两种失败类型的比例都接近 50:50。

要点： 规模稳定的程序子集会触发循环不变式细化；触发细化的逻辑失败类型在蕴含 1 失败和蕴含 2 失败之间均匀分布；而所需的精化次数在不同实验运行中差异显著。

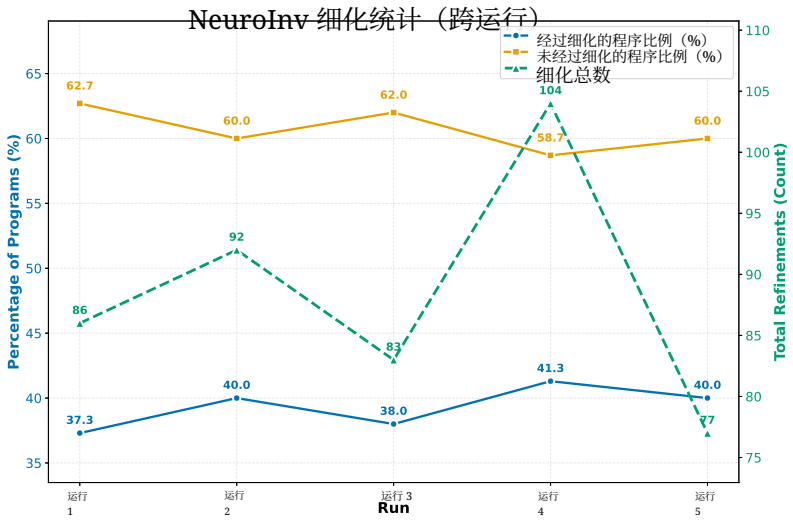

图 4 每个实验轮次中需要（和不需要）进行循环不变式细化的程序比例（实线）。此外，虚线表示每个实验轮次所需的细化总数。

3.2.3 修复结果与消融分析

为了理解神经符号方法中的各个模块如何共同生成正确的循环不变式，我们进一步讨论 NeuroInv*（即神经模块）和 NeuroInv（神经模块与符号模块的组合）。

标准基准测试的总体结果（表 2）显示，NeuroInv* 和 NeuroInv 的平均成功率分别为 84.7% 和 99.5%。换言之，加入符号模块使平均成功率提高了 +14.8%。另一方面，请注意，当神经模块生成的循环不变式（即 NeuroInv*）无法借助 OpenJML 实现程序验证时，符号模块会执行循环不变式修复（第 2.2 节）。鉴于 NeuroInv* 的平均成功率为 84.7%（表 2），这意味着绝大多数情况下都不需要符号模块进行不变式修复。表 2 中列出的平均修复迭代次数（0.24）也进一步证实了这一点。

此外，表 2 中的 Pass@5 分数表明，存在一部分程序，NeuroInv* 始终无法推导出正确的循环不变式，而 NeuroInv 至少成功推导出一次。我们将这组程序称为“问题”集。

经调查，我们发现问题集包含 5 个程序，并展现了纯神经方法的一项关键局限。对于问题集中的每个程序，神经模块生成的候选循环不变式都未能通过 OpenJML 验证——每种情况下，原因都是该不变式在循环开始前不成立。事实上，神经模块也能够指出这一局限：它在推理过程结束时检查了定义 3 的蕴含。不过，此时发现问题已经为时过晚——当前神经模块的实现无法有效回溯其推理过程，因此，OpenJML 和符号模块提供的反馈对于生成正确的循环不变式至关重要。

要点： NeuroInv 纳入符号模块后，平均成功率比 NeuroInv* 提高了 +14.8%，而符号模块对于生成

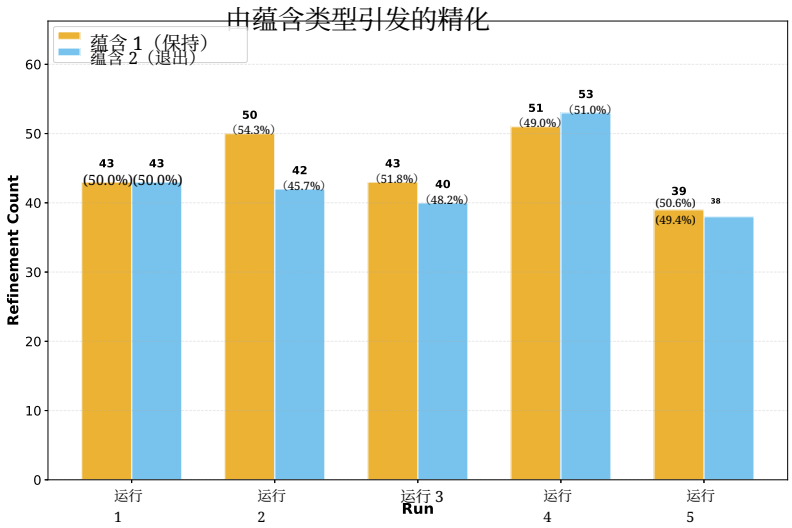

图 5 蕴含 1 和蕴含 2 失败所引发的精化比例，基于 5 次实验运行进行评估。

表 3 困难基准测试的组成与主要特征。

程序	# 循环	# 数组	∀-不变式	∃-不变式	含噪	代码行数
1	2	0	✗	✗	✗	29
2	3	0	✗	✗	✗	43
3	4	0	✗	✗	✗	56
4	5	0	✗	✗	✗	77
5	4	2	✓	✗	✗	62
6	4	3	✓	✓	✗	62
7	8	1	✓	✗	✗	78
8	8	3	✓	✓	✗	95
9	10	2	✓	✗	✓	152
10	12	2	✓	✗	✗	117

一小部分标准基准测试的正确循环不变式至关重要。

3.3 简要可扩展性分析：困难基准测试

为了测试 NeuroInv 对复杂程序的可扩展性，我们创建了一个由 10 个程序组成的困难基准测试；表 3 展示了这些程序的特征。按照标准基准测试评估的流程（即采用第3.1.3节的实现细节），我们对临时提示（第 3.1.2 节）和 NeuroInv（及其包含的 NeuroInv*）进行了 5 次实验运行 —— 鉴于 SpecGen 在标准基准测试（第 2 节）上的表现，我们在此不对其进行评估。表4呈现了这些结果。

NeuroInv 的平均成功率为 62%，而临时提示和 NeuroInv* 的成功率分别只有 14% 和 40%——NeuroInv 的成功率比临时提示高出 +48个百分点。这些结果表明，NeuroInv 能够扩展到困难程序（尽管与标准基准测试中的程序相比，性能有所下降），而临时提示则无法做到这一点。这些结果还凸显了以下方法仍然具有价值：

表4 在 5 次实验运行中获得的困难基准测试总体结果。

方法	平均值成功率 (%)	标准差 (%)	平均值细化迭代次数	平均修复迭代次数
临时的	14	5.48	-	-
NeuroInv*	40	7.07	2.9	-
NeuroInv	62	10.95	2.9	0.54

NeuroInv* 在此场景中采用的系统化方法（即 NeuroInv 的神经模块）。

此外，神经模块显然执行了更多的精化；表4中的平均值为 2.9，与标准基准测试中对应的数值（0.59）相比，几乎增加了5倍——这可能是由于困难基准测试中每个程序的平均循环数量大幅增加。相比之下，符号模块执行修复的平均次数仅增加了约2倍（0.54 对比 0.24）；经调查，我们发现绝大多数失败（94%）发生在神经模块执行不变式细化时（即达到特定循环的最大精化次数后，进程被中止）——因此，符号模块执行不变式修复的机会更少。

要点： NeuroInv 能够扩展到复杂程序，并以显著优势成功应对临时提示词（高出 +48 个百分点）；将 NeuroInv 应用于困难基准测试程序，会导致神经模块执行更多的循环不变式细化，而观察到的绝大多数失败都发生在这一过程中（94%）。

4 讨论

4.1 有效性威胁

本研究有效性面临的主要威胁源于我们对大语言模型的使用。大语言模型的输出本质上具有随机性，因此本研究的结果无法完全复现——我们通过针对每项评估进行多次实验运行，尽量降低这一威胁。数据泄漏也构成类似威胁：当测试集中的信息出现在大语言模型的训练数据中时，就会发生数据泄漏。为缓解这一具体威胁，我们的基准测试程序中有很大一部分取自 `o4-mini` 知识截止日期之后的研究成果（即 2024 年 6 月 1 日之后）；此外，基准测试中源自该日期之前的程序，均由其原始编程语言翻译而来（例如，Code2Inv 衍生基准测试程序由 C语言翻译为 Java，见第 3.1.1 节）。另一个威胁来自本研究仅使用 `o4-mini`；有人可能会认为，由此得到的结果存在偏差，而且其他供应商提供的其他模型与所评估的方法结合使用时，表现可能显著更好或更差（见第 3.1.2 节）。

4.2 未来工作

本研究自然引出了若干未来工作的方向。

构成神经模块反向链接式最弱前置条件推理的链接（在第 2.1 节中介绍）目前仍然脆弱。换言之，目前并没有对这些链接所产生的内容作出强有力的保证；例如，无循环链接类型（图 2）生成的最弱前置条件并未经过任何形式的正式验证——我们

假设，对于给定程序，如果最终的循环不变式正确，那么构成它的最弱前置条件也很可能正确。使用未经验证、由大语言模型生成的最弱前置条件并非毫无依据；此前的研究，例如 [17] 和 [16] 表明，推理模型通常无需额外的工具干预或反馈，就能生成正确的方法级最弱前置条件。在不变式生成过程中更早地引入验证器，以减轻这些链接的脆弱性，将有助于将潜在错误局限在局部；我们将这项改进留待未来工作。

类似地，第 3.2.3 节指出了神经模块固有的一项缺陷。当前的实现无法在发现为程序生成的最终最弱前置条件不满足定义 3 时，沿着推理过程回溯。实现这种有效回溯的能力是未来工作中一个有价值的方向；第 3.2.3 节的结果表明，解决这一问题将使神经模块能够有效生成更多正确的循环不变式，而无需 OpenJML 提供反馈。

扩充并标准化困难基准测试（第 3.3 节）是未来工作的另一个相关方向；据我们所知，目前还不存在类似的、针对困难的（即包含 2+个涉及多个数组等情况的循环）基于 Java 的循环不变式生成问题的基准测试。

5 相关工作

循环不变式生成是自动化程序验证中的核心挑战。尽管理论上的局限意味着不可能存在完整的解决方案，但此前已有大量工作通过各种方法来应对这一问题。

5.1 传统技术

我们将传统技术定义为不采用基于学习的方法或依赖大语言模型的技术；这些技术涵盖动态方法和静态程序分析的整个范围。

动态方法试图从程序执行轨迹中推断循环不变式。Daikon [7]，是一种面向 Java 的流行不变式生成工具，试图根据给定程序的测试套件轨迹推导不变式（包括前置条件和后置条件）。尽管功能强大，Daikon 天生受限于给定测试套件的质量和覆盖率；因此，Daikon 生成的不变式可能并非普遍成立。

另一方面，静态方法试图在不执行程序的情况下对循环不变式进行推理；与动态方法相比，这些技术通常依赖坚实的理论基础，并能提供更强的保证。在这一领域中，已经发展出众多技术并应用于循环不变式生成，例如抽象解释 [4]，逻辑溯因 [1]，递推分析 [13]，符号执行 [21]，等。这些技术往往在定义明确的上下文中表现出色，但难以扩展并推广到更广泛的场景。

5.2 基于学习和大语言模型的方法

基于学习的方法会以某种方式利用机器学习来辅助循环不变式生成。Code2Inv [26] 是这类方法的典型示例：它通过与证明检查器交互，利用深度强化学习（RL）学习循环不变式。类似地，LIPuS [30] 使用基于强化学习的剪枝和 SMT 求解来生成循环不变式。

近来，研究人员开始探索利用大语言模型生成程序性质，例如最弱前置条件 [16, 17]，后置条件 [6]，以及完整的程序规格 [19]。与此同时，研究人员也开始研究如何利用

大语言模型生成循环不变式； [15, 23, 14] 的研究为此类探索奠定了基础，并确立了这样一种观点：大语言模型确实能够生成有用的循环不变式。此后，涌现出大量后续研究。

[24] 的作者使用大语言模型生成循环不变式候选，然后由 Vampire 定理证明器 [18] 对其进行验证，并在有界模型检测 [3] 的背景下用它们替换循环。类似地， [28] 的研究同时使用大语言模型和有界模型检测：他们的方法利用大语言模型生成循环不变式候选，并利用有界模型检测筛除错误谓词。Lemur 框架 [29] 使用大语言模型生成并修复候选循环不变式，随后在 ESBMC [20] 和 UAutomizer [11] 框架中将其用于程序验证。

这些近期研究呈现出的一个共同主题，是将大语言模型与形式化工具相结合的益处；这些方法可归入“神经符号人工智能”的范畴。我们的工作采取了类似的思路，但在以下方面有所不同：（1）使用基于大语言模型的最弱前置条件推理进行系统性的反向链接；（2）双层神经符号细化与修复，将基于大语言模型的内部霍尔逻辑推理与 OpenJML 的外部反馈相结合；（3）自动处理（顺序组合的）多循环程序；（4）聚焦于 Java，与主要面向 C语言的现有研究形成对比。

6 结论

本文介绍了 NeuroInv，这是一种用于循环不变式生成的神经符号方法，采用双层细化与修复机制。NeuroInv 包含两个关键模块：（1）神经推理模块，利用大语言模型和霍尔逻辑，通过反向链式最弱前置条件推理来推导和细化候选循环不变式；（2）验证引导的符号模块，利用 OpenJML 提供的反例迭代修复不变式。

我们在一个包含 150 个 Java 程序的综合基准测试上评估 NeuroInv。该基准测试涵盖了包含单个循环、多个（顺序）循环、多个数组、随机分支和含噪代码片段的程序。NeuroInv 的成功率达到 99.5%，显著优于临时提示（66.4%）、纯神经消融实验（84.7%），以及先进的、基于大语言模型的 Java 规格生成器（28.5%） [19]。

此外，由 10 个规模更大的多循环程序（平均包含 7 个循环）组成的困难基准测试进一步表明，NeuroInv 能够扩展到更复杂的场景。在这一设置下，NeuroInv 与临时提示之间的性能差距进一步扩大：NeuroInv 的成功率达到 62%，而临时提示仅为 14%。

鉴于所取得的成果，NeuroInv 构成了一个有效的循环不变式生成框架。我们概述的未来工作旨在将这一神经符号方法扩展到更广泛、更具挑战性的验证场景。

参考文献

1 Cristiano Calcagno、Dino Distefano 和 Viktor Vafeiadis。双溯源资源不变式综合。载于亚洲编程语言与系统研讨会，第 259–274 页。Springer，2009 年。2 Mark Chen。评估基于代码训练的大语言模型。*arXiv* 预印本 *arXiv:2107.03374*，2021 年。3 Edmund Clarke、Armin Biere、Richard Raimi 和 Yunshan Zhu。使用可满足性求解进行有界模型检测。《系统设计中的形式化方法》，19(1):7–34，2001 年。4 Patrick Cousot 和 Radhia Cousot。抽象解释：一种统一格模型，通过构造或近似不动点对程序进行静态分析。载于 第 *4* 届

ACM SIGACT-SIGPLAN 编程语言原理研讨会，第 238–252 页，1977 年。 5 Edsger W. Dijkstra。守卫命令、非确定性与程序的形式化推导。《ACM 通讯》，18(8):453–457，1975 年 8 月。URL： https://doi-org.elib.tcd.ie/10.1145/360933.360975, doi:10.1145/360933.360975。 6 Madeline Endres、Sarah Fakhoury、Saikat Chakraborty 和 Shuvendu K Lahiri。大语言模型能否将自然语言意图转化为形式化方法后置条件？《ACM 软件工程汇刊》，1(FSE):1889–1912，2024 年。 7 Michael D Ernst、Jeff H Perkins、Philip J Guo、Stephen McCamant、Carlos Pacheco、Matthew S Tschantz 和 Chen Xiao。用于动态检测可能不变式的 Daikon 系统。《计算机程序科学》，69(1-3):35–45，2007 年。 8 Cormac Flanagan 和 K Rustan M Leino。Houdini：一种用于 ESC/Java 的注释辅助工具。载于欧洲形式化方法国际研讨会，第 500–517 页。Springer，2001 年。 9 Robert W Floyd。为程序赋予含义[1]。《计算机科学的数学方面》，19:19，1967 年。 10 Artur d’Avila Garcez 和 Luis C Lamb。神经符号人工智能：第三次浪潮。《人工智能评论》，56(11):12387–12406，2023 年。 11 Matthias Heizmann、Daniel Dietsch、Marius Greitschus、Jan Leike、Betim Musa、Claus Schätzle 和 Andreas Podelski。采用双轨证明的 Ultimate Automizer：（竞赛论文）。载于系统构造与分析工具和算法国际会议，第 950–953 页。Springer，2016 年。 12 C. A. R. Hoare。计算机编程的公理基础。《ACM 通讯》，12:576–580，1969 年 10 月。 13 Andreas Humenberger、Maximilian Jaroschek 和 Laura Kovács。多项式赋值多路径循环的不变式生成。载于验证、模型检测与抽象解释国际会议，第 226–246 页。Springer，2017 年。 14 Christian Janßen、Cedric Richter 和 Heike Wehrheim。ChatGPT 能否支持软件验证？载于软件工程基础方法国际会议，第 266–279 页。Springer，2024 年。 15 Adharsh Kamath、Aditya Senthilnathan、Saikat Chakraborty、Pantazis Deligiannis、Shuvendu K Lahiri、Akash Lal、Aseem Rastogi、Subhajit Roy 和 Rahul Sharma。使用大语言模型寻找归纳循环不变式。arXiv 预印本 arXiv:2311.07948，2023 年。 16 Daragh King、Vasileios Koutavas 和 Laura Kovacs。Fuzzfeed：一种使用大语言模型和模糊测试自动生成最弱前置条件的方法，2025 年。URL： https://arxiv.org/abs/2507.05272, arXiv:2507.05272。 17 Daragh King、Vasileios Koutavas 和 Laura Kovács。基于大语言模型生成最弱前置条件和精确数组不变式。载于 IEEE/ACM 第十三届软件工程形式化方法国际会议（FormaliSE），第 1–5 页，2025 年。 doi:10.1109/FormaliSE66629.2025.00016。 18 Laura Kovács 和 Andrei Voronkov。一阶定理证明与 Vampire。载于国际计算机辅助验证大会，第 1–35 页。Springer，2013 年。 19 Lezhi Ma、Shangqing Liu、Yi Li、Xiaofei Xie 和 Lei Bu。SpecGen：通过大语言模型自动生成程序形式化规约。arXiv 预印本 arXiv:2401.08807，2024 年。 20 Rafael Sá Menezes、Mohannad Aldughaim、Bruno Farias、Xianzhiyu Li、Edoardo Manino、Fedor Shmarov、Kunjian Song、Franz Brauße、Mikhail R Gadelha、Norbert Tihanyi 等。ESBMC v7.4：充分利用区间的能力：（竞赛论文）。载于系统构造与分析工具和算法国际会议，第 376–380 页。Springer，2024 年。 21 ThanhVu Nguyen、Matthew B Dwyer 和 Willem Visser。Symlnfer：使用符号状态推断程序不变式。载于 2017 年第 32 届 IEEE/ACM 自动化软件工程国际会议（ASE），第 804–814 页。IEEE，2017 年。

22 OpenAI。《介绍 o3 和 o4-mini》，2025。网址： https://openai.com/index/introducing-o3-and-o4-mini/。23 Kexin Pei、David Bieber、Kensen Shi、Charles Sutton 和 Pengcheng Yin。大语言模型能否推理程序不变式？载于国际机器学习大会，第 27496–27520 页。PMLR，2023 年。24 Muhammad AA Pirzada、Giles Reger、Ahmed Bhayat 和 Lucas C Cordeiro。无需循环展开、用于有界模型检测的大语言模型生成不变式。载于第 *39* 届 *IEEE/ACM* 自动化软件工程国际会议论文集，第 1395–1407 页，2024 年。25 Sander Schulhoff、Michael Ilie、Nishant Balepur、Konstantine Kahadze、Amanda Liu、Chenglei Si、Yinheng Li、Aayush Gupta、HyoJung Han、Sevien Schulhoff 等。提示词报告：提示词工程技术的系统性综述。*arXiv* 预印本 *arXiv:2406.06608*，2024 年。26 Xujie Si、Aaditya Naik、Hanjun Dai、Mayur Naik 和 Le Song。Code2inv：用于程序验证的深度学习框架。载于国际计算机辅助验证大会，第 151–164 页。Springer，2020 年。27 Cheng Wen、Jialun Cao、Jie Su、Zhiwu Xu、Shengchao Qin、Mengda He、Haokun Li、Shing-Chi Cheung 和 Cong Tian。利用静态分析和程序验证，借助大语言模型提升程序规约合成。载于国际计算机辅助验证大会，第 302–328 页。Springer，2024 年。28 Guangyuan Wu、Weining Cao、Yuan Yao、Hengfeng Wei、Taolue Chen 和 Xiaoxing Ma。大语言模型遇上有界模型检测：神经符号循环不变式推断。载于第 *39* 届 *IEEE/ACM* 自动化软件工程国际会议论文集，第 406–417 页，2024 年。29 Haoze Wu、Clark Barrett 和 Nina Narodytska。Lemur：将大语言模型整合到自动化程序验证中。*arXiv* 预印本 *arXiv:2310.04870*，2023 年。30 Shiwen Yu、Ting Wang 和 Ji Wang。通过 SMT 求解增强的强化学习推断循环不变式。载于第 *32* 届 *ACM SIGSOFT*国际软件测试与分析研讨会论文集，第 175–187 页，2023 年。

A 提示词

清单 1 用于循环不变式生成的临时提示词。

```
Compute the loop invariant(s) for the loop(s) in the following Java
code. Please use the format of JML annotations. Annotate the loop(s)
with the invariant(s). Provide no additional explanations beyond the
program code and the required annotation. Reason through your
solution internally.
```

清单 2 用于最弱前置条件计算的 NeuroInv 提示词。

```
**Task: Calculate Weakest Precondition**

You are given a Java program with segmented code. Your task is to
calculate the weakest precondition (WP) for a specific code
segment with respect to a given postcondition.

**Segmented Program:**
‘‘‘java
{segmented_program}
‘‘‘

**Code Segment to Analyze:**
```

```
Segment index: {segment_index} {tag_description}

**Postcondition (must be true AFTER the segment executes):**
{postcondition}

**Instructions:**
1. Identify the code segment between the specified tags
2. Apply weakest precondition calculus rules to work backward
through the statements
3. For assignments: WP(x := e, R) = R[x -> e] (substitute e for x
in R)
4. For sequences: WP(S1; S2, R) = WP(S1, WP(S2, R))
5. For conditionals: WP(if B then S1 else S2, R) = (B => WP(S1,
R)) AND (NOT B => WP(S2, R))
6. Simplify your result as much as possible
7. Express the result using logical notation
(AND, OR, =>, NOT, FORALL, EXISTS)

**Example:**
For code segment: `x = x + 1;` with postcondition: `x > 5`
The WP would be: `x + 1 > 5` which simplifies to `x > 4`
```

清单 3 用于循环不变式生成的 NeuroInv 提示。

```
**Task: Generate Loop Invariant**

You are given a Java program with segmented code. Your task is to
generate a suitable loop invariant for a specific while loop.

**Segmented Program:**
```java
{segmented_program}
```

While Loop to Analyze:
Loop index: {loop_index} (between tags `// while {loop_index}
open` and `// while {loop_index} close`)

Loop Postcondition (must be true after loop exits):
{loop_postcondition}

Instructions:
1. Identify the while loop between the specified tags
2. Extract the loop guard condition B
3. Analyze the loop body to understand what the loop does
4. Generate the weakest possible loop invariant I that:
   - Is true before the first iteration (initialization)
   - Is preserved by each iteration (I AND B => WP(loop_body, I))
   - Combined with loop exit (I AND NOT(B)) establishes the loop
   postcondition, {loop_postcondition}
5. Express the invariant using logical notation
(AND, OR, =>, NOT, FORALL, EXISTS)

A good loop invariant typically includes:
- Bounds on loop variables (e.g., 0 <= i <= N)
```

```
- Relationships between variables (e.g., i < j)
```

列表 ng 4 用于检查蕴含关系的 NeuroInv 提示　　之一。

```
**Task: Verify Implication One (Loop Preservation)**

You are given a loop invariant and must verify that it is
preserved through one iteration of the loop.

**Segmented Program:**
```java
{segmented_program}
```

While Loop:
Loop index: {loop_index} (between tags `// while {loop_index}
open` and `// while {loop_index} close`)

Loop Invariant (I):
{loop_invariant}

Loop Guard (B):
{loop_guard}

Weakest Precondition of Loop Body w.r.t. I:
WP(loop_body, I) = {wp_loop_body}

Implication to Verify:
I AND B => WP(loop_body, I)

Explanation:
If the invariant holds and the loop continues (guard is true),
does executing the loop body preserve the invariant?

CRITICAL CONSTRAINT:
A loop invariant must be **self-sufficient** for formal
verification. You may ONLY use `I AND B` to prove WP(loop_body, I).
You MUST NOT reference:
- Method preconditions
- Variable values from before the loop
- Any facts not explicitly stated in the invariant I

If you need external facts (like preconditions) to complete the
```

```
insufficient and must be strengthened**Instructions:**
```

图 5 用于检查第二个蕴含的 NeuroInv 提示词。

```
**Task: Verify Implication Two (Loop Termination Correctness)**

You are given a loop invariant and must verify that when the loop



**Segmented Program:**
```java
{segmented_program}
```

While Loop:
Loop index: {loop_index} (between tags `// while {loop_index}
open` and `// while {loop_index} close`)

Loop Invariant (I):
{loop_invariant}

Loop Guard (B):
{loop_guard}

Loop Postcondition (must be true after loop exits):
{loop_postcondition}

Implication to Verify:
I AND NOT(B) => LoopPostCondition

Explanation:
If the invariant holds and the loop exits (guard is false), is
the loop postcondition satisfied?

CRITICAL CONSTRAINT:
A loop invariant must be **self-sufficient** for formal
verification. You may ONLY use `I AND NOT(B)` to prove the loop
postcondition. You MUST NOT reference:
- Method preconditions
- Variable values from before the loop
- Any facts not explicitly stated in the invariant I

If you need external facts (like preconditions) to complete the
```

```
**Instructions:**
1. Assume ONLY that I and NOT(B) (negation of the loop guard) are
true
2. Using ONLY these facts, determine if the loop postcondition

3. Do NOT use preconditions, even if variables are unmodified
4. If you need any additional facts to prove the loop

5. If it fails, identify exactly what facts are missing from the
```

清单 6 用于循环不变式细化的 NeuroInv 提示。

```
**Task: Refine Loop Invariant**

A loop invariant has failed verification. Your task is to refine
it by strengthening or adjusting it to satisfy the Hoare logic
requirements.

**Segmented Program:**
```java
{segmented_program}
```

While Loop:
Loop index: {loop_index} (between tags `// while {loop_index}
open` and `// while {loop_index} close`)

Current Invariant (FAILED):
{current_invariant}

Failed Implication:
{implication_name}

Failure Reason:
{failure_reason}

{refinement_guidance}

Instructions:
1. Analyze why the current invariant failed
2. Identify what additional conditions or strengthenings are
needed
3. Propose a refined invariant that:
   - Still holds initially (before the first iteration)
   - Is strong enough to pass the failed implication
   - Is not overly restrictive (can be maintained by the loop)
4. Explain your refinement strategy
```

清单 7 蕴含失败的细化指导。

```
**Refinement Strategy for Implication One:**
- The invariant is not preserved through the loop body
- The invariant must be self-sufficient - it cannot rely on
```

```
preconditions or external facts
- Consider strengthening the invariant by adding conditions about:
  - Facts from preconditions about unmodified variables/arrays
  (if needed to prove preservation)
  - Variables modified in the loop body
  - Relationships that must be maintained across iterations
  - Bounds that are always respected
  - Properties that hold before and after each iteration
- Ensure the strengthened invariant still allows initialization
(holds before first iteration)
```

列表 8 针对两个失败蕴含的细化指导。

```
**Refinement Strategy for Implication Two:**
- The invariant + loop exit doesn't imply the loop postcondition
- The invariant must be self-sufficient - it cannot rely on
preconditions or external facts
- Consider strengthening the invariant to explicitly include:
  - Properties that hold throughout but aren't stated in the
  current invariant
  - What has been accomplished when the loop exits
  - Relationships needed for the loop postcondition
  - Quantified properties over the full range
- Ensure the strengthened invariant can still be preserved by the
loop
```

Listing 9 用于循环不变式修复的 NeuroInv 提示　修复。

```
**Task**
For the Java method {method_name} shown below (under **Program**),



- You MAY modify loop invariants and, if present, the loop

- You MUST NOT modify method preconditions, postconditions,


- You MUST NOT modify the Java code.

The OpenJML output (under **OpenJML Output**) describes the
failures. Use it carefully to strengthen the loop invariant and/
or adjust the variant to ensure inductiveness and termination as
required.

**Scope**
- Allowed changes:
  - Loop invariants
  - Loop variants (if present)
- Forbidden changes (leave exactly as-is):
  - Method-level requires/ensures/assignable clauses
  - Class-level specifications
```

```
  - Any Java implementation code

**Deliverable**
Return the complete program with only the loop invariant(s) (and
variant(s), if present) updated. Do not include explanations or
comments beyond what is explicitly requested.

**OpenJML Output**
{openjml_output}

**Program**
{program}
```