软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):2694−2718 [doi: 10.13328/j.cnki.jos.007590] [CSTR: 32375.14.jos.007590]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于大语言模型的 Java 新特性测试程序生成*

赵英全[1], 张博凯[1], 王 赞[1], 郭以勒[1], 陈佳丽[2], 陈 翔[3], 陈俊洁[1]

[1](天津大学 智能与计算学部, 天津 300350)

[2](龙岩学院 数学与信息工程学院, 福建 龙岩 364012)

[3](南通大学 人工智能与计算机学院, 江苏 南通 226019)

通信作者: 陈俊洁, E-mail: junjiechen@tju.edu.cn

摘 要: Java 编程语言自诞生以来, 就始终处于不断发展和演变的过程中. 随着新语言特性及编程范式的不断涌现, Java 的表达能力和执行效率在不断提升, 推动了整个软件生态的进步. 为了确保 Java 生态的安全性和稳定性, 研究人员设计了多种针对 Java 编译器及虚拟机的测试程序生成方法, 以检测潜在的缺陷. 然而, 现有工作主要针对成熟的语法设计测试程序生成方法或变异策略, 难以对语言新特性进行有效的测试. 为此, 提出一种基于大语言模型的 Java 新特性测试程序生成方法 LumiX. 首先, LumiX 利用大模型对使用自然语言描述的新语言特性进行总结, 概括得到新特性的使用描述; 随后, LumiX 以历史揭错测试程序作为种子程序, 利用程序分析工具提取种子程序中可复用的程序元素, 结合新特性的使用描述, 利用大模型生成针对新特性的代码片段. 新生成的代码片段将被插入至种子程序中, 生成能够覆盖新特性的测试程序. 最后, LumiX 设计了双层的差分测试方法, 分别利用不同的 Java 编译器 (javac 和 ECJ) 及 Java 虚拟机 (HotSpot 和 OpenJ9) 对新生成的测试程序进行编译和执行, 通过对比不同编译器和虚拟机的执行结果来检测潜在的缺陷. 实验结果表明, LumiX 能够有效生成覆盖 Java 语言新特性的测试程序, 并提升现有工具对 Java 语言新特性的测试能力. 同时, 将 LumiX 应用于最新发布的 Java 编译器以及虚拟机的测试中, 累计发现 16 个未知缺陷, 其中 12 个已被开发人员确认或修复.

关键词: 大语言模型; 测试程序生成; Java 编译器; Java 虚拟机; 编译器测试; JVM 测试

中图法分类号: TP311

中文引用格式: 赵英全, 张博凯, 王赞, 郭以勒, 陈佳丽, 陈翔, 陈俊洁. 基于大语言模型的Java新特性测试程序生成. 软件学报, 2026, 37(7): 2694–2718. http://www.jos.org.cn/1000-9825/7590.htm

英文引用格式: Zhao YQ, Zhang BK, Wang Z, Guo YL, Chen JL, Chen X, Chen JJ. Java New Feature Test Program Generation Based on Large Language Models. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 2694–2718 (in Chinese). http://www.jos.org.cn/1000-9825/7590.htm

Java New Feature Test Program Generation Based on Large Language Models

ZHAO Ying-Quan[1], ZHANG Bo-Kai[1], WANG Zan[1], GUO Yi-Le[1], CHEN Jia-Li[2], CHEN Xiang[3], CHEN Jun-Jie[1]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

[2](School of Mathematics and Information Engineering, Longyan University, Longyan 364012, China)

[3](School of Artificial Intelligence and Computer Science, Nantong University, Nantong 226019, China)

Abstract: Since its inception, the Java programming language has been in a process of constant development and evolution. As new language features and programming paradigms continuously emerge, Java's expressiveness and execution efficiency are steadily improving,

* 基金项目: 国家重点研发计划 (2024YFB4506300); 国家自然科学基金 (62472310, 62322208, 62232001, 12411530122)
本文由"智能化基础软件可信构造和供应链安全"专题特约编辑王林章教授、司徒凌云研究员、钟浩研究员、 向剑文教授、张路教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20, 2025-12-04; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-06-02

driving progress in the whole software ecosystem. To ensure the security and stability of the Java ecosystem, researchers have designed multiple test program generation methods for Java compilers and virtual machines to detect potential defects. However, the existing research primarily focuses on designing test program generation methods or mutation strategies for mature Java syntax, making it challenging to effectively test new language features. To this end, this study proposes LumiX, a test program generation method for Java's new features based on large language models (LLMs). First, LumiX adopts LLMs to summarize new language features using natural language descriptions and obtains the usage descriptions of the new features. Then, it employs historically bug-revealing test programs as seed programs and applies program analysis tools to extract reusable program elements from the seed programs. By combining the usage descriptions of new features, LLMs are employed to generate code snippets that incorporate the new features. The newly generated code snippets are then inserted into the seed programs to generate test programs that can cover the new features. Finally, LumiX designs a dual-layer differential testing method, which utilizes different Java compilers (javac and ECJ) and virtual machines (HotSpot and OpenJ9) to compile and execute the newly generated test programs, with potential defects detected by comparing the execution results of different compilers and virtual machines. Experimental results show that LumiX can effectively generate test programs that cover Java's new language features and enhance the testing capabilities of existing tools for Java's new language features. Additionally, this study applies LumiX to the latest released Java compilers and virtual machines, and a total of 16 unknown defects have been discovered, 12 of which have been confirmed or fixed by developers.

Key words: large language model (LLM); test program generation; Java compiler; Java virtual machine (JVM); compiler testing; JVM testing

自 1995 年诞生以来, Java 编程语言就始终保持着旺盛的生命力. 从最初 "一次编写, 随处运行"的理念到近年来对云原生架构的支持, Java 持续的语言特性和编程范式更新, 不仅推动了自身生态的发展, 也对整个软件工程领域产生了深远的影响. 如今, Java 编程语言仍在以每半年一次的速度进行版本迭代, 目前版本号已经更新至 Java 24[1]. 每个版本都引入了不同的新语言特性 (即新特性) 或编程范式, 例如 Java 8 中的 Lambda 表达式[2]、Java 19 中的结构化并发 (structured concurrency)[3]等. 然而, 新特性的引入在提升语言功能的同时, 也带来了诸如新特性实现正确性以及兼容性等方面的挑战. 作为 Java 软件生态的重要组成部分, 新特性实现的正确性直接关系到 Java 编译器、Java 虚拟机 (Java virtual machine, JVM) 等核心基础设施的稳定性和可靠性. 其中, Java 编译器旨在将由高级语言编写的 Java 源程序编译为平台无关的字节码文件, 编译生成的字节码文件则由 JVM 进行加载、验证与执行. 因此, 对引入的新特性进行针对性的测试, 确保新特性的引入不会破坏 Java 生态的稳定性和可靠性至关重要[4].

为了保障 Java 生态核心基础设施 (即 Java 编译器及 JVM) 的质量安全, 现有研究工作大多通过设计基于规约[5–8]或基于变异[9–14]的测试程序生成方法来对其进行测试. 其中, 基于规约的测试程序生成工作大多通过随机组合目标语言的不同语法规则, 从无到有地生成新测试程序[6]. 例如基于 Java 语法的测试程序生成方法 JavaFuzzer[5], 其通过组合已有的不同 Java 语法来生成新的测试程序, 以验证 JVM 对不同语法规则的解析及执行是否正确. 基于变异的测试程序生成方法则针对特定的语法结构或代码优化功能, 设计相应的变异策略来生成新的测试程序. 例如, 针对 JVM 启动阶段进行测试的 classfuzz[15], 其设计了大量针对 Java 语法层面的变异策略 (如, 修改变量修饰符等) 来生成新的测试程序, 以检测 JVM 对特定语法结构的解析是否正确. JITFuzz[12]则针对 JVM 即时编译器 (just-in-time, JIT) 的代码优化功能进行测试, 设计能够触发 JIT 编译器优化的变异策略 (如, 插入可内联函数等) 来生成新的测试程序. 然而, 现有的研究方法仅针对较为稳定的语法结构或代码优化进行测试, 对于新引入的语言特性, 现有方法往往很难直接应用. 其主要原因在于: (1) 针对新特性的测试往往需要针对性地设计代码生成规则或变异策略, 而现有方法往往仅针对特定的语法结构或代码优化, 且大多在字节码层面操作, 难以覆盖体现于源码的语言新特性, 例如, Java 语法糖; (2) 现有方法大多依赖包含特定特性特征的种子程序, 并借助程序分析工具 (例如, Soot[16]) 进行解析. 但新特性在发布初期通常缺乏具有代表性的种子程序, 且主流的分析工具在未适配前难以正确解析新特性的语法, 这限制了部分方法的应用. 此外, 尽管已有大量利用大模型生成测试程序的研究工作[17–20], 但这些工作均不是针对语言新特性而设计, 因此同样无法对其进行有效的测试. 因此, 如何生成针对新特性的测试程序, 保障新特性实现的正确性, 是亟待解决的问题.

为此, 本文提出一种基于大语言模型的 Java 新特性测试程序生成方法 LumiX, 旨在利用大模型强大的文本理解及代码生成能力, 结合程序分析方法, 生成能够覆盖语言新特性的测试程序, 从而检测 Java 编译器及 JVM 对新

特性的解析和执行是否正确. 鉴于语言新特性通常体现在高级语言层面, 如语法糖, LumiX 选择在源代码层面生成测试程序. 具体来说, LumiX 首先利用大模型强大的文本理解能力, 对使用自然语言描述的语言新特性进行总结, 从而概括获得语言新特性的使用描述. 该新特性使用描述包含了其使用场景、使用方法以及示例代码等摘要信息. 随后, LumiX 利用大模型强大的代码生成能力, 根据新特性的使用描述, 生成能够覆盖新特性的代码片段. 为了提升生成测试程序的揭错能力, LumiX 使用历史揭错测试程序作为种子程序, 为新生成的新特性代码片段提供丰富的上下文信息. 同时, 为了增强种子程序与新特性代码片段之间的交互, LumiX 利用程序分析工具对种子程序中可复用的变量及函数信息进行提取, 结合新特性使用描述生成更复杂的代码片段. 新生成的代码片段将被插入至种子程序中, 从而生成新的测试程序. 最后, LumiX 设计了双层的差分测试 (differential testing) 方法. 具体来说, LumiX 首先利用不同的 Java 编译器 (即 javac 和 ECJ) 对生成的测试程序进行编译, 通过对比不同编译器的编译结果来分析其是否检测出潜在的编译器缺陷. 随后, 为了进一步分析 JVM 关于新特性的实现是否正确以及 Java 编译器中是否存在错误编译 (miscompilation)[4], LumiX 利用不同的 JVM (即 HotSpot 和 OpenJ9) 对编译后的测试程序进行执行, 通过对比不同 JVM 的执行结果来检测 JVM 中潜在的缺陷.

为了评估 LumiX 的有效性, 本文在 4 个长期维护的 OpenJDK 版本 (即 OpenJDK 8、11、17、21)、两个 Java 编译器 (javac 及 ECJ) 以及两个主流的 JVM 实现 (HotSpot 及 OpenJ9) 上进行了广泛的对比实验. 考虑到本文是针对 Java 语言新特性的测试程序生成方法, 以及 LumiX 在源码层面上生成新的测试程序, 本文将 LumiX 与现有基于规约的测试程序生成方法 JavaFuzzer 以及通用模糊测试工具 Fuzz4All[21]进行对比. 同时, 本文设计了 LumiX 的变体方法, 通过使用 JavaFuzzer 生成的测试程序作为种子程序, 从而评估本方法是否能有效提升现有工具对 Java 语言新特性的测试能力. 实验结果表明, 相较于 JavaFuzzer 和 Fuzz4All, LumiX 能够有效生成覆盖 Java 语言新特性的测试程序. 同时, LumiX 能够有效提升现有工具对 Java 语言新特性的测试能力. 特别地, 本文将 LumiX 应用于最新版本的 Java 编译器以及 JVM 测试中, 累计发现 16 个未知缺陷, 其中 12 个已被开发人员确认或修复, 且部分未知缺陷被开发人员标记为长期存在的缺陷.

本文的主要贡献总结如下.

- 提出针对 Java 语言新特性的测试程序生成方法, 通过挖掘 Java 新特性文本描述, 结合大模型与程序分析生成针对新特性的测试程序.
- 在代码片段级别生成新特性测试代码, 结合历史揭错测试程序中丰富的上下文, 提升生成新特性测试程序的揭错能力.
- 设计并实现基于大语言模型的 Java 新特性测试程序生成方法 LumiX, 且将 LumiX 所有源代码及使用的数据集进行开源[22], 促进后续研究的参考与使用.
- 针对最新版本的 Java 编译器及 JVM 的测试结果表明, 本文方法能够有效生成针对新特性的测试程序, 以及提升现有工作对新特性测试的覆盖能力. 累计发现 16 个未知缺陷, 其中 12 个均已被开发人员确认或修复.

1 相关工作

本节主要对 Java 测试程序生成的相关工作进行介绍, 针对的测试对象为 Java 生态中的两大核心基础设施: Java 编译器及 JVM, 涉及方法主要为基于规约以及基于变异的测试程序生成方法. 同时, 强调本文与现有工作的区别以及本文的创新性.

1.1 基于规约的测试程序生成

基于规约的测试程序生成方法旨在通过设计特定编程语言的语法规则或人为设计的约束, 生成符合语法规则或约束的测试程序[4]. 通常来说, 基于规约的测试程序生成主要分为两类, 分别是: 随机差分测试 (random differential testing, RDT)[6–8]和程序合成 (program synthesis)[23–27]. 其中, 随机差分测试通过随机组合编程语言的语法规则, 从无到有地随机生成符合目标语言语法规则的测试程序, 随后采用差分测试作为测试预言检测编译器中潜在的缺陷[6]. 程序合成则通过设计特定的程序合成规则, 根据预定义的约束条件在种子程序 (或模板程序) 上生成新的测试程

序, 提升生成测试程序的复杂性和多样性[23].

目前, 随机差分测试已被广泛应用于编译器的测试工作中, 是编译器测试的主要方法之一. 比较经典的方法为 Yang 等人[6]提出的针对 C 编译器的测试程序生成工具 Csmith. Csmith 中设计了大量 C 语言语法的生成规则, 例如, 变量声明、控制流语句以及指针和数组等. 通过随机地组合这些语法规则, Csmith 可以生成大量符合 C 语言语法的测试程序, 并基于不同的 C 编译器实现 (例如 GCC、LLVM) 构造测试语言, 检测 C 编译器中潜在的缺陷. 由于 Java 程序往往存在复杂的内部依赖关系以及复杂的面向对象类型关系[28], 现有针对 Java 的随机差分测试工作相对较少. 具有代表性的工作为 JavaFuzzer[5]. JavaFuzzer 在 Java 源程序的层面上设计代码生成规则, 通过生成包含复杂循环结构、算术运算以及异常处理等结构的 Java 测试程序, 对安卓虚拟机以及其他 JVM 实现进行测试.

不同于随机差分测试方法需要设计大量语法的生成规则, 程序合成方法则需要生成符合特定约束条件的测试程序. 例如, 基于模板的测试程序合成方法 JAttack[26], 其通过设计针对 JIT 编译器的测试程序模板, 程序模板中包含使用领域特定语言 (domain specific language, DSL) 描述的待填充空白区域 (hole). 随后通过向模板中的空白区域填充符合约束条件的代码片段, 生成多样的测试程序对 JIT 编译器进行测试. 在此基础上, Zang 等人[26]进一步提出了自动化的测试程序模板生成方法 LeJit, 其通过从已有的项目中提取 Java 程序, 并自动化地将程序中的表达式转换为用 DSL 描述的空白区域, 构建更多样化的程序模板. 此外, 文献 [23–25] 提出了一系列基于历史揭错测试程序的 JVM 测试程序合成方法. 具体来说, Zhao 等人[23]首先提出了 JVM 测试程序合成方法 JavaTailor, 其通过设计 5 种代码片段衡量规则, 从历史揭错测试程序中提取所有符合规则的代码片段, 并将提取到的代码片段随机插入至新的种子程序中, 从而生成新的测试程序. 然而, 不同的代码片段往往会存在相似的程序特征, 例如, 相似的控制流. 因此, 在此基础上, Gao 等人[24]利用代码表示方法对代码片段进行向量化表示和聚类, 并采用基于反馈的代码片段选择策略, 进一步提出了增强 JVM 测试效果的方法 VECT. 考虑到有限的历史揭错代码片段往往会限制测试输入空间. 因此, Zhao 等人[29]提出了代码片段抽象到实例化的 JVM 测试程序生成方法 Jetris, 通过将代码片段的控制流和数据流信息抽象为揭错模式, 并结合新的程序元素将揭错模式实例化为更多样的代码片段. 新生成的代码片段将被插入至新的上下文中, 从而增强新生成测试程序的多样性, 拓展测试输入空间. 相关的研究工作还有: 文献 [30,31] 通过引入外部知识或改进测试种子的构造方式来增强基于搜索的软件测试方法, 从而提升对复杂程序结构和潜在缺陷的检测能力.

尽管现有的基于规约的测试程序方法已经取得较好的结果, 检测到了很多未知的 JVM 缺陷. 然而, 现有方法均不包含新特性的程序特征, 无法对新特性进行有效覆盖. 一方面, 现有随机差分测试方法所设计的代码生成规则均为较基础的语法结构, 不包含新特性的语法结构, 且为所有的新特性实现相应的代码生成规则成本较高[5,25]; 另一方面, 现有的基于程序合成的方法往往依赖于存在相应测试特征的种子程序, 例如, 历史揭错程序[23,24,26]. 然而, 新特性在发布时往往并不具备充足的历史揭错程序. 同时, 代码片段的分析和提取往往依赖程序分析工具, 在新特性发布的初期, 同样没有程序分析工具的支持. 本方法利用大模型生成新特性代码片段很好地解决了需要针对性设计代码生成规则的高成本问题, 利用程序分析工具分析历史揭错种子则避免了新特性对程序分析工具的依赖等问题.

1.2 基于变异的测试程序生成

基于变异的测试程序生成方法通常在已有的种子程序上, 围绕编程语言的语法规则、语义约束以及编译器不同的优化功能来设计相应的变异策略, 从而生成种子程序的变异体来检测潜在的编译器缺陷[9,11,32]. 具有代表性的工作是 Le 等人[9]提出的等价取模输入 (equivalence modulo input, EMI) 方法, 该方法利用覆盖分析工具识别已有测试程序中未被执行的代码语句. 随后, 通过随机删除未被执行的代码语句, 生成与原始种子程序在语义上等价的变异体.

基于变异的 Java 测试程序生成方法在过去一段时间中得到了广泛的关注, 已成为提升 Java 编译器与 JVM 正确性的重要手段之一. 例如, Chaliasos 等人[33]提出的针对编译器类型系统的测试程序编译方法 Hephaestus, 其通过基于类型擦除及类型重写设计的变异规则, 对种子程序进行变异. 生成的测试程序有效地检测出了 Java 编译器

中多个与类型相关的编译器缺陷. Chen 等人[15]提出的 Java 测试程序变异方法 classfuzz, 其设计了一系列针对字节码文件的变异规则, 例如, 修改变量类型及函数返回值类型等. 通过在种子程序上应用不同的变异规则生成新的测试程序, 从而对 JVM 的启动阶段 (即加载、链接及初始化) 进行测试. 在此基础上, Chen 等人[32]进一步提出基于活字节码 (live bytecode) 的测试程序生成方法 classming, 其首先获取已有测试程序中被执行的字节码序列, 随后通过修改被执行字节码的控制流和数据流, 生成大量语义不同的变异体. 最后基于差分测试检测 JVM 执行引擎中潜在的缺陷.

其他基于变异的 Java 测试程序生成方法均针对 JVM 中的 JIT 编译器进行测试. 例如, Wu 等人[12]提出了针对 JVM 中 JIT 编译器的测试程序变异方法 JITFuzz, 其针对 JIT 编译器中不同的代码优化设计相应的变异规则, 例如, 函数内联及逃逸分析等. Jia 等人[34]提出从两个维度对 JIT 编译器进行测试的方法 JOpFuzzer, 其利用 TBar[35]来对已有的测试程序进行变异, 涉及的变异规则包括修改数据类型以及函数调用等. 此外, JOpFuzzer 同时考虑生成 JIT 编译器的不同优化选项配置, 通过打开或关闭不同的优化选项, 使得生成的测试程序能够覆盖更多的 JIT 编译器功能模块. Xie 等人[13]提出了基于优化交互的 JIT 编译器测试方法 MopFuzzer, 其同样基于 JIT 编译器的优化功能设计不同的变异规则, 例如, 循环展开调用及锁消除调用等. 随后, 利用 JVM 的运行时数据引导生成触发不同 JIT 优化组合的测试程序, 从而对 JIT 编译器不同优化的交互进行测试. Li 等人[14]提出了编译空间的概念, 并提出基于变异的编译空间探索方法 Artemis, 其通过对测试程序中的函数调用进行修改, 改变其执行状态, 即解释执行或 JIT 编译执行. 通过对比不同执行状态的执行结果, 从而检测 JIT 编译器中潜在的缺陷.

相似地, 现有基于变异的测试程序生成方法同样无法对新特性进行有效覆盖. 一方面, 现有工作的变异规则主要围绕 JVM 中的特定组件 (例如, JIT 编译器) 而设计, 无法覆盖新特性[12–14]; 另一方面, 现有工作大多在字节码层面直接生成 JVM 可以运行的测试程序[15,32,34]. 然而, 语言新特性往往以语法糖的形式体现于 Java 源码层, 这导致现有方法无法生成具备新特性触发条件的测试程序, 从而限制了其在新特性测试中的应用. 不同于上述方法, 本方法一方面提升了对新特性行为的覆盖能力, 另一方面能够有效暴露 Java 编译器和 JVM 在支持新特性过程中可能存在的实现缺陷, 弥补了语言新特性测试的不足.

2 动机示例

本节通过一个示例来展示 LumiX 的研究动机. 同时, 基于该示例分析为何现有研究方法无法对新特性进行有效的测试.

图 1 展示了由 LumiX 生成的测试程序, 该测试程序成功检测出了 Java 编译器 ECJ 中关于 Java 8 新特性 Lambda 表达式的未知缺陷 (Bug #3792[36]). 具体来说, 该测试程序在第 1 行定义了一个值为 false 的布尔类型变量 DEBUG, 并在第 3 行定义了一个类型为 Object 的局部变量 o, 并在第 5 行为其重新赋值为一个新的 Object 类型对象. 随后在第 6 行, 测试程序通过 if 语句判断 DEBUG 变量是否为 true, 若条件成立, 则执行 if 语句的代码体. 该代码体中创建了一个新的线程, 并在线程中通过 Lambda 表达式指定了线程的执行逻辑. 然而, Lambda 表达式使用了局部变量 o, 而 o 并未被声明为 final 变量, 且不满足 effectively final 条件 (即变量没有显式声明为 final, 但其值在初始化后未被修改). 根据 Java 8 中 Lambda 的语法规范, 表达式中引用的局部变量必须是显式声明为 final 或者是 effectively final, 否则将被视为非法引用, 从而可能导致线程安全问题或内存泄漏等隐患. 在对该测试程序进行编译时, javac 编译器能够成功检测到此类非法引用并抛出编译错误, 而 ECJ 编译器则未能识别出该错误, 并且生成了一个错误的 class 文件.

需要注意的是, 图 1 中浅蓝色标记部分为 LumiX 向种子程序中插入的包含 Java 新特性的代码片段. 其中, Lambda 表达式是 Java 8 中引入的新特性, 用于简洁地表示匿名函数, 可作为参数传递给方法或赋值给变量, 是 Java 向函数式编程迈出的关键一步. LumiX 首先利用程序分析工具识别出种子程序中可以复用的程序元素, 例如, 本示例中的局部变量 o. 随后, 利用大模型生成 Lambda 表达式的代码片段, 并在生成的过程中复用了种子程序中的局部变量 o, 如测试程序第 10 行所示. 该测试程序最终成功触发了 javac 编译器和 ECJ 编译器的不一致行为. 本工作随后

将其提交给 ECJ 的开发人员. 经开发人员确认, 该缺陷的根因在于, 由于 DEBUG 变量恒为 false, 因此, if 代码体中的语句被 ECJ 编译器识别为死代码. ECJ 在编译过程中未对死代码中的 Lambda 表达式进行语义检查, 最终导致了该缺陷的产生. 目前, 开发人员已经对该缺陷进行了确认和修复, 并表示该缺陷可能存在于 Lambda 最初的实现版本中 (“Problem goes back a long ways, I could see it in 4.18 the earliest version I have but it likely goes back all the way to the original lambda implementation!”), 是一个长期存在的编译器缺陷.

```
1  private static final boolean DEBUG = false;
2  public static void main (String[] args) {
3      Object o = new Object();
4      for (int i = 1; i < 10; i++) {
5          o = new Object();
6          if (DEBUG) { //Identified as dead code in ecj
7              Thread virtualThread = Thread.startVirtualThread(() → {
8              try {
9                  java.lang.reflect.Method method = Test.class.getMethod (“funcName”, Object.class);
10                 method.invoke(Object.class, o);
11             } catch (Exception e) {…});
12             try {virtualThread.join(); }
13             catch (InterruptedException e) {e.printStackTrace();}
14         }
15     }
16 }
```

图 1 ECJ 编译器缺陷 (Bug #3792)

通过上述示例程序可以看出, 新引入的语言特性同样会引入新的编译器缺陷. 然而, 现有的测试程序生成方法均未能有效地检测出此类缺陷. 首先, 随机差分测试方法 JavaFuzzer 无法检测到此类缺陷, 其所设计的生成规则仅针对基础的语法, 无法涵盖 Lambda 表达式等新特性的测试. 其次, 基于程序合成的测试程序生成方法 JavaTailor 无法检测到此类缺陷, 其通过将历史揭错测试程序中的代码片段插入至新的上下文中生成新的测试程序. 然而, 历史揭错测试程序中并不包含覆盖新特性的测试程序, 无法对其进行有效的测试. 最后, 基于变异的测试程序方法 classming 同样无法检测到此类缺陷, 其通过改变种子程序的控制流和数据流来生成新的测试程序. 然而, 此类变异方法同样无法使变异后的种子程序具备针对新特性的测试能力.

3 方法介绍

本文提出一种基于大语言模型的 Java 新特性测试程序生成方法 LumiX. LumiX 旨在利用大模型强大的文本理解与代码生成能力, 结合程序分析方法, 生成能够覆盖 Java 新特性的测试程序, 以检测 Java 编译器及 JVM 对新特性的解析和执行是否正确. 具体来说, LumiX 主要包含以下 3 个步骤: (1) 使用大模型总结 Java 新特性的使用描述; (2) 结合大模型与程序分析工具生成能够覆盖 Java 新特性的代码片段, 并插入至种子程序中生成新测试程序; (3) 采用双层差分测试方法, 分别利用不同的 Java 编译器和 JVM 对生成的测试程序进行编译和执行, 以检测潜在的缺陷.

图 2 展示了 LumiX 方法的整体流程图. 为获取 Java 新特性的使用描述, LumiX 首先以 Java 增强提案 (Java enhancement proposal, JEP) 作为输入, 利用大模型的文本理解能力, 总结出不同 Java 新特性的使用描述, 构建出新特性的使用描述池, 如图 2 中①②所示. 随后, LumiX 从使用描述池以及历史揭错种子程序池中分别随机选择一个新特性使用描述和一个历史揭错种子程序 (如图 2 中③所示), 用于生成新的代码片段. 为了提升生成代码片段与种子程序之间的交互, LumiX 利用程序分析工具对种子程序进行解析, 提取可复用的程序元素 (即变量及函数定义), 如图 2 中④⑤所示. 在此基础上, LumiX 利用大模型的代码生成能力, 生成能够覆盖新特性的代码片段, 并插入种子程序中生成新的测试程序, 如图 2 中⑥⑦所示. 最后, LumiX 首先利用不同的 Java 编译器 (即 javac 和 ECJ) 对新生成的测试程序进行编译, 通过对比编译结果来检测是否存在潜在的编译器缺陷. 如果编译成功, LumiX 则会使用不同的 JVM (即 HotSpot 和 OpenJ9) 来分别执行两个 Java 编译器编译后的字节码文件, 通过对比不同 JVM

的执行结果来检测潜在的 JVM 缺陷. 如果在编译和执行的任意一阶段出现不一致行为, LumiX 则会输出不一致报告以及相应的揭错测试程序. 如果在编译和执行阶段均未出现不一致行为, 新生成的测试程序将被放回至历史揭错种子程序池中, 用于接收其他新特性的代码片段. 接下来, 本节将对 LumiX 的 3 个主要步骤进行详细的介绍.

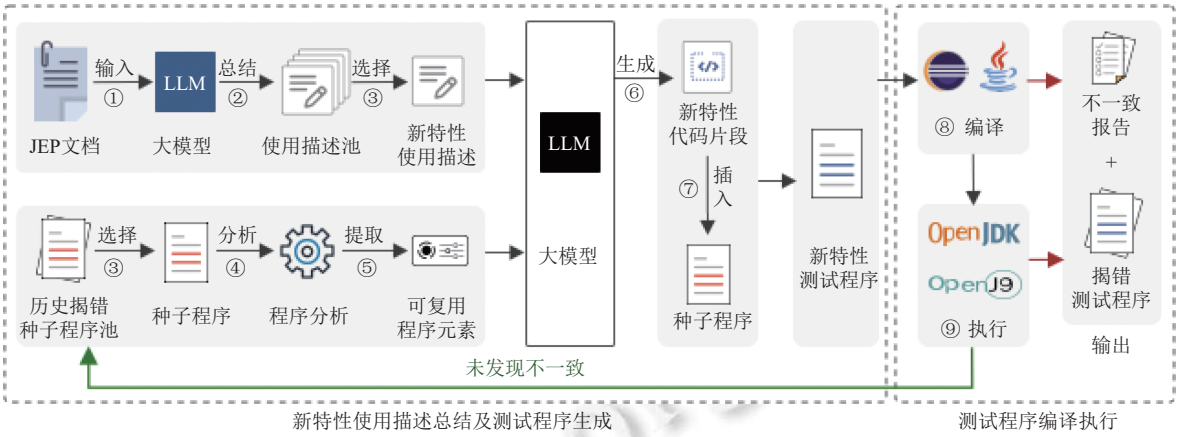

图 2 LumiX 整体流程图

3.1 新特性使用描述总结

为了生成针对新特性的测试程序, 首先需要获取新特性的准确使用描述, 即明确其语法形式以及在实际代码中的典型用法. 这一描述为测试程序生成提供了结构化的信息基础, 确保生成的测试用例能够覆盖新特性的关键行为和边界条件. 通常, Java 的新特性会在官方文档中进行详细的描述, 如 JEP 提案[2]. JEP 是 Java 生态中用于收集和管理 Java 开发工具包 (Java development kit, JDK) 增强功能的提案, 涵盖 JDK 各个方面, 如 JVM、JRE 以及语言特性等. JEP 的存在使得 Java 生态能够收集来自不同社区的需求, 不断增强和完善 Java 体系, 使其更加适应现代软件开发的需求.

具体来说, JEP 中包含了不同状态的提案, 如进行中的提案 (in-flight JEP)、已交付的特性与基础设施提案 (delivered feature and infrastructure JEP)、已撤回的提案 (withdrawn JEP) 等. 鉴于是对已经实现且集成到 JDK 中的新特性进行测试, 本文仅针对已交付的特性与基础设施提案中的新特性进行测试. 每个 JEP 提案中都包含对应新特性的详细描述, 如新特性的总结、动机、描述及测试方式等. 例如, 图 3 展示了 JEP 126 提案中的内容. 该提案描述了 Java 8 中引入 Lambda 的目的、动机以及使用方法. 图 3 中, 标题栏中的 JEP 编号 (即 JEP 126) 为该提案的唯一标识符, 提案标题为: “Lambda 表达式及虚拟扩展方法”. 版本信息中描述了提案作者信息、创建日期、发布版本 (Release 8) 以及所属组件 (tools/javac) 等. 其中, 发布版本标志该提案所属的 Java 版本, 可以用于引入该提案的版本以及后续版本的测试. 因为 Java 向前兼容, 所以后续版本的 Java 编译器和 JVM 能够正确解析和执行历史版本中的语言特性. 例如, Java 11 可以覆盖 Java 8 中引入的新特性, 而 Java 8 则无法覆盖 Java 11 中的新特性.

在总结中, 包含了提案主要改动的简要描述. 在动机及描述中, 则通常会包含引入该特性的目的、使用场景、方法以及示例代码. 例如, 图 3 中的动机描述了引入 Lambda 表达式的目的以及示例代码. 这些信息对于准确地生成能够覆盖新特性的测试程序非常重要, 同时, 示例代码也为 LumiX 生成针对新特性的代码片段提供了参考. 除此之外, 提案的内容还包括该特性的测试方法及依赖关系等信息. 为了能对新特性进行有效的测试, LumiX 首先过滤了非语言特性的 JEP. 例如, 针对 JDK 文档、平台适配、编译方式以及虚拟机选项验证等. 这些 JEP 并非引入新的特性, 而是针对工具链、运行环境或者平台兼容性进行优化, 因其难以通过代码生成的方式对其进行测试, 故对语法新特性的研究意义较小. 在完成对 JEP 的筛选与过滤后, LumiX 对剩下的 JEP 进行文本解析, 提取 JEP 中的标题、发布版本、动机、描述、示例代码等关键信息. 其中, 特性的引入版本用于明确该语言特性首次出现于哪个 Java 版本, 为后续测试程序的编译与运行阶段选择兼容的 Java 编译器及 JVM 环境提供依据. 其次, JEP 中通

常会包含展示新特性典型使用方法的代码示例, 这对于生成高质量、真实有效的新特性代码片段至关重要.

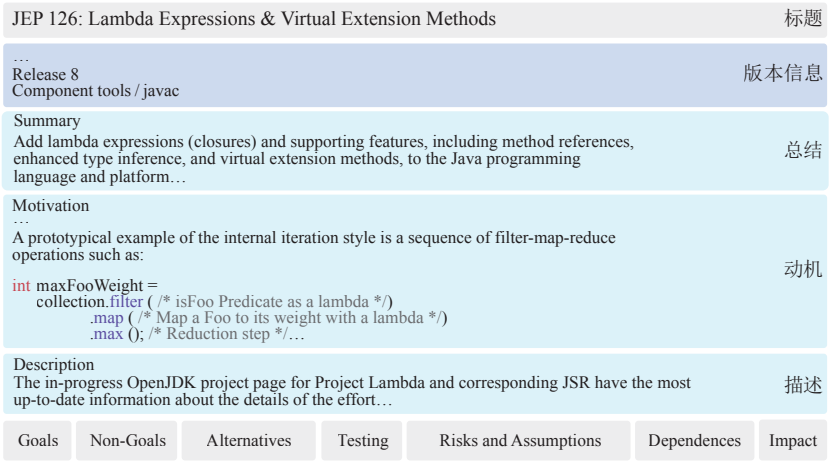

图 3 JEP 126 提案内容示例

然而, 直接将完整的 JEP 描述文本输入大模型进行处理, 往往会面临两个挑战: 一方面, JEP 中包含大量背景性、实现性或非关键性的信息, 容易干扰大模型对测试重点的识别, 削弱生成结果对目标语言特性的聚焦程度, 从而影响测试的有效性; 另一方面, JEP 文本通常篇幅较长, 直接传入会显著增加 Token 消耗, 造成计算资源的浪费, 进而降低测试的效率. 因此, 有必要对 JEP 内容进行精炼和结构化处理, 过滤掉与目标无关的信息, 确保大模型能够聚焦于被测语言特性本身. 为此, LumiX 充分利用大模型在自然语言理解与压缩方面的能力, 对 Java 语言中新引入特性的使用方式进行提炼与总结, 生成高度浓缩的使用描述, 从而在保证信息完整性的前提下显著提升测试任务的针对性与效率. 具体来说, LumiX 设计了一套针对 JEP 解析的大模型提示词模板, 用于自动提取并总结对 JEP 的使用描述. 为了最大程度地保障提取信息的准确性和一致性, LumiX 采用提示词工程 (prompt engineering) 中的常用实践方法, 系统化地构造提示词的不同要素, 包括角色设定、任务描述、提案内容以及输出格式的定义. 以此引导大模型正确、高效地理解 JEP, 并总结出新特性的使用描述, 为后续代码生成任务提供参考.

图 4 展示了 LumiX 生成新特性使用的总结提示词示例. 在该示例中, LumiX 首先将大模型设定为精通自然语言处理, 编程语言以及软件工程的角色, 以确保大模型能够从专业的角度出发, 准确地捕获 JEP 中的关键信息. 现有研究表明[37], 合理的角色设定能够有效提升大模型的任务完成质量, 使其在特定的任务中表现更加出色. 通过赋予其专家身份, 可以引导大模型用更精确的术语总结 JEP 使用描述, 减少无关信息的干扰. 其次, LumiX 在提示词中明确指定具体的任务, 即总结 JEP 中的新特性使用描述, 并强调总结的内容将用于后续的代码生成任务. 最后, 为了确保总结结果的结构化和可用性, LumiX 定义了严格的输出格式, 输出内容包括新特性的标题、发布版本、描述以及示例代码等信息.

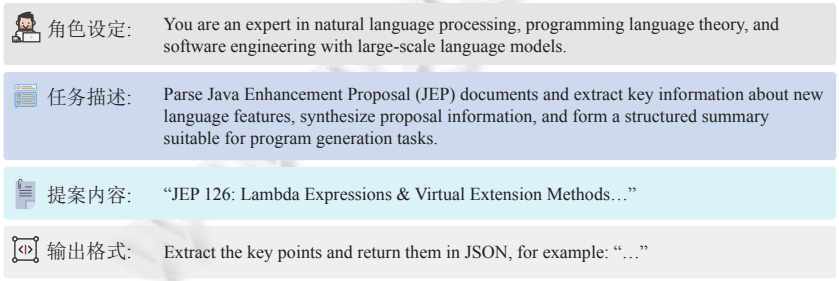

图 4 JEP 126 提案总结提示词示例

图 5 展示了 JEP 126 中 Lambda 表达式的使用描述总结示例. 在该示例中, LumiX 对 Lambda 表达式相关内容进行了结构化归纳, 具体包括该特性的名称、所属 JEP 编号、首次引入的 Java 版本, 以及对 JEP 核心内容的简洁描述. 同时, LumiX 还为新特性的使用提供了一段代表性的代码示例, 用以展示 Lambda 表达式在实际编程中的典型用法, 为后续自动化的代码生成与推荐任务提供了明确的语义和结构参考.


```
"id": 5,
"featureName": "Lambda Expressions & Virtual Extension Methods",
"version": 8,
"description": "Introduces lambda expressions and default methods in interfaces.",
"jepNumber": "JEP 126",
"exampleCode": "names.forEach (name → System.out.println ("Hello," + name));…",
"sourceUrl": "https://openjdk.org/jeps/126",
"isPreview": false,
"availability": true,
```


图 5 新特性使用描述总结示例

在完成 JEP 中所有新特性使用描述的总结后, LumiX 便可以构建一个新特性使用描述池. 该描述池包含每个特性的名称、引入版本、功能描述以及典型使用代码示例等关键信息, 为后续基于语言特性的代码生成任务提供内容支持.

3.2 新特性测试程序生成

给定随机选择的新特性使用描述, LumiX 会以该描述作为生成依据, 结合大模型和程序分析工具构造一段能够覆盖对应语言特性使用方式的代码片段, 并将其插入至种子程序中生成新的测试程序. 需要注意以下几点. (1) LumiX 利用大模型所生成的并非完整的测试程序, 而是聚焦于语言新特性的代码片段. 其主要原因在于, 通过控制生成内容的规模和复杂度, 使大语言模型能够更专注于特定语言特性的代码生成任务, 从而降低生成过程中因引入上下文噪声或无关结构而导致的偏差. (2) 为了增强生成新特性测试程序的揭错能力, LumiX 使用历史揭错测试程序作为种子程序, 为新生成的新特性代码片段提供丰富的上下文信息. 其主要原因在于, 历史揭错程序曾触发测试目标的历史缺陷, 因此包含更具揭错能力的程序特征和路径. 为了提升代码片段与种子程序的交互, LumiX 利用程序分析工具解析当前种子程序中可复用的程序元素, 以此引导大模型在生成代码片段的过程中引入与种子程序的语义交互. (3) 新生成的代码片段将被插入至种子程序中, 从而生成可以覆盖新特性的测试程序. 接下来, 本节将分别介绍 LumiX 中可复用程序元素识别、新特性代码片段生成以及测试程序生成这 3 个任务.

3.2.1 可复用程序元素识别

历史揭错种子程序可以为新生成的代码片段提供丰富的上下文信息. 然而, 直接将新生成代码片段插入至种子程序中, 通常难以构造具有语义交互与结构复杂性的测试程序. 孤立的代码片段通常缺乏与种子程序内容已有代码的联系, 从而限制了测试程序的揭错能力. 但若将完整种子程序直接传给大模型, 让其选择插入点及构建丰富的语义交互, 则会造成大模型幻觉的累积, 降低生成的效率及合法性[38]. 为此, LumiX 利用程序分析工具分析当前种子程序中可复用的程序元素, 如已定义的变量, 函数返回值等信息. 可复用元素的识别一方面为大模型的代码生成提供了丰富的上下文信息, 另一方面通过变量修改与函数调用改变执行路径, 从而影响程序的控制流与数据流依赖, 实现新特性片段与种子程序之间的有效交互.

具体来说, 给定种子程序 S, LumiX 首先会将 S 中所有的方法都解析为抽象语法树 (abstract syntax tree, AST). 随后, LumiX 随机选择种子程序中的一个方法以及该方法中的一个 AST 节点作为新生成代码片段的插入点. 需要注意的是, LumiX 选择对 AST 进行解析而非控制流和数据流. 其主要原因在于, AST 是编程语言编译过程中的基础组成部分, 避免了对更高级程序分析工具的依赖. 接下来, LumiX 将对该节点所有可复用的程序元素进行分析, 从而为新生成的代码片段提供上下文信息. 算法 1 展示了 LumiX 获取种子程序中可复用程序元素的算法, 该算法以种子程序 S、被随机选中的插入点 I 以及插入点所在方法的抽象语法树 T 作为输入, 输出则是在插入点 I 处所有可复用的程序元素集合 R. 在该算法中, LumiX 首先在 *IdentifyReuseableElements* 函数中识别可以复用的全局变量以及函数定义 (第 3、4 行). 随后, LumiX 通过调用 *TraverseAST* 函数识别被选择方法中插入点 I 之前可以被复

用的局部变量. *TraverseAST* 首先将插入点 *I* 标记为终止节点 *stop* (第 8 行). 随后, 从 *stop* 节点开始采用倒序遍历的方式, 识别 *stop* 节点的父节点 *father* (第 10 行). 通过对 *father* 节点的所有子节点进行遍历, 如果在遍历过程中匹配到终止节点 *stop*, 则会终止对当前子树的遍历 (第 12–14 行); 如果在遍历的过程中识别到变量定义, 则会将该节点所对应的变量名称及类型保存至集合 *R* 中 (第 15–19 行). 当 *father* 节点的所有子节点都遍历完成之后, *stop* 节点将被更新为当前的 *father* 节点 (第 21 行). 并将继续向上遍历其父节点的子节点, 直至遍历至当前方法 AST 的 *root* 节点为止 (第 22 行). 最终, *TraverseAST* 将会返回识别到的可复用程序元素集合 *R* (第 23 行), 用于为代码片段的生成提供上下文信息.

算法 1. 可复用程序元素识别.

输入: 种子程序 *S*; 被选中的插入点 *I*; 插入点所在方法的抽象语法树 *T*;
输出: 可复用的程序元素集合 *R*.

```
1.  Function IdentifyReuseableElements (S, T, I)
2.      R ← []; /*记录可复用程序元素*/
3.      R ← R ∪ GetReuseableGlobalFields(S); /*识别可复用全局变量*/
4.      R ← R ∪ GetReuseableFunctions(S); /*识别可复用函数定义*/
5.      R ← TraverseAST(T.root, I, R); /*识别可复用局部变量*/
6.      return R;
7.  Function TraverseAST(T.root, I, R)
8.      stop = I;
9.      do
10.         father = stop.father;
11.         for each child in father.children do
12.             if child == stop then
13.                 break; /*匹配到终止节点, 终止当前子树遍历*/
14.             end
15.             if child is a variable declaration then
16.                 varName ← child.name;
17.                 varType ← child.type;
18.                 R ← R ∪ {(varName, varType)}; /*记录变量名称及其类型*/
19.             end
20.         end
21.         stop = father;
22.     while stop != T.root /*遍历直至根节点*/
23.     return R;
```

为了进一步说明可复用程序元素的提取过程, 图 6 展示了图 1 中种子程序 (不包含浅蓝色生成代码部分) 可复用元素的提取过程. 为了方便说明, 图 6 中对方法 AST 中的部分节点进行了省略和合并, 以减少庞大的节点数量. 在该示例中, 每个 AST 节点对应种子程序中的一个组成部分. 其中, 标记为 VarDel 的节点表示变量定义, 例如节点 2、3 对应图 1 中种子程序的第 3 行. 需要注意的是, 标记为 Expr 的节点为表达式节点, 例如, 节点 8, 该节点的后继节点为对象 o 创建的一个新的引用 (对应图 1 中的第 5 行). 该节点未声明新的变量定义, 因此节点 8 未被标记为 VarDel 节点.

为了提取可复用的程序元素, LumiX 首先在 AST 中随机选择一个节点作为新生成代码的插入点, 即新生成的

代码片段将被插入至该节点之前, 如图 6 中标记为 12 的节点所示. 随后, LumiX 对插入点之前的节点进行分析, 提取其中可被复用的程序元素信息. 通过分析发现, 在插入点之前存在两个变量定义节点, 即节点 2、3 与节点 5、6 (如图 6 中绿色标记节点所示). 在节点 2 中定义了一个类型为 Object 类型的变量 o, 在节点 3 中对其进行了初始化 (对应图 1 种子程序中的第 3 行); 同样, 节点 5 定义了一个类型为 int 类型的变量 i, 在节点 6 中对其进行了初始化, 并赋值为 1 (对应图 1 种子程序中的第 4 行). 识别出来的变量定义将被 LumiX 保存至可复用程序元素池中, 用于为新生成的代码片段提供上下文信息. 需要注意的是, LumiX 提取的可复用程序元素不仅包括局部变量的定义, 还包括在种子程序中定义的全局变量以及成员函数, 如算法 1 中的第 4 行所示. 这些信息都将作为新生成代码片段可利用的程序元素保存至可复用程序元素池中.

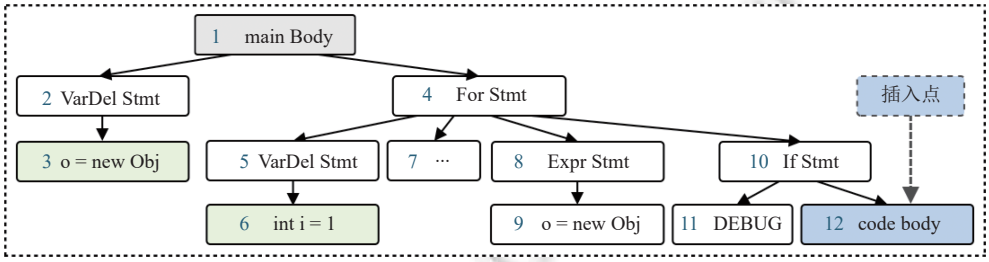

图 6 可复用程序元素识别示例

3.2.2 新特性代码片段生成

在获取到指定插入点的上下文之后, LumiX 便会从新特性使用描述池中随机选择一个新特性使用描述作为待实例化的新特性. LumiX 将随机选择的新特性使用描述与提取的上下文信息结合, 构造相应的代码生成提示词, 引导大模型生成能够覆盖新特性的代码片段. 已有研究表明[38], 模型在处理简洁明确任务时生成效果更好, 而在面对长文本或结构复杂任务时, 其准确性和语义一致性显著下降, 生成代码越长, 正确性和有效性越低[38]. 因此, LumiX 选择生成能够覆盖新特性的代码片段, 而非完整的测试程序. 此外, 本文在第 4.3.4 节对此做了进一步的实验验证, 证明了生成代码片段的合理性及有效性.

新生成代码片段不仅需要体现新特性的用法, 还需要遵循语法和语义正确性要求, 确保可编译性与可执行性. 为了提升生成代码片段的准确性和有效性, LumiX 设计了一套针对代码生成的高质量提示词, 以最大程度保障生成质量. 与新特性使用描述总结提示词的设计过程类似, LumiX 在代码生成提示词中明确赋予了其角色信息 (专家身份)、具体的任务 (生成能够覆盖新特性的代码片段) 以及可复用的程序元素信息 (提升与种子程序的融合). 需要注意的是, 为了提升生成代码片段的有效性, 即生成的代码片段符合 Java 的语法、语义规范, LumiX 在代码生成提示词中添加了大量的约束条件. 约束条件主要包含两部分内容, 分别是: 禁止生成约束 (prohibited generation constraint) 以及必须满足约束 (must satisfy constraint). 禁止生成约束用于限制生成包含导致执行结果不一致逻辑的代码片段, 例如, 禁止生成包含随机赋值语句的代码片段; 必须满足约束则用于保障生成代码片段的有效性, 例如, 代码片段中使用的变量必须在种子程序上下文信息中被定义或在新生成的代码片段中被定义. 通过约束条件的设计, 能够提升生成代码片段的有效性, 进而提升生成测试程序的有效性.

图 7 展示了 LumiX 生成代码片段的提示词示例. 在该示例中, LumiX 首先将大模型设定为精通程序合成、对 JVM 以及 JDK 有深入了解的专家角色, 从而确保大模型能够从专业的角度出发, 准确地生成能够覆盖新特性的代码片段. 其次, LumiX 在提示词中明确指定了生成的具体任务, 即生成能够覆盖语言新特性使用描述 (“FEATURE”) 的代码片段, 同时生成的代码片段必须满足约束条件 (“CONSTRAINTS”) 来保障语法和语义正确. 在生成的过程中, 可以通过复用种子程序中的元素信息 (“REUSEABLE PROGRAM ELEMENTS”) 来提升生成代码片段与种子程序的交互. 需要注意的是, LumiX 构造的提示词中仅包含可复用的程序元素, 而非整个种子程序. 其主要原因在于: 一方面, 如第 3.2.1 节所述, 复杂的提示词会导致大模型幻觉的累积, 影响生成代码片段的有效性; 另一方面, 复杂的提示词还会导致大模型推理成本过高, 从而影响生成代码片段的效率. 最后, LumiX 通过设定生成代码片段的

输出格式, 确保其结构化和规范化, 避免生成其他冗余信息影响后续测试程序的合成.

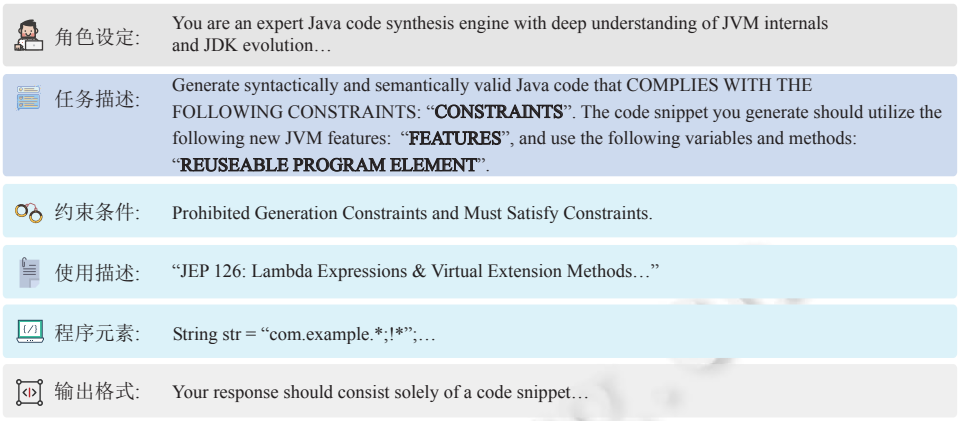

图 7 代码片段生成提示词示例

3.2.3 测试程序生成

在新特性代码片段生成之后, LumiX 会将其直接插入至种子程序中随机选择的插入点中, 从而生成完整的测试程序. 具体来说, LumiX 会将大模型返回的代码片段解析为 AST, 随后将其插入至种子程序 AST 中被选中的插入点之前. 相较于现有工作, 如 JavaTailor、VECT 需要为插入的代码片段设计额外的被破坏约束修复策略, LumiX 生成的代码可以直接插入到指定的插入点之前, 而无需额外的修复策略. 其主要原因在于: 一方面, LumiX 严格约束代码片段使用的程序元素必须在上下文信息中被定义或在新生成的代码片段中被定义; 另一方面, LumiX 设计了大量的约束规则保障生成代码片段的有效性. 最后, LumiX 将插入新生成代码片段之后的 AST 转换为源码, 生成新的测试程序.

3.3 测试程序编译与执行

在 Java 生态中, 语言新特性的实现不仅依赖于 Java 编译器的支持, 还依赖 Java 虚拟机对相应字节码的正确解析与执行. 因此, 语言新特性的缺陷可能出现在 Java 编译器生成字节码的过程中, 也可能出现在 JVM 对生成字节码的解析和执行过程. 为了系统性地发现这些缺陷, 对于新生成的测试程序, LumiX 采用双层差分测试的机制, 即分别使用不同的 Java 编译器和 JVM 对其进行编译和执行, 以检测它们中潜在的缺陷.

图 8 展示了双层差分测试的示意图.

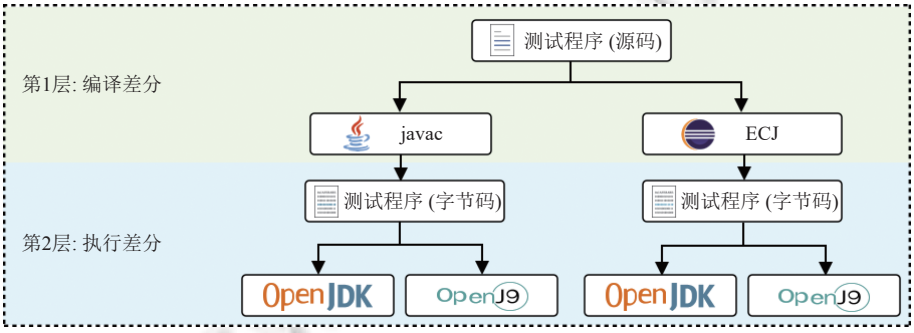

图 8 双层差分测试示意图

具体来说, 对于新生成的测试程序, LumiX 首先使用 javac 和 ECJ 两个 Java 编译器进行编译, 如图 8 中的第 1 层差分所示. 如果两个不同的编译器实现在编译阶段出现不一致行为, 例如, 对于一个测试程序, 一个 Java 编译器正

常编译, 而另一个抛出语法错误, 则表明两个 Java 编译器存在不一致行为. 对于此类不一致行为, LumiX 则会直接输出不一致报告以及相应的揭错测试程序. 如果在编译差分阶段均未出现不一致行为, 且测试程序在两个 Java 编译器上均成功编译, 则两个 Java 编译器均会生成对应的二进制字节码程序. 随后, LumiX 则会使用不同的 JVM 实现 (HotSpot 和 OpenJ9) 分别执行两个编译器编译生成的字节码测试程序, 如图 8 中第 2 层差分所示. 如果在执行阶段出现不一致行为, 即在两个编译器编译生成的字节码测试程序上的 4 次 JVM 执行中出现任意一个与其他执行结果不一致的行为, LumiX 将会输出不一致报告以及相应的测试程序. 如果在编译和执行结束之后均未出现不一致行为, 且测试程序可以正常地执行至程序结束, LumiX 则会将该测试程序放回至种子程序池中, 用于接收其他新特性的代码片段.

4 实验设计与结果分析

本节将通过收集 Java 新特性并在两款主流的 Java 编译器以及两款主流的 Java 虚拟机上进行实证研究, 来检验本文方法和工具的有效性.

4.1 研究问题

为了评估本文所提方法的有效性, 本文设计了以下 4 个研究问题.

RQ1: LumiX 是否能有效地生成触发 Java 新特性缺陷的测试程序?

RQ2: LumiX 是否能够取得更高的 Java 编译器及 JVM 代码覆盖率?

RQ3: LumiX 是否能够检测出 Java 编译器及 JVM 中与 Java 新特性相关的未知缺陷?

RQ4: LumiX 的各个组件在生成针对 Java 新特性的测试程序方面是否有效?

4.2 实验设置

4.2.1 语言新特性

为了对语言新特性进行测试, 本文选取 4 个长期支持 (long-term support, LTS) 版本的 OpenJDK (8、11、17 和 21) 作为测试版本, 分别收集对应 Java 8、Java 11、Java 17 和 Java 21 中可以用于测试的新特性. 具体来说, 每个 Java 版本对应一个特定的 Java 编程语言规范, 而 OpenJDK 则是该规范的标准开源实现, 代表了对应 Java 版本的实际实现. 此外, 本文同样收集了 Java 9、Java 10、Java 12、Java 13、Java 14、Java 15 以及 Java 16 中引入的新特性. 尽管这些 Java 版本对应的 OpenJDK 并不是 LTS 版本, 但由于不同的 Java 版本需要向前兼容, 因此这些 Java 版本中包含了部分在后续版本中依旧存在且可以用于本实验测试的新特性. 例如, 在 Java 9 中引入的反序列化过滤器 (ObjectInputFilter) 特性在后续版本中仍被保留, 可以用于 Java 11、Java 17 和 Java 21 中关于该特性实现的测试. 最终, 在对收集到的 Java 语言新特性进行过滤和总结之后, 本文共整理出 52 个可用于测试的新特性描述, 如表 1 所示.

表 1 待测新特性描述

数量	Java 8	Java 11	Java 17	Java 21	总计
新特性数量	10	16	14	12	52
版本测试数量	10	26	40	52	—

表 1 中, “新特性数量”表示 4 个 LTS Java 版本中收集整理到的新增语言特性数量. 每个 Java 版本所统计的新特性, 均为其前一个 LTS 版本发布之后、到当前版本发布之间所引入的全部新特性之总和. 例如, Java 11 的新特性数量指的是 Java 9–Java 11 之间引入的所有新特性; Java 17 对应的是 Java 12–Java 17 之间的新特性; 以此类推. “版本测试数量”则表示可用于当前 Java 版本进行测试的新特性描述数量. 由于 Java 向前兼容的特性, 后续版本通常保留并支持之前版本中引入的语言特性. 因此, 较新版本的测试能力是递增的. 例如, Java 11 可用于测试 Java 8 引入的新特性, Java 21 则能够覆盖之前所有版本中的语言新特性. 因此, Java 21 中的“版本测试数量”为全部新特性的总数.

4.2.2 种子程序及测试对象

文献 [23,24] 研究表明, 种子程序能够为生成的代码片段提供丰富的上下文信息, 从而提升生成测试程序的揭

错能力. 为此, 本实验从现有工作中收集了历史揭错测试程序作为种子程序[34]. 这些测试程序由现有工作从 OpenJDK 仓库中收集整理, 用于分析 JVM 缺陷在不同模块中的分布以及对应模块测试程序的程序特征. 为确保种子程序的有效性, 本实验对原始数据集进行了清洗, 过滤了其中存在编译错误、执行异常以及可以直接触发 Java 编译器和 JVM 不一致行为的测试程序, 仅保留其余可以正常编译并执行的测试程序作为种子程序集合.

最终, 本实验收集到共计 390 个历史揭错测试程序作为种子程序, 种子程序信息如表 2 所示. 表 2 中展示了不同种子程序所测试的模块、各模块下种子程序的数量、所有种子程序的代码行数量、定义的变量和函数数量. 通过对种子程序的分析可以发现, 本实验所使用的种子程序覆盖了 HotSpot 的 3 个核心模块, 分别是 Compiler、Runtime 以及 GC (garbage collection). 根据已有工作的分析, 这些测试程序针对不同模块的功能进行设计, 具有各自显著的程序结构特征. 此外, 本文对种子程序的结构特征做了进一步分析, 结果表明: 所有种子程序中共计包含 7071 个变量定义和 1 154 个函数定义. 这些丰富的程序特征以及变量和函数定义, 为新生成的代码片段提供了丰富的上下文信息, 有助于生成更高质量的测试程序.

表 2 种子程序描述

数据集名称	模块	种子程序数量	代码行数量	变量定义数量	函数定义数量
OpenJDKTests	Compiler	352	25 873	6 614	1 078
	Runtime	25	462	88	57
	GC	13	696	369	19
总计		390	27 031	7 071	1 154

本文在 4 个长期支持的 OpenJDK 版本 8、11、17 以及 21 上进行测试, 分别选择两款主流的 Java 编译器 javac 和 ECJ 以及两款主流的 Java 虚拟机 HotSpot 和 OpenJ9 作为测试对象. 其中, javac 是 OracleJDK 和 OpenJDK 中提供的官方 Java 编译器, 而 ECJ 则是 Eclipse 项目中的 Java 编译器实现, 被广泛地应用于 Eclipse IDE 和相关的工具链之中. 相似地, HotSpot 是 OracleJDK 和 OpenJDK 中默认的 JVM 实现, 以其性能优化 (如 JIT 编译器、逃逸分析等) 而闻名, 是目前最主流的 Java 执行引擎之一. OpenJ9 是由 IBM 维护的高性能 JVM 实现, 起源于 IBM J9 VM, 现为 Eclipse 基金会的一部分. OpenJ9 以其低内存开销和快速启动时间等优势, 目前广泛应用于云计算和容器化环境中.

表 3 展示了本实验所使用的 Java 编译器和 JVM 的详细信息.

表 3 测试对象描述

OpenJDK版本	测试对象		历史版本	最新版本
OpenJDK 8	Java编译器	javac ECJ	1.8.0_302 3.10.0	1.8.0_442 3.41.0
	Java虚拟机	HotSpot OpenJ9	25.211-b12 0.27.0	25.431-b10 0.49.0
OpenJDK 11	Java编译器	javac ECJ	11 3.16.0	11.0.26 3.41.0
	Java虚拟机	HotSpot OpenJ9	11+28 0.27.0	11.0.25+9 0.49.0
OpenJDK 17	Java编译器	javac ECJ	17 3.28.0	17.0.14 3.41.0
	Java虚拟机	HotSpot OpenJ9	17+35 0.29.1	17.0.13+10 0.49.0
OpenJDK 21	Java编译器	javac ECJ	21 3.36.0	21.0.6 3.41.0
	Java虚拟机	HotSpot OpenJ9	21+35 0.42.0	21.0.5+9 0.49.0

具体来说, 本实验分别选择了每个 Java 编译器和 JVM 实现的历史版本和最新版本作为测试对象. 其中, 历史

版本的实现中包含更多的历史缺陷, 有助于使实验结果更具显著性. 最新版本的实现则是本实验进行时各个编译器和虚拟机实现官方发布的最新版本. 最新版本一方面可以用于评估历史版本中检测到的缺陷是否被修复, 例如, 生成的测试程序在历史版本上触发了不一致, 但不一致在最新版本上消失, 则表明历史缺陷被修复; 另一方面, 可以用于检测最新版本中是否存在未知的缺陷, 例如, 最新版本上依然存在的不一致有很大可能是现有测试工作中尚未被发现的缺陷, 需要提交给开发人员做进一步的分析. 需要注意的是, ECJ 编译器并不会针对每个 Java 版本单独发布对应的编译器版本, 而是在一个发布版本中通过参数支持多个 Java 语言版本的编译. 例如, 为了生成符合 Java 8 规范的测试程序, 只需在编译时指定“-release 8”参数, 即可确保 ECJ 使用 Java 8 的编程语言规范对测试程序进行编译. 因此, 在 ECJ 的最新版本中, 所有发布的 ECJ 版本均为 3.41.0. 对于历史版本, 本文分别选取了 ECJ 最早支持对应 Java 版本的发布版本, 以确保测试程序能够被正常地编译执行.

4.2.3 对比方法及评价指标

根据本文调研, 现有针对语言新特性的测试程序生成方法较少. 考虑到 LumiX 在源代码级别生成测试程序, 本文选择同样在源码上生成测试程序的 JavaFuzzer[5]进行对比, 以评估 LumiX 的有效性. JavaFuzzer 是一个基于语法的 Java 测试程序生成工具, 其设计了大量关于 Java 语法的生成规则, 包括循环结构、条件语句、函数调用以及各种复杂的算术表达式等. 此外, 本文将 LumiX 与通用模糊测试工具 Fuzz4All[21]进行对比. Fuzz4All 是首个利用大语言模型作为输入生成与变异引擎的通用模糊测试方法, 能够支持多语言及多特征测试, 已在 Java 等语言上取得一定成效, 可为新特性测试提供参考. 为了进一步评估 LumiX 中各个组成部分的有效性, 本文设计了 LumiX 的变体方法 LumiJ. 具体说来, LumiJ 将 LumiX 中的历史揭错种子程序替换为 JavaFuzzer 随机生成的测试程序, 而保持其他组成部分不变. 设计该变体的目的在于评估 LumiX 中历史揭错种子程序以及大模型生成的新特性代码片段的有效性. 为了进行公平的对比, 本实验进一步利用 JavaFuzzer 生成了 390 个 (与历史揭错种子程序数量相同) 随机测试程序作为 LumiJ 的种子程序, 累计包含 79 956 行代码、22 568 个变量定义和 1 987 个函数定义.

为了评估不同方法的有效性, 本实验设计了如下 3 个评价指标.

(1) 不一致缺陷数量

在本实验中, 不一致缺陷数量包括不同方法在编译和执行阶段的不一致行为数量. 在编译阶段, 不一致缺陷数量表示两个 Java 编译器在同一个测试程序的编译结果不一致的数量. 在执行阶段, 不一致缺陷数量则表示两个 JVM 对同一个测试程序的执行结果不一致的数量. 通常来说, 编译结果的差异体现为编译错误信息的不一致, 因此, 本实验使用编译错误信息作为不一致的标志对编译阶段的不一致缺陷进行去重. 而执行阶段的不一致则通常体现为程序执行异常不一致或程序输出结果不一致. 为此, 本实验参考现有工作: 对于异常不一致, 根据程序执行过程中触发的异常信息或崩溃信息进行去重; 对于输出结果不一致, 本实验利用 Bug-inducing Commit 方法进行去重. 最终, 去重后的编译阶段和执行阶段的不一致缺陷数量总和即为不一致缺陷数量评价指标.

(2) 代码覆盖率

本实验代码覆盖率包括不同方法在编译和执行阶段的代码覆盖率, 分别为编译器代码覆盖率和 JVM 代码覆盖率. 具体来说, 本实验选择 OpenJDK 21 下的 javac 编译器和 HotSpot 虚拟机作为代表, 分别收集不同方法在 javac 编译器和 HotSpot 上的代码覆盖率. 为了收集 javac 编译器的代码覆盖率, 本实验使用 Java 代码覆盖率收集工具 (javac 编译器基于 Java 语言开发) JaCoCo[39], 对编译阶段的代码进行覆盖率的收集. 为了收集 HotSpot 的代码覆盖率, 本实验对 OpenJDK 21 (build 21-internal-adhoc.quand.openjdk21) 进行编译, 并在编译时开启“-enable-native-coverage”选项. 随后, 利用 Gcov[40]和 Lcov[41]来收集和分析每种方法在 HotSpot 上取得的代码覆盖率.

(3) 未知缺陷数量

本实验未知缺陷的数量同样包括编译和执行阶段检测到的未知缺陷数量. 为了评估 LumiX 是否能检测出未知缺陷, 本实验在测试对象的最新版本上运行 LumiX 进行测试. 对于在最新版本上发现的编译和执行不一致缺陷, 本实验将对测试程序进行约简, 最后将约简后的测试程序分别提交给对应开发人员. 最终, 根据开发人员的反馈来判定 LumiX 检测到的缺陷是否为未知缺陷, 并统计其中被确认并修复的缺陷数量, 以评估 LumiX 在检测未知缺陷方面的有效性.

4.2.4 方法实现及实验过程

本文基于广泛被使用的 OpenJDK 8 来实现 LumiX. 为了识别种子程序中可复用的程序元素, LumiX 使用 JavaParser[42]对种子程序进行分析. JavaParser 是一个开源的 Java 源码分析工具, 支持将 Java 源码解析为 AST, 并提供了丰富的 API 用于对 AST 的操作. 具体来说, LumiX 使用 JavaParser 将 Java 源码解析为 AST, 提取出其中可复用的程序元素信息, 最后将插入新代码片段的 AST 转换为源代码, 从而生成新的测试程序. 为了总结新特性使用描述以及生成新的代码片段, LumiX 使用通义千问-Max (Qwen-Max-0125) 大模型[43,44]. 通义千问 2.5 是阿里云发布的大语言模型, 是本实验进行时通义千问系列效果最好的大模型, 具有强大的文本理解、代码生成以及逻辑推理能力. 为了避免生成不合法或包含死循环、无限递归等容易引入误报的程序特征, LumiX 一方面在构造的提示词中加入严格的约束条件; 另一方面设置程序的编译和执行时间为 30 s, 当生成的测试程序执行时间超过时间限制时, LumiX 将会终止编译或执行并输出超时报告.

为了回答研究问题 1, 本实验对比了 LumiX、LumiJ、JavaFuzzer 以及 Fuzz4All 在 4 个 OpenJDK 历史版本上的缺陷检测效果, 包含编译阶段 (javac、ECJ) 和执行阶段 (HotSpot、OpenJ9) 的唯一不一致缺陷数量. 为保证实验的公平性, 本文对 Fuzz4All 进行了适配: 在其文档中补充 JEP 增强提案, 使其具备对语言新特性的支持; 同时, 实验统一采用通义千问-Max (Qwen-Max-0125) 作为大语言模型. 对于所有方法, 测试时间统一设置为 24 h. 为了回答研究问题 2, 本实验选择 OpenJDK 21 作为测试对象的代表, 在 OpenJDK 21 上收集不同方法的 Java 编译器代码覆盖率以及 JVM 代码覆盖率. 考虑到分支覆盖与函数覆盖通常与代码行覆盖表现为相似的分布, 因此本文仅对比不同方法在代码行覆盖上的差异. 覆盖率实验的运行时间同样为 24 h, 在不同方法运行过程中, 本实验以 1 h 为间隔分别收集 Java 编译器和 JVM 的代码覆盖率. 为了回答研究问题 3, 本实验将 LumiX 应用于最新版本 Java 编译器和 JVM 的测试中, 即表 3 中“最新版本”列对应的 Java 编译器和 JVM 版本, 并统计检测到的未知缺陷数量. 为了回答研究问题 4, 本实验以 OpenJDK17 以及 OpenJDK 21 作为测试对象, 对 LumiX 的不同关键组件进行消融. 消融实验针对 LumiX 的关键部分, 即新特性使用描述总结及代码片段生成进行消融. 此外, 本实验通过替换使用的大模型, 来探索不同大模型对 LumiX 效果的影响. 本实验使用的大模型均通过相应厂商提供的 API 进行访问. 除了 LumiX 使用的通义千问-Max (Qwen-Max-0125) 大模型, 在研究问题 4 中, 本实验还对比了 DeepSeek-V3[45]、Qwen2.5-72b、Qwen2.5-7b 来对比不同模型以及不同参数大小对 LumiX 效果的影响. 需要注意的是, 为了确保公平性, 本实验参数设置均参考已有研究的最佳实践 (如模型温度设为 0.7) 进行设置[46]. 已有研究表明, 更优的参数设置或提示词设计, 如 CoT 推理及 In-context learning 等策略能够提升模型表现[47–50], 这些手段同样有望增强 LumiX 的效果. 本文将在后续研究中探索和引入这些策略, 以更进一步地提升方法性能.

本文所有实验均在同一台服务器上进行, 服务器具有两个 12 核 CPU Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz, 125 GB 内存, 操作系统为 Ubuntu 20.04.6 LTS (64 位).

4.3 结果分析

基于上述实验设置, 本节将对 LumiX 与其他对比方法的实验结果进行分析, 以回答本文的研究问题.

4.3.1 研究问题 1: 历史缺陷检测效果分析

表 4 展示了 LumiX、JavaFuzzer、Fuzz4All 和 LumiJ 在 4 个 OpenJDK 版本上不同测试对象上的检测效果. 本实验对每个方法检测到的不一致缺陷做了进一步的区分, 分别包括 Java 编译器不一致缺陷以及 Java 虚拟机 (JVM) 不一致缺陷, 其中加粗数字为在对应测试对象下表现最优的方法 (下同). 需要注意的是, 表 4 中的不一致缺陷数量均为去重后的不一致缺陷数量.

从表 4 中可以看出, LumiX 在所有 OpenJDK 版本上的效果均优于现有工作, 在 4 个 OpenJDK 版本的 Java 编译器和 JVM 上累计检测到 25 个不一致缺陷. 而 JavaFuzzer 在所有的 OpenJDK 版本上仅检测到 3 个 JVM 不一致缺陷, 且未在 Java 编译器上检测到不一致缺陷. Fuzz4All 在所有的版本上仅检测到 6 个编译器不一致, 但未在 JVM 上检测到任何不一致缺陷. LumiJ 则在所有的 OpenJDK 版本上一共检测到 17 个不一致缺陷. 首先, 通过对比 LumiX 和 JavaFuzzer 的结果可以发现, LumiX 比 JavaFuzzer 多检测出 22 个不一致缺陷, LumiX 效果显著优于

JavaFuzzer; 同时, LumiX 在 Java 编译器和 JVM 上均能检测出一定数量的不一致缺陷, 而 JavaFuzzer 仅能在 JVM 上检测出少量的不一致缺陷. 这一对比结果表现出 LumiX 的有效性以及现有工作对 Java 编译器和 JVM 测试的不充分. 其次, 通过对比 LumiX 和 LumiJ 的结果可以发现, LumiX 比 LumiJ 多检测出 8 个不一致缺陷, 缺陷检测效果提升 47.0%. 这表明 LumiX 使用历史揭错测试程序作为种子程序的有效性. 尽管这些历史揭错测试程序中并不包含 Java 新特性的程序特征, 但历史揭错测试程序中包含了大量历史揭错的程序特征, 这些特征结合新特性的代码片段, 有助于生成复杂性更高的新特性测试程序. 最后, 通过对比 LumiJ 和 JavaFuzzer 的结果可以发现, LumiJ 比 JavaFuzzer 多检测出 14 个不一致缺陷, LumiJ 的效果同样显著优于 JavaFuzzer. 考虑到 LumiJ 中的种子程序为 JavaFuzzer 生成的随机测试程序, 这进一步表明了 LumiX 中方法的有效性. 同时, 这也表明 LumiX 可以增强现有工具生成的测试程序, 从而使其具备对语言新特性进行测试的能力.

表 4　不一致缺陷数量对比分析

OpenJDK 版本	LumiX		JavaFuzzer		Fuzz4All		LumiJ	
	编译器	虚拟机	编译器	虚拟机	编译器	虚拟机	编译器	虚拟机
OpenJDK 8	**2**	**3**	0	1	2	0	1	1
OpenJDK 11	3	**4**	0	0	1	0	3	1
OpenJDK 17	**5**	4	0	1	2	0	3	4
OpenJDK 21	3	1	0	1	1	0	3	1
总计	**13**	**12**	0	3	6	0	10	7

本实验进一步分析了 LumiX 显著优于 JavaFuzzer 以及 Fuzz4All 的原因. 对于 JavaFuzzer, 其设计的语法规则均为较基础的 Java 语法, 且依赖随机生成策略, 难以生成高复杂度的测试程序, 因而导致生成程序在复杂性和多样性上的不足. 相比之下, LumiX 利用大模型生成新特性代码片段扩充了对新特性的覆盖, 提升了测试程序的多样性; 同时, LumiX 将生成的代码片段插入至历史揭错测试程序中, 并且增强与历史揭错测试程序上下文的交互, 进一步提升了生成测试程序的揭错能力. 此外, JavaFuzzer 需要从零构造新的测试程序, 这导致其生成效率较低. 例如, 在 OpenJDK 21 的实验中, 在 24 h 的测试时间中, LumiX 一共生成并执行了 3 919 个测试程序, 而 JavaFuzzer 仅生成并执行了 1 452 个测试程序. 现有研究工作表明[25], 生成效率直接影响方法效果, 因此 LumiX 在相同时间内能够生成并执行更多测试程序, 从而取得更优的表现.

相较于 Fuzz4All, 尽管其同样依赖大语言模型生成新特性测试程序. 然而, Fuzz4All 并未针对语言新特性进行系统性的总结, 即便本实验将语言新特性的文档作为补充输入给 Fuzz4All, 其检测效果仍然有限, 表现出对新特性的覆盖不足. 同时, 其生成的测试程序规模普遍偏小, 平均仅有 20 行, 语法与控制流复杂性不足, 导致测试程序的复杂性受限. 相比之下, LumiX 以历史揭错测试程序作为种子, 在此基础上利用大模型插入语言新特性相关的代码片段生成新的测试程序, 使得生成测试程序平均规模达到 115 行, 语法结构与控制流路径更加丰富, 从而能够更有效地揭示潜在缺陷.

4.3.2　研究问题 2: 代码覆盖率对比分析

图 9 展示了不同方法在 Java 编译器上取得的代码覆盖率随时间变化的曲线图. 从图 9 中可以看出, LumiX 相较于其他方法可以取得更高的覆盖率. 首先, 通过比较 LumiX 与 JavaFuzzer 和 Fuzz4All 的结果可以发现, LumiX 生成的测试程序可以取得更高的代码覆盖率, 表明现有方法对 Java 编译器的测试是不充分的. 其次, 通过对比 LumiX 和 LumiJ 的结果可以发现, 相较于 JavaFuzzer 随机生成的测试程序, 历史揭错测试程序有助于提升 Java 编译器的代码覆盖率. 这表明历史揭错测试程序中往往包含更复杂多样的程序特征, 有助于生成覆盖更多 Java 编译器功能模块的测试程序. 最后, 通过对比 LumiJ 和 JavaFuzzer 的结果可以发现, 大模型生成的新特性代码片段有助于大幅提升 Java 编译器的代码覆盖率. 其主要原因在于, 大模型生成的代码片段向种子程序中引入了新的程序元素, 从而扩展了测试空间并提高了代码覆盖. 上述结果也进一步解释了研究问题 1 中 LumiX 优于其他方法的原因. 值得注意的是, 不同方法取得的代码覆盖率随时间的变化曲线均表现出较为稳定的趋势, 这表明不同方法在程序测试的开始阶段就已经覆盖了大部分易于覆盖的代码模块. 而剩余尚未被覆盖的部分通常需要更智能化的引导策

略, 例如, 覆盖率引导的测试程序生成方法[51]. 这也促使后续的研究工作探索在大模型生成中引入覆盖率引导的策略, 以进一步提升代码覆盖率, 进而增强缺陷检测效果.

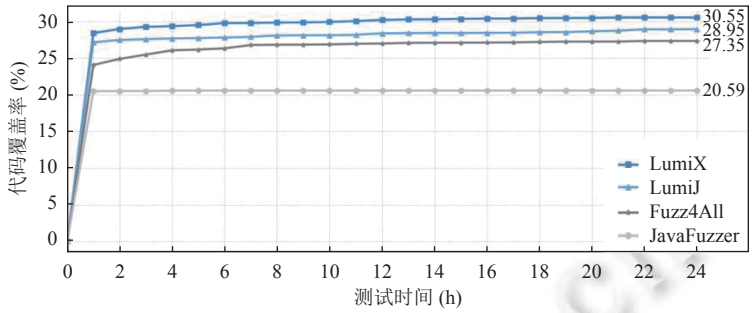

图 9 Java 编译器代码覆盖率随时间变化曲线图

图 10 展示了不同方法在 JVM 上取得的代码覆盖率随时间变化图. 从图 10 中可以看出, LumiX 相较于其他方法同样可以取得更高的代码覆盖率. 不难看出, 图 10 和图 9 的覆盖率结果具有一定的相似性, 因此可以得出与图 9 相似的结论. 值得注意的是, Fuzz4All 在 JVM 上的代码覆盖率远小于其他方法, 尽管其在 Java 编译器上的覆盖率高于 JavaFuzzer, 但受限于其生成的代码规模较小, 所以其在检测 JVM 缺陷上的效果很差. 进一步分析发现, Fuzz4All 生成的测试程序能覆盖新特性语法但控制结构较为简单, 而 JavaFuzzer 虽缺乏新特性相关语法, 却包含更复杂的控制结构 (如循环嵌套, 可触发 JVM 的 JIT 优化). 在未来的研究工作中可结合两者优势, 以提升缺陷检测效果. 此外, LumiX 与 JavaFuzzer 在 JVM 上的代码覆盖率差异相较于 Java 编译器上的差异更小. 例如, 在 Java 编译器上, 不同方法执行结束后, LumiX 的代码覆盖率相较于 JavaFuzzer 提升了 9.96%. 而在 JVM 上, 在方法执行结束后, LumiX 的代码覆盖率相较于 JavaFuzzer 仅提升了 7.6%. LumiX 相较于 LumiJ 在两者代码覆盖率上的差异变化不大, 这表明两者在此场景下的覆盖提升能力较为接近. 这也进一步表明, 语言新特性的体现往往更集中于 Java 源码层面, 因此, 基于源码构造面向语言新特性的测试程序是更直接、有效的, 有助于提升编译阶段的测试覆盖能力.

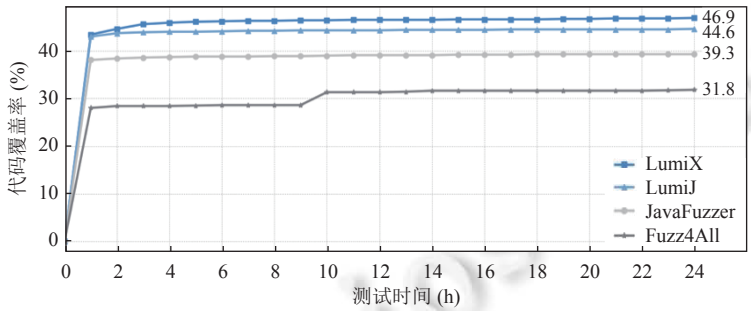

图 10 JVM 代码覆盖率随时间变化曲线图

4.3.3 研究问题 3: 未知缺陷检测效果分析

表 5 展示了 LumiX 在最新版本的 Java 编译器和 JVM 上检测到的被确认及修复的未知缺陷. 其中, 缺陷 ID 列指开发人员为本文提交缺陷分配的标识符; 测试对象列为当前缺陷所影响的实现; 影响版本为缺陷影响的 OpenJDK 版本; 影响功能/特性则表明了当前缺陷是否是新特性相关缺陷, 如果不是, 则统计其影响的功能模块. 总的说来, LumiX 一共向不同的 Java 编译器和 JVM 开发人员提交了 16 个未知缺陷, 其中 12 个已经被确认和修复. 其中, 部分缺陷在 Java 编译器中长期存在, 直至本实验才被成功检测出来. 例如, 第 2 节提到的 Bug #3792, 该缺陷自 Lambda 表达式引入以来就一直存在, 且未能在官方的测试过程中被及时发现. 这表明当前对 Java 编译器部分

语言新特性的测试依然存在局限性, 特别是长期遗留缺陷的覆盖不足. 此外, 这也进一步强调了增强对引入新特性测试的必要性. 从表 5 中可以看出, LumiX 不仅可以检测出 Java 编译器中新特性相关的缺陷, 而且可以检测出 JVM 中的与新特性相关的未知缺陷. 这表明新特性相关缺陷不仅仅局限于编译器的解析过程中, 同样会出现在运行时. 同时, 这也表明了 LumiX 采用双层差分测试的有效性.

表 5 确认及修复的未知缺陷

编号	缺陷ID	测试对象	影响版本	影响功能/特性	状态
1	Bug #3789	ECJ	OpenJDK 8	JEP 259	Confirmed
2	Bug #3790	ECJ	OpenJDK 21	JEP 126	Confirmed
3	Bug #3792	ECJ	OpenJDK 21	JEP 126	Fixed
4	Bug #3826	ECJ	OpenJDK 8	Compiler	Confirmed
5	Bug #4031	ECJ	OpenJDK 21	JEP 378	Fixed
6	Bug #21417	OpenJ9	OpenJDK 11, 17, 21	Runtime	Confirmed
7	Bug #21419	OpenJ9	OpenJDK 11, 17, 21	JEP 264	Fixed
8	Bug #21872	OpenJ9	OpenJDK 21	JIT	Confirmed
9	Bug #21886	OpenJ9	OpenJDK 21	JEP 121	Fixed
10	Bug #21931	OpenJ9	OpenJDK 21	JEP 264	Fixed
11	Bug #8356696	javac	OpenJDK 17, 21	JEP 181	Confirmed
12	Bug #8356989	HotSpot	OpenJDK 21	JEP 431	Fixed

接下来, 本实验通过两个具体的例子来进一步说明 LumiX 在检测新特性未知缺陷方面的有效性.

图 11 展示了 LumiX 生成的测试程序. 该测试程序成功检测出了 Java 编译器 ECJ 中的一个编译缺陷 (Bug #3790). 具体来说, 该测试程序通过设置全局的反序列化过滤器工厂, 用于控制 Java 反序列化过程中可接收的类. 为此, 该测试程序在第 3 行设置了一个 Lambda 表达式作为 setSerialFilterFactory 方法的参数. 该表达式包含两个参数, 分别为反序列化时的上下文对象 (context) 和默认过滤器 (currentFilter). 随后, 在 Lambda 表达式中, 测试程序在第 5 行通过对比 context 和 Object 类型是否相等, 来判断是创建一个新的过滤器还是使用默认的过滤器. 然而, 该类型对比操作是错误的, 因为 context 是泛型类型, 而 Object 是引用类型. 但在使用不同的 Java 编译器 (即 javac 和 ECJ) 对该测试程序进行编译时, javac 正确地报告了编译错误, 而 ECJ 则成功地编译了该测试程序, 并且生成了一个错误的 class 文件. 需要注意的是, 图 11 中浅蓝色标记部分为 LumiX 向种子程序中插入的包含语言新特性的代码片段. 其中, Lambda 表达式是在 Java 8 中引入的新特性, 而反序列化过滤器工厂 (java.io.ObjectInputFilter) 则是在 Java 9 中引入的新特性. LumiX 迭代地生成包含新特性的测试程序, 最终生成了触发了不同 Java 编译器不一致行为的测试程序. 经过进一步的分析, 本文向 ECJ 编译器的开发人员提交了该缺陷报告. 目前, ECJ 的开发人员已经对该缺陷进行了复现和确认.

```
1  private static final long serialVersionUID = 1L;
2  public static void main (String[] args) {
3      java.io.ObjectInputFilter.Config.setSerialFilterFactory ((context, currentFilter) → {
4          …
5          if (context.getClass() == Object.class) {  //不兼容类型对比
6              return ObjectInputFilter.Config.createFilter("com.example.*;!*");
7          }
8          return currentFilter;
9      });
10     try {
11         … //覆盖上述代码
12     } catch (Exception e) {e.printStackTrace();}
13 }
```

图 11 ECJ 编译器缺陷 (Bug #3790)

图 12 展示了 LumiX 生成的测试程序. 该测试程序成功检测出了 Java 虚拟机 OpenJ9 中的缺陷 (Bug #21419). 其中, 蓝色标记的代码区域为 LumiX 生成的新特性代码片段, 该代码片段中 String.Logger 是 Java 11 中引入的新

特性 (JEP 264[52]). 在该测试程序中, LumiX 将新生成的代码片段插入在函数 mainTest 中, 然后, 该函数会在第 6、7 行通过反射无限递归地调用 mainTest 函数本身. 通过利用不同的 JVM 执行该测试程序, HotSpot 会陷入无限递归调用的死循环, 而 OpenJ9 则会在执行一段时间后产生崩溃. 这表明 OpenJ9 在处理包含 String.Logger 的无限递归调用时存在潜在的缺陷. 本文随即将其提交给 OpenJ9 的开发人员. 目前, 开发人员已经对缺陷进行了修复.

```
1  public static void mainTest(String[] strArr1) {
2      for (int i = 1; i < 5; i++) {
3          System.Logger logger = System.getLogger(Test.class.getName());
4          logger.log(System.Logger.Level.INFO, "Logging example");
5          try {
6              java.lang.reflect.Method method = Test.class.getMethod("mainTest", String[].class);
7              method.invoke(this, (Object)strArr1);
8          } catch(NoSuchMethodException e) {
9              … //省略程序异常语句
10         }
11     }
12     …
13 }
```

图 12 OpenJ9 缺陷 (Bug #21419)

4.3.4 研究问题 4: 关键组件消融效果分析

为了评估 LumiX 不同组件对缺陷检测的贡献, 本实验分别对新特性使用描述总结组件以及代码片段生成组件进行消融. 具体来说, 分别使用新特性完整描述替换使用新特性使用描述总结以及生成完整的测试程序替换生成代码片段, 来验证两个组件对 LumiX 效果的影响. 此外, 本实验进一步探索了不同大模型对 LumiX 效果的影响. 接下来本实验将分别对这 3 组实验的结果进行分析.

表 6 展示了使用描述总结组件的消融实验对比分析结果, 其中 w/o LumiXsum 指使用新特性完整描述替换使用描述总结的方法. 为了从多角度评估方法的有效性, 本实验进一步设计了有效性指标及平均词元 (token) 指标. 其中, 有效性指标代表方法生成可以正确编译, 或错误编译但触发编译器不一致的测试程序数量占总生成测试程序数量的比例. 如果生成的测试程序包含语法错误且在不同的编译器之间语法错误一致, 则表明生成的测试程序无效. 有效性指标越高代表方法生成的测试程序质量越高. 平均词元指标代表生成单个测试程序需要消耗的平均词元数量, 平均词元的值越高代表生成测试程序的成本越高. 通过分析表 6 中的结果可以发现, 两个方法在检测到的不一致数量上没有显著的差异, LumiX 仅在 Java 编译器上比 w/o LumiXsum 多检测出一个缺陷. 其主要原因在于, 一方面, LumiX 中的使用描述总结同样是基于大模型进行总结, 在文本理解和语义抽取上具备相似的能力; 另一方面, 大语言模型本身具备一定程度的通用知识和代码相关背景知识, 因此在该任务中的表现趋于一致, 从而导致两者在缺陷检测能力上的整体差异较为有限.

表 6 使用描述总结组件消融实验对比分析

OpenJDK 版本	LumiX				w/o LumiXsum			
	编译器	虚拟机	有效性 (%)	平均消耗词元	编译器	虚拟机	有效性 (%)	平均消耗词元
OpenJDK 17	**5**	**4**	**82.38**	**1 298.7**	3	3	59.54	3 447.1
OpenJDK 21	3	1	**82.31**	**1 328.8**	3	**2**	60.78	3 614.0
总计	**8**	5	—	—	6	5	—	—

然而, 在移除使用描述总结组件后, LumiX 在生成测试程序的有效性以及平均词元指标上效果较差. 例如, OpenJDK 17 的实验中, w/o LumiXsum 在有效性指标上相较于完整的 LumiX 下降了 22.84%. 同时, w/o LumiXsum 生成单个测试程序需要消耗的平均词元显著增长. 例如, 在 OpenJDK 的实验中, w/o LumiXsum 在平均词元指标上相较于 LumiX 增加了 165.4%. 这表明, 过长且未经提炼的新特性使用描述可能导致大模型难以聚焦于核心要点, 从而影响生成测试程序的质量; 同时, 冗长的输入也显著增加了测试生成过程中的计算成本和资源消耗. 总的说

来, 该实验结果验证了 LumiX 中新特性使用描述总结组件的有效性: 该组件能够在减少词元消耗的同时, 有效提升测试程序的生成质量.

表 7 展示了代码片段生成组件的消融实验对比分析结果, 其中 w/o LumiXseg 指直接生成完整的新特性测试程序而非代码片段的测试程序生成方法. 在评价指标方面, 本实验进一步设计了针对代码生成的评价指标: 平均生成长度. 该指标通过统计大模型生成测试程序的平均词元长度, 从而衡量其生成测试程序的大小. 平均生成长度的指标越小则表明其生成的测试程序越精简. 需要注意的是, 在 LumiX 中, 本实验仅统计大模型生成的代码词元长度而不包含其插入的种子程序的长度. 其主要原因在于, 从分析成本上看, 在对检测到的缺陷进行人工约简和定位分析的工作中, 基于 LumiX 生成的测试程序可以很快定位到代码片段级别, 而无需对整个测试程序进行分析. 因此, 本实验仅统计大模型生成的代码词元长度.

表 7　代码片段生成组件消融实验对比分析

OpenJDK 版本	LumiX				w/o LumiXseg			
	编译器	虚拟机	有效性 (%)	平均生成长度	编译器	虚拟机	有效性 (%)	平均生成长度
OpenJDK 17	**5**	**4**	**82.38**	**625.0**	3	0	33.78	2 204.7
OpenJDK 21	**3**	**1**	**82.31**	**611.6**	1	1	31.49	2 241.8
总计	**8**	**5**	—	—	4	1	—	—

通过分析表 7 中的结果可以发现, LumiX 在缺陷检测上的效果显著优于 w/o LumiXseg. 在两个 OpenJDK 版本的实验中, LumiX 累计检测到 12 个不一致缺陷, 而 w/o LumiXseg 仅检测到 5 个不一致缺陷. 通过进一步分析两个方法的有效性和平均生成长度指标发现, 在生成完整的测试程序后, w/o LumiXseg 生成的测试程序长度远高于 LumiX. 例如, 在 OpenJDK 17 的实验中, w/o LumiXseg 平均生成的测试程序长度是 LumiX 的 3.5 倍. 同时, 过长的生成内容导致大模型产生了幻觉的累积, 导致其生成代码的有效性显著降低. 在 OpenJDK 17 及 21 的实验中, w/o LumiXseg 生成测试程序的有效性相较于 LumiX 下降了 49% 以上. 这也进一步验证了现有工作中, 大模型在面对长文本或结构复杂任务时, 其准确性和语义一致性会显著下降, 生成代码越长, 正确性和有效性越低的结论. 随着有效性的降低, w/o LumiXseg 的不一致缺陷检测效果也随之显著降低. 总的说来, 该实验结果验证了 LumiX 中代码片段生成组件的有效性: 该组件可以有效提升生成测试程序的有效性, 从而提升不一致缺陷的检测能力.

表 8 展示了使用不同大模型替换 LumiX 中使用的通义千问-Max (Qwen-Max-0125) 大模型后的方法效果对比. 通过对比表 7 中原始 LumiX 的效果可以发现, 大模型的选择对 LumiX 的效果有一定的影响. 具体而言, 当将通义千问-Max 替换为 DeepSeek-V3 后, LumiX 在两个版本上共减少了 3 个缺陷的检测, 同时生成测试程序的有效性下降超过 13%. 值得注意的是, 虽然 Qwen2.5-72b 模型相较于通义千问-Max 效果较差, 但其在缺陷检测和有效性方面与 DeepSeek-V3 (模型参数 671B) 取得了相似的结果. 这表明在本任务中, 72B 规模的大模型已经能够取得较为理想的效果. 但随着模型参数的进一步减小, 性能下降趋势则更为明显. 在 Qwen2.5-7b 模型上, LumiX 的整体表现显著下降, 有效性最高仅为 27.17%. 尽管如此, 即便在参数规模较小的情况下, LumiX 仍然能够检测出一定数量的不一致缺陷. 总的来说, 该实验验证了 LumiX 方法本身的有效性, 同时也表明大模型的选择会在一定程度上影响其具体表现. 在实际测试应用中, 可根据任务需求与计算资源预算, 灵活选择合适规模的大模型, 以在效果与成本之间取得平衡.

表 8　大模型替换实验结果对比分析

OpenJDK 版本	DeepSeek-V3			Qwen2.5-72b			Qwen2.5-7b		
	编译器	虚拟机	有效性 (%)	编译器	虚拟机	有效性 (%)	编译器	虚拟机	有效性 (%)
OpenJDK 17	4	2	68.19	1	3	67.84	1	3	27.17
OpenJDK 21	2	1	68.68	4	1	73.21	2	1	26.37
总计	6	3	—	5	4	—	3	4	—

5 讨 论

本文提出基于大语言模型的 Java 新特性测试程序生成方法 LumiX. LumiX 通过利用大模型强大的文本理解及代码生成能力, 结合程序分析方法, 生成能够覆盖 Java 语言新特性的测试程序. 接下来本节将对可能影响到本文有效性的因素进行分析, 并对未来的研究工作进行展望.

5.1 有效性影响因素分析

● 内部有效性分析. 本文的内部有效性影响因素主要来自使用的模型及技术实现. 为了减少这些因素给方法有效性带来的影响, 本文均使用官方提供的 API 来访问大模型. 模型选择和参数设置可能影响本文方法的性能, 构成潜在威胁. 为缓解该威胁, 本文选取了通义千问-Max 和 DeepSeek 两种国内代表性模型进行实验, 它们在模糊测试、代码生成等任务中均表现良好. 参数设置则参考已有研究的建议实践进行设置. 同时, 本文程序相关部分均采用了成熟的第三方框架, 如 JavaParser 等. 此外, 本文参考了现有工作的代码实现, 如 JavaFuzzer、JavaTailor 等, 本文的开发人员也对编码实现进行了多次验证, 最大程度地保证了本文代码实现的正确性.

● 外部有效性分析. 影响本文实验结果结论的外部因素主要包含 3 个因素. 首先, 本文使用的种子程序是否具有代表性. 为了保障实验结果的有效性, 本文使用的种子程序为现有工作 JOpFuzzer 整理的历史揭错测试程序集, 该数据集覆盖了多种 Java 语法特征及 JVM 的不同模块. 同时, 本文进一步考虑了使用测试程序生成工具 JavaFuzzerr 生成的测试程序作为种子程序, 并对比其与历史揭错测试程序之间效果的差异. 其次, 本文的实验平台是否具有一致性. 为了减少实验平台对实验结果的影响, 本文的所有实验均在同一个实验平台上运行. 最后, 随机性是否会对本文的结论产生影响. 为了减少随机性带来的影响, 本文选择了 4 个长期支持的 OpenJDK 版本作为测试对象, 在每个版本上都进行相同的实验, 以此来确保实验结果在不同的实验版本上具有相似的结论, 减少随机性带来的影响.

● 对比有效性分析. 影响本文实验结果对比有效性的主要因素为实验所选择的评价指标是否合理. 为此, 本文采用了现有 JVM 测试工作中常用的评价指标来评估本文的有效性, 即不一致缺陷数量、代码覆盖以及未知缺陷检测的数量. 在不同组件与大模型的消融实验中, 本文针对性地设计了有效性、平均消耗词元以及平均生成长度等指标进行评估, 从而从多方面提升实验结论的有效性.

5.2 未来工作展望

本文是针对语言新特性的测试程序生成方法, 有效地弥补了现有测试方法在新特性测试方面的不足. 尽管本文在新特性缺陷检测上已取得了一定的效果, 但其仍存在一定的改进空间. 首先, 不同的大模型在代码生成、上下文理解能力上存在较大的差异. 因此, 在未来的研究中, 有必要对不同的大模型 (例如, 代码大模型) 的能力进行深入的对比分析, 并探索多模型融合以提升测试生成效果. 同时, 目前本文仅通过缩减代码生成规模缓解幻觉问题, 后续可结合反馈式生成 (feedback-guided generation) 或检索增强等策略进一步降低模型幻觉. 其次, 本工作参考现有 JVM 测试工作中常用的评价指标来评估不同组件消融后的效果, 而未对大模型总结与生成的准确性以及生成质量影响因素 (如插入点随机性所带来的影响) 进行系统的评估, 但细粒度的评估方法有助于进一步加深对方法机理与有效性的理解. 在未来工作将引入更细粒度的度量方法, 如基于文本相似度评估大模型对新特性的总结准确性, 基于代码表示评估生成片段的语义正确性, 以系统分析各组件的独立贡献, 从而使方法验证更为全面严谨. 最后, 本文实验仅在 Java 语言新特性上进行了验证. 然而, 本文所提出的方法具有通用性, 其依赖于大模型对语言特性的总结与代码生成, 并结合历史揭错程序中丰富的上下文来生成测试程序, 并不局限于 Java 特定语法. 因此, 本方法可以迁移至未来尚未发布的语言新特性, 乃至其他编程语言 (如 C#、Kotlin) 的新特性测试场景.

6 总 结

本文提出一种基于大语言模型的语言新特性测试程序生成方法 LumiX, 旨在生成能够覆盖新特性的测试程序. LumiX 首先利用大模型文本理解能力, 对 Java 的新特性描述进行解析, 从而总结出新特性的使用描述; 随后,

LumiX 利用大模型强大的代码生成能力, 通过结合历史揭错测试程序的上下文以及新特性的使用描述, 生成新的针对新特性的代码片段. 新生成的代码片段将被插入至历史揭错测试程序中, 从而生成新的测试程序. 新生成的测试程序采用两阶段的差分测试, 分别用 Java 编译器和 JVM 对其进行编译和执行, 从而全面地检测 Java 编译器和 JVM 中与新特性相关的缺陷. 本文在 4 个长期支持版本的 OpenJDK、两款主流的 Java 编译器以及两款主流的 JVM 上进行了广泛的实验. 实验结果表明, LumiX 在检测新特性缺陷方面优于现有工作, 且能够有效提升 Java 编译器和 Java 虚拟机的代码覆盖率. 此外, 在实验过程中, LumiX 累计检测出 16 个未知缺陷, 其中 12 个已经被开发人员确认或修复.

References

[1] OpenJDK. Java Release. 2025. https://openjdk.org/projects/jdk/24/

[2] Darcy JD. JEP 126: Lambda expression & virtual extension methods. 2015. https://openjdk.org/jeps/126

[3] Bateman A, Pressler R. JEP 437: Structured concurrency (second incubator). 2023. https://openjdk.org/jeps/437

[4] Chen JJ, Patra J, Pradel M, Xiong YF, Zhang HY, Hao D, Zhang L. A survey of compiler testing. ACM Computing Surveys (CSUR), 2021, 53(1): 4. [doi: 10.1145/3363562]

[5] Haghighat MR, Khukhro D, Yakovlev A. JavaFuzzer. 2018. https://github.com/AzulSystems/JavaFuzzer

[6] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 283–294. [doi: 10.1145/1993498.1993532]

[7] Livinskii V, Babokin D, Regehr J. Random testing for C and C++ compilers with YARPGen. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 196. [doi: 10.1145/3428264]

[8] Chen JJ, Wang GC, Hao D, Xiong YF, Zhang HY, Zhang L. History-guided configuration diversification for compiler test-program generation. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 305–316. [doi: 10.1109/ASE.2019.00037]

[9] Le V, Afshari M, Su ZD. Compiler validation via equivalence modulo inputs. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Edinburgh: ACM, 2014. 216–226. [doi: 10.1145/2594291.2594334]

[10] Le V, Sun CN, Su ZD. Finding deep compiler bugs via guided stochastic program mutation. In: Proc. of the 2015 ACM SIGPLAN Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. Pittsburgh: ACM, 2015. 386–399. [doi: 10.1145/2814270.2814319]

[11] Sun CN, Le V, Su ZD. Finding compiler bugs via live code mutation. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. Amsterdam: ACM, 2016. 849–863. [doi: 10.1145/2983990.2984038]

[12] Wu MY, Lu MH, Cui HM, Chen JJ, Zhang YQ, Zhang LM. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 56–68. [doi: 10.1109/ICSE48619.2023.00017]

[13] Xie ZF, Wen M, Qiu SY, Jin H. Validating JVM compilers via maximizing optimization interactions. In: Proc. of the 29th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Vol. 4. ACM, 2024. 345–360. [doi: 10.1145/3622781.3674188]

[14] Li C, Jiang YY, Xu C, Su ZD. Validating JIT compilers via compilation space exploration. In: Proc. of the 29th Symp. on Operating Systems Principles. Koblenz: ACM, 2023. 66–79. [doi: 10.1145/3600006.3613140]

[15] Chen YT, Su T, Sun CN, Su ZD, Zhao JJ. Coverage-directed differential testing of JVM implementations. In: Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Santa Barbara: ACM, 2016. 85–99. [doi: 10.1145/2908080.2908095]

[16] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot—A Java bytecode optimization framework. In: Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative Research. Mississauga: IBM Press, 1999. 13. [doi: 10.5555/781995.782008]

[17] Gu QH. LLM-based code generation method for golang compiler testing. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 2201–2203. [doi: 10.1145/3611643.3617850]

[18] Liu KB, Liu YY, Chen ZP, Zhang JM, Han YD, Ma Y, Li G, Huang G. LLM-powered test case generation for detecting tricky bugs. arXiv:2404.10304v1, 2024.

[19] Tu HX, Zhou ZD, Jiang H, Yusuf INB, Li YX, Jiang LX. LLM4CBI: Taming LLMs to generate effective test programs for compiler bug isolation. arXiv:2307.00593v1, 2023.

[20] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and

generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[21] Xia CS, Paltenghi M, Tian JL, Pradel M, Zhang LM. Fuzz4All: Universal fuzzing with large language models. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 126. [doi: 10.1145/3597503.3639121]

[22] Zhang BK, Zheng K. LumiX. 2025. https://github.com/zhang-bokai/LumiX

[23] Zhao YQ, Wang Z, Chen JJ, Liu MD, Wu MY, Zhang YQ, Zhang LM. History-driven test program synthesis for JVM testing. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1133–1144. [doi: 10.1145/3510003.3510059]

[24] Gao TC, Chen JJ, Zhao YQ, Zhang YQ, Zhang LM. Vectorizing program ingredients for better JVM testing. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 526–537. [doi: 10.1145/3597926.3598075]

[25] Zhao YQ, Chen JJ, Fu RF, Ye HJ, Wang Z. Testing the compiler for a new-born programming language: An industrial case study (experience paper). In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 551–563. [doi: 10.1145/3597926.3598077]

[26] Zang ZQ, Yu FY, Thimmaiah A, Shi A, Gligoric M. Java JIT testing with template extraction. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 1129–1151. [doi: 10.1145/3643777]

[27] Lidbury C, Lascu A, Chong N, Donaldson AF. Many-core compiler fuzzing. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 65–76. [doi: 10.1145/2737924.2737986]

[28] Lindholm T, Yellin F, Bracha G, Buckley A, Smith D. The Java® virtual machine specification. 2015. https://docs.oracle.com/javase/specs/jvms/se24/html/index.html

[29] Zhao YQ, Wang Z, Chen JJ, Fu RF, Lu YZ, Gao TC, Ye HJ. Program ingredients abstraction and instantiation for synthesis-based JVM testing. In: Proc. of the 2024 on ACM SIGSAC Conf. on Computer and Communications Security. Salt Lake City: ACM, 2024. 3943–3957. [doi: 10.1145/3658644.3690366]

[30] Ren XX, Ye XY, Lin Y, Xing ZC, Li SQ, Lyu MR. API-knowledge aware search-based software testing: Where, what, and how. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1320–1332. [doi: 10.1145/3611643.3616269]

[31] Lin Y, Ong YS, Sun J, Fraser G, Dong JS. Graph-based seed object synthesis for search-based unit testing. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 1068–1080. [doi: 10.1145/3468264.3468619]

[32] Chen YT, Su T, Su ZD. Deep differential testing of JVM implementations. In: Proc. of the 41st Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1257–1268. [doi: 10.1109/ICSE.2019.00127]

[33] Chaliasos S, Sotiropoulos T, Spinellis D, Gervais A, Livshits B, Mitropoulos D. Finding typing compiler bugs. In: Proc. of the 43rd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. San Diego: ACM, 2022. 183–198. [doi: 10.1145/3519939.3523427]

[34] Jia HX, Wen M, Xie ZF, Guo XC, Wu RX, Sun ML, Chen K, Jin H. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 43–55. [doi: 10.1109/ICSE48619.2023.00016]

[35] Liu K, Koyuncu A, Kim D, Bissyandé TF. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 31–42. [doi: 10.1145/3293882.3330577]

[36] Zhao YQ. Silent acceptance of non-final variable in unreachable lambda expression #3792. 2025. https://github.com/eclipse-jdt/eclipse.jdt.core/issues/3792

[37] Lu KM, Yu BW, Zhou C, Zhou JR. Large language models are superpositions of all characters: Attaining arbitrary role-play via self-alignment. In: Proc. of the 62nd Annual Meeting of the Association for Computational Linguistics. Bangkok: ACL, 2024. 7828–7840. [doi: 10.18653/v1/2024.acl-long.423]

[38] Wang ZJ, Zhou ZJ, Song D, Huang YH, Chen SM, Ma L, Zhang TY. Towards understanding the characteristics of code generation errors made by large language models. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering. Ottawa: IEEE, 2025. 2587–2599. [doi: 10.1109/ICSE55347.2025.00180]

[39] JaCoCo. JaCoCo Java code coverage library. 2025. https://www.eclemma.org/jacoco/

[40] GCov. 11 gcov—A test coverage program. 2025. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[41] LCov. 2025. https://github.com/linux-test-project/lcov

[42] JavaParser. Tools for your Java code. 2025. https://javaparser.org/

[43] Qwen Team. Qwen2.5 technical report. arXiv:2412.15115, 2024.

[44] Alibaba Cloud Bailian. Tongyi Qianwen-max. 2025 (in Chinese). https://bailian.console.aliyun.com/model-market?tab=model#/model-market/detail/qwen-max

[45] DeepSeek-AI. DeepSeek-V3 technical report. arXiv:2412.19437, 2025.

[46] Kang S, Yoon J, Askarbekkyzy N, Yoo S. Evaluating diverse large language models for automatic and general bug reproduction. IEEE Trans. on Software Engineering, 2024, 50(10): 2677–2694. [doi: 10.1109/TSE.2024.3450837]

[47] Li J, Li G, Li YM, Jin Z. Structured chain-of-thought prompting for code generation. ACM Trans. on Software Engineering and Methodology, 2015, 34(2): 37. [doi: 10.1145/3690635]

[48] Yang G, Zhou Y, Chen X, Zhang XY, Zhuo TY, Chen TL. Chain-of-thought in neural code generation: From and for lightweight language models. IEEE Trans. on Software Engineering, 2024, 50(9): 2437–2457. [doi: 10.1109/TSE.2024.3440503]

[49] Zhang JW, Hu X, Xia X, Cheung SC, Li SP. Automated unit test generation via chain of thought prompt and reinforcement learning from coverage feedback. ACM Trans. on Software Engineering and Methodology, 2025, 35(4): 1–30. [doi: 10.1145/3745765]

[50] Deng YL, Xia CS, Yang CY, Zhang SD, Yang SJ, Zhang LM. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 70. [doi: 10.1145/3597503.3623343]

[51] Wang Z, Zhao YQ, Liu S, Sun J, Chen X, Lin HR. MAP-coverage: A novel coverage criterion for testing thread-safe classes. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 722–734. [doi: 10.1109/ASE.2019.00073]

[52] Fuchs D. JEP 264: Platform logging API and service. 2014. https://openjdk.org/jeps/264

附中文参考文献

[44] 阿里云百炼. 通义千问-Max. 2025. https://bailian.console.aliyun.com/model-market?tab=model#/model-market/detail/qwen-max

作者简介

赵英全, 博士, 副研究员, CCF 专业会员, 主要研究领域为软件测试, 软件分析.

张博凯, 硕士生, 主要研究领域为软件测试.

王赞, 博士, 教授, CCF 专业会员, 主要研究领域为软件测试, 机器学习.

郭以勒, 硕士生, 主要研究领域为编译器测试.

陈佳丽, 副教授, 主要研究领域为智能软件工程.

陈翔, 博士, 副教授, CCF 高级会员, 主要研究领域为智能软件工程, 软件仓库挖掘, 经验软件工程.

陈俊洁, 博士, 教授, CCF 高级会员, 主要研究领域为软件分析与测试.