计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241200059

基于语义变化的缺陷生成与缺陷预测模型测试

郭力玮[1]　吴永豪[2]　刘　勇[1]
1 北京化工大学信息科学与技术学院　北京 100029
2 北京石油化工学院信息工程学院　北京 102627
(liwei.glw@outlook.com)

摘　要　近年来,机器学习技术在软件开发中的缺陷预测领域取得了显著进展,能够在大规模代码库中自动检测错误。这些进展有望提升软件的可靠性、安全性和整体质量。缺陷预测模型可以自动化检测代码中是否包含错误。然而,现有的缺陷预测模型虽然具有一定优势,但往往无法准确识别那些标记为无问题的有缺陷代码。目前缺乏对缺陷检测模型质量的系统性的实证研究,现有方法 DPTester 通过生成缺陷代码来检测缺陷模型的能力,该方法通过修改代码中的 if 条件来产生缺陷代码。然而,现有方法自动生成的缺陷代码过于简单,评估场景也未包括最新大语言模型在内的广泛模型。基于此,提出了改进方法 DefectGen,通过引入多种策略来生成更符合现实问题的缺陷代码,并且评估的缺陷模型包含了大语言模型。实验结果表明,DefectGen 在生成复杂缺陷代码的能力上较之前的方法有显著提升,能够在单个正确代码上生成 1.2 倍的缺陷代码。在测试 CodeT5+,CodeBERT 和 GPT-4o 模型时,发现缺陷预测有误的数量占比分别为 62%,78% 和 30%。与此同时,DefectGen 在测试输入生成和缺陷检测阶段展现出更高的效率,每条测试输入的生成时间和检测时间分别为 0.003 s 和 0.02 s。这些结果表明,DefectGen 不仅有效揭示了现有模型的局限性,还为改进缺陷预测模型和提升软件质量保障流程提供了新可能。

关键词:缺陷预测;机器学习;大语言模型
中图分类号　TP311.53

Semantic Variations Based Defect Generation and Prediction Model Testing

GUO Liwei[1],WU Yonghao[2] and LIU Yong[1]
1 College of Information Science and Technology,Beijing University of Chemical Technology,Beijing 100029,China
2 School of Information Engineering,Beijing Institute of Petrochemical Technology,Beijing 102627,China

Abstract　In recent years,machine learning techniques have made significant advancements in defect prediction within software development,enabling the automatic detection of errors in large-scale codebases. These advancements are expected to enhance the reliability,security,and overall quality of software. Defect prediction models can autonomously identify whether code contains errors. However,existing models,while having certain advantages,also exhibit limitations. They often fail to accurately identify vulnerabilities or incorrectly label defective code segments as problem-free. Currently,there is a lack of systematic empirical studies on the quality of defect detection models. The existing method,DPTester,assesses the effectiveness of defect models by generating defective code through modifications to if conditions in the code. However,the defect code produced by this method is overly simplistic,and the evaluation scenarios do not cover a wide range of models,including the latest large language models. To address this gap,this paper proposes an improved method called DefectGen. This new approach introduces multiple strategies to generate defect code that more closely reflects real-world issues. Furthermore,the evaluation of defect models includes large language models. Experimental results indicate that DefectGen significantly enhances the ability to generate complex defect code compared to previous methods,producing 1.2 times more defective code from a single correct code instance. When testing the CodeT5+,CodeBERT,and GPT-4o models,the proportions of incorrect defect predictions were found to be 62%,78%,and 30%. Additionally,DefectGen demonstrates higher efficiency in both test input generation and defect detection phases,with generation and detection times of 0.003 seconds and 0.02 seconds per test input. These results suggest that DefectGen not only effectively exposes the limitations of existing models but also provides new opportunities for improving defect prediction models and enhancing software quality assurance processes.

Keywords　Defect prediction,Machine learning,Large language models

基金项目:国家自然科学基金(61902015,61872026,61672085)
This work was supported by the National Natural Science Foundation of China(61902015,61872026,61672085).
通信作者:刘勇(lyong@mail.buct.edu.cn)

1 引言

随着信息技术的发展,软件已成为日常生活中不可或缺的一部分。在软件开发和维护中,及时检测和修复软件缺陷至关重要。近年来,基于机器学习的缺陷预测技术引起了广泛关注。这种技术极大地改变了软件质量保证领域,能够自动识别代码库中的潜在问题[1-4],显著提升了软件的可靠性与安全性。然而,缺陷预测模型的质量参差不齐,可能会出现预测错误,导致实际无缺陷的代码被标记为有缺陷,或者将正确的代码错误地标记为有问题[5]。

为了评估模型的能力,通常使用测试用例来验证模型的精度[6]。因此,需要根据现有代码生成大量测试用例。然而,现有方法多关注保持代码语义的一致性,例如通过重命名变量或标识符来生成测试用例。尽管这种方法在一定程度上维持了输入代码的结构,但它忽视了在真实开发场景中,由于业务需求变化或开发实践演进,代码语义可能发生变化的情况。为了应对这些不足,Xu 等提出了 DPTester 方法[7]。该方法通过主动在代码中引入缺陷,模拟软件开发中可能遇到的错误,以此测试缺陷预测模型的鲁棒性和适应性。DPTester 的核心思想是通过修改条件语句(如将 if 语句改为始终为 True 或 False)来生成潜在的缺陷,并利用这些缺陷测试现有的缺陷预测模型。然而,尽管 DPTester 为缺陷测试提供了一种有效框架,但其缺陷生成方式较为简单,主要集中于条件语句的修改,缺乏对更复杂缺陷场景的模拟。

为此,本文在 DPTester 的基础上进行了重要改进。首先,引入了更多种类的缺陷生成策略,如修改条件逻辑、破坏循环结构以及随机修改变量数值等,以增加缺陷的多样性和复杂性。同时,本文还引入了大语言模型的缺陷预测功能,以验证本文方法能否有效检测最新大语言模型中的预测缺陷。通过这些创新,本文提出的 DefectGen 方法能够更全面地模拟实际开发中可能出现的复杂问题,并对现有的缺陷预测模型进行更为严苛的测试。

DefectGen 采用两步框架:缺陷测试输入生成和缺陷预测模型评估。

在缺陷测试输入生成阶段,DefectGen 系统性地修改源代码中的关键部分,包括条件语句、循环结构和数值表达式。通过调整条件逻辑、改变循环结构和修改数值等方法,引入潜在缺陷,从而创建更加复杂和多样化的测试输入集。这些输入模拟了各种在实际开发环境中可能出现的复杂缺陷场景,为模型评估提供了丰富的测试条件。

随后,在缺陷预测模型评估阶段,DefectGen 将生成的测试输入应用于缺陷预测模型,以检测模型识别人为引入错误代码的能力。如果模型未能准确识别这些缺陷,DefectGen 会将其标记为模型的潜在问题,并提供反馈以促进进一步优化。

实验基于现有的缺陷预测数据集[8],设计了 3 项研究问题(RQs),用于评估 DefectGen 方法的有效性和效率。实验结果表明,DefectGen 能够高效生成测试输入,总共生成了 265 901 个有效测试用例,平均每段代码生成 14.51 个测试输入,且所有的测试输入均为有效输入(RQ1)。实验还揭示了在预测过程中,CodeT5+,CodeBERT 和 GPT-4o 存在大量预测有误的情况。在所有的缺陷测试输入中,它们的错误数量占比分别为 62%、78% 和 30%(RQ2)。此外,DefectGen 在执行效率上也表现优异,生成每个测试输入的平均时间仅为 0.003 s,而每次问题检测的平均耗时为 0.02 秒(RQ3)。

综上所述,与 DPTester 相比本文的主要贡献包括:

1)本文提出了 DefectGen 方法,该方法通过增加缺陷的多样性和复杂性,能够更全面地模拟实际开发中可能出现的复杂缺陷情景。此外,DefectGen 方法在单个代码片段上生成的缺陷代码数量是 DPTester 方法的 1.2 倍。

2)与 DPTester 方法相比,DefectGen 在模型检测上表现更佳。实验结果表明,在测试 CodeT5+,CodeBERT 和 GPT-4o 模型时,分别发现 62%,78% 和 30% 的错误预测数量,平均比 DPTester 方法多检测出 13% 的错误。

3)本文测试了大语言模型 GPT-4o 的缺陷检测能力。实验结果显示,DefectGen 方法能够检测出大语言模型在代码缺陷预测方面的不足,其中有 20% 的缺陷预测是错误的。

2 相关工作

本章介绍了针对深度代码模型的自动化测试方法。Yefet 等[9]提出了一种基于梯度计算的变量修改方法,该方法仅适用于使用单热编码(one-hot encoding)处理代码的模型。然而,该方法不兼容于采用高级编码技术(如 CodeBERT[10],CodeT5[11]和 CodeT5+[12])的模型。

Zhang 等[13]提出了使用标识符重命名转换的迭代方法进行测试。此外,Pour 等[14]探索了一种基于搜索的策略,结合了迭代重构特性。Henkel 等[15]使用基于梯度的优化生成对抗样本,其中包括重命名和插入无用代码等策略。同样地,Jha 等[16]提出结合语言模型和贪婪搜索机制的方案,用于识别易受攻击的替换令牌。

这些方法的一个共同限制在于,它们往往未能保持生成样本的自然性。为了解决这一问题,Tian 等[17]提出了基于代码差异的技术,通过给定代码片段和包含小范围代码修改但产生不同预测结果的输入来引导生成。与此类似,Li 等[18]提出了一个针对代码补全的测试方法。

然而,这些方法通常不适用于缺陷预测模型的测试,或仅关注于标签保持的测试策略。为此,本文引入了一种面向缺陷预测模型的缺陷引入方法,侧重于涉及标签更改的测试的替代性。

3 本文方法

图 1 给出了 DefectGen 的整体框架。DefectGen 的核心思想是通过引入复杂的语义缺陷来系统性地生成测试输入,从而全面评估缺陷预测模型在复杂语义变化下的性能。与传统方法不同,DefectGen 不仅修改条件语句,还通过多种缺陷生成策略(修改条件逻辑、破坏循环结构以及随机修改变量数值等)来增加缺陷的多样性和复杂性。DefectGen 的流程主要包括两个步骤:缺陷测试输入生成以及缺陷预测模型评估。

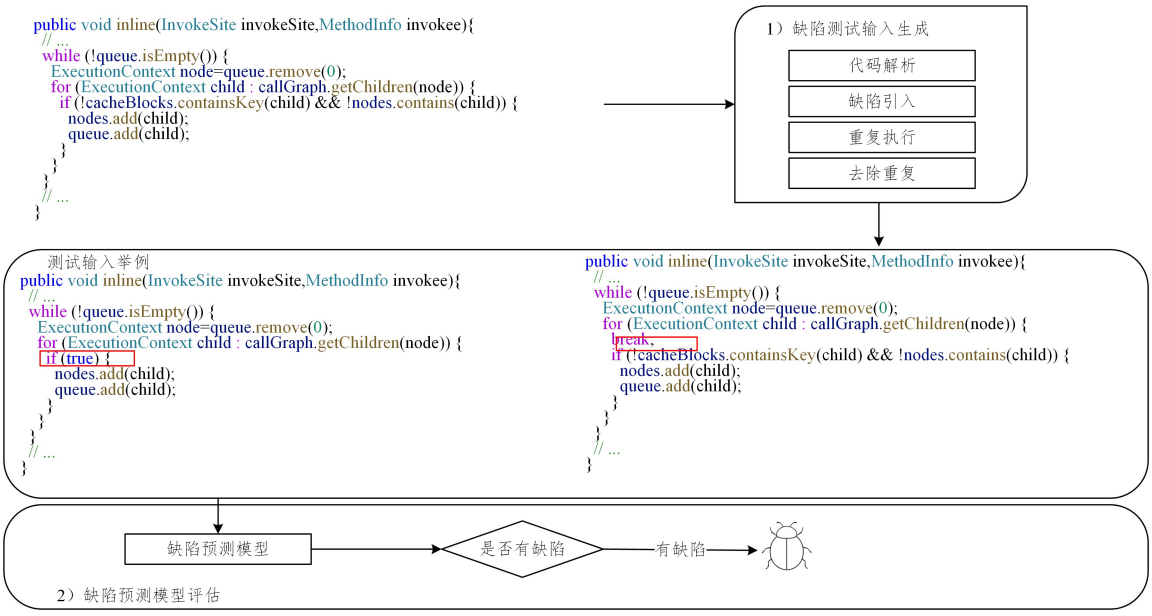

图 1　DefectGen 整体框架

Fig. 1　DefectGen framework

3.1　缺陷测试输入生成

在生成缺陷测试输入的过程中,生成多样化和复杂的测试输入至关重要。DefectGen 通过以下步骤生成测试输入,以有效模拟软件代码中的潜在复杂缺陷。为了生成这些测试输入,DefectGen 采用了多种策略,如修改条件语句、循环结构和变量依赖等关键代码元素。

3.1.1　代码解析

DefectGen 首先利用 Tree-sitter 工具将源代码解析为抽象语法树(AST)。AST 以层次化的方式表示代码结构,能够有效地定位代码中的特定元素,如条件语句、循环语句和变量定义等。Tree-sitter 提供了强大的代码解析能力,能够遍历整个 AST,逐一识别代码中的每个目标元素,并获取相关的上下文信息,如变量类型和作用域。

3.1.2　缺陷引入

在定位到关键代码元素后,DefectGen 通过多种策略对其进行修改,从而引入潜在的缺陷。具体而言,DefectGen 随机选择以下几种策略来引入缺陷。

1)条件逻辑修改:DefectGen 通过对条件语句的操作符进行反向修改,来增加条件表达式的复杂性。例如,将大于改为小于,或将小于改为大于,或将等于改为不等于,以此来模拟不同类型的逻辑错误。如果条件表达式中没有明显的比较操作符,则随机选择 true 或 false,以增加代码的非确定性。这样修改后的条件逻辑会引入确定的缺陷,作为后续生成缺陷代码的步骤之一。

2)循环结构修改:在循环结构中,DefectGen 通过随机插入 break 语句来模拟潜在的循环缺陷。例如,在循环体中插入 break 语句,可能会导致程序提前退出或逻辑错误。此操作有助于测试循环结构在特定条件下的行为。

3)数值变量修改:如果 AST 节点是数值类型(如整数或浮点数),DefectGen 会随机对其进行加法操作,模拟计算错误或数据偏差。例如,在一个数值节点上随机增加一个整数,以改变原有数值,从而引入潜在的逻辑问题。

这些修改直接在 AST 上进行,并且随机植入错误,保证至少会植入一种错误,之后将修改后的 AST 转换回源代码,确保修改后的代码在语法上仍然正确且结构完整。

3.1.3　重复执行

上述操作会重复执行多次,每次修改不同的代码元素,以生成多个包含不同缺陷类型的代码版本。实验中,DefectGen 方法和 DPTester 方法一样,通过缺陷引入的方法产生多个不同的变异代码,最多生成 30 个不同版本的程序,用于模拟多样化的缺陷场景。

3.1.4　去除重复

在生成多版本代码后,DefectGen 通过比较新生成的代码,去除那些虽然语义等效但功能重复的测试输入,以确保测试输入集的唯一性和高效性。这一过程有助于优化测试输入的质量,避免冗余测试,从而提高后续测试阶段的效率。

4　实验设置

本章介绍了评估 DefectGen 在通过修改条件语句生成测试输入和检测问题方面性能的实验框架。

4.1　研究问题

RQ1:DefectGen 在生成用于缺陷预测的有效测试输入方面的表现如何?

此问题旨在评估 DefectGen 生成的测试输入是否能够成功模拟可能导致缺陷暴露的场景。部分测试输入将接受人工检查,以验证其在缺陷预测中的有效性和相关性。在确认 DefectGen 能够生成相关测试输入后,下一步将分析其在识别缺陷问题上的具体表现。

RQ2:DefectGen 在检测缺陷预测模型中的问题时表现如何?

针对 RQ2,DefectGen 通过缺陷预测模型分析 RQ1 中生成的测试输入,并测量其缺陷检测能力。测试输入的准确性也将在实验中进行评估。在评估了 DefectGen 的缺陷检测效果后,将进一步探讨其效率表现。

RQ3:DefectGen 在进行缺陷预测模型测试时的效率表现如何?

针对 RQ3，将通过测量 DefectGen 生成测试输入所需的时间，并分析这些输入定位代码缺陷的时间，来评估其效率。本次评估旨在确定 DefectGen 在实际应用中的可行性，重点考虑缺陷检测的全面性与测试过程中资源消耗之间的平衡。

4.2 评价指标

为评估 DefectGen 在这些研究问题中的性能，采用准确率作为评估指标。缺陷检测准确率定义为正确识别的缺陷数量与实际缺陷总数的比例。该指标用于反映 DefectGen 在测试输入中识别真实缺陷的效果。

为了全面评估缺陷预测模型在复杂语义变化下的性能，DefectGen 设计了一套细化的测试结果生成框架。通过该框架，DefectGen 能够系统性地验证模型是否准确检测到由各种语义变化所引入的缺陷。

DefectGen 的核心目标是通过修改条件语句、循环结构和变量数值等，来评估缺陷预测模型对复杂语义变化的敏感性。具体过程如下。

1）缺陷代码输入：将所有生成的测试输入提供给缺陷预测模型进行评估。假设通过改变代码的关键逻辑部分（如修改条件逻辑、破坏循环结构以及随机修改变量数值等），可以引入潜在的语义缺陷。

2）结果分析：如果缺陷预测模型未能检测到测试输入中的缺陷（即预测结果为“无缺陷”），DefectGen 会将此问题记录为模型未能识别缺陷的实例。

3）综合评估：DefectGen 将对所有测试输入进行评估，全面记录模型在各种语义修改场景下的表现，从而揭示模型的局限性和改进空间。

4.3 被测缺陷预测模型

本研究聚焦于评估 3 种前沿模型的缺陷检测能力：CodeBERT[19]，CodeT5+[12] 和 GPT-4o。这些模型因其在理解和分析代码方面的卓越性能而备受关注，特别是 GPT-4o 作为最新的大语言模型，其强大的自然语言处理能力使其成为跨多种编程语言的理想选择。

1）由 Feng 等提出的 CodeBERT 是一种双模态模型，能够处理自然语言文本和源代码。其基于 Transformer 的架构使其能够深入理解代码的语义细节，从而检测需要深度理解代码功能和意图的复杂缺陷。CodeBERT 还具备结合代码与相关注释进行分析的能力，能够全面检查潜在的不一致性和缺陷。在缺陷预测任务中，输入一段代码后，CodeBERT 会判断该代码片段中是否存在缺陷。

2）由 Wang 等提出的 CodeT5+ 是对基于 Transformer 的 T5 模型的扩展，专门针对代码领域进行优化。通过在多种编程语言的多样化数据集上进行预训练，CodeT5+ 能够掌握广泛的编程结构和习惯表达。在缺陷检测任务中，CodeT5+ 凭借对代码语义和结构的深刻理解，成为跨不同代码库识别缺陷的有力工具。输入一段代码后，CodeT5+ 会判断该代码片段中是否存在缺陷。

3）由 OpenAI 提出的 GPT-4o 是基于最新的 GPT-4 模型开发的多模态大语言模型，具备强大的自然语言理解和生成能力。在缺陷预测任务中，GPT-4o 能够理解并分析源代码的上下文信息，并结合大规模的预训练数据，评估代码中的潜在问题和缺陷。通过对代码片段的深度分析，GPT-4o 能有效检测到复杂的语义错误和潜在的逻辑缺陷。输入代码后，GPT-4o 会对代码进行语义分析并预测可能的缺陷。

4.4 数据集

在缺陷检测实验中，使用了 Recoder 数据集[8]。该数据集包含大量实例，每个实例由一个具体的软件缺陷及其对应的修复代码组成，均来自基于 Java 的项目。这些数据精心收集自多个真实的开源软件项目，涵盖了多样化的测试场景，为评估缺陷预测模型提供了丰富且全面的实验数据。

Recoder 数据集的多样性和复杂性使其成为评估 DPTester 性能的理想选择。该数据集涵盖了从简单语法错误到复杂逻辑问题的广泛缺陷类型，为模型提供了严格的测试环境。这种设计不仅能评估模型的缺陷预测准确性，还可以衡量其处理不同复杂度缺陷的能力。因此，Recoder 数据集为分析模型的可扩展性和适应性提供了全面的实验基础，有助于深入理解模型在各种缺陷场景下的表现。

该数据集中包含 297 039 对代码片段，每对由一段缺陷代码（修复前）和对应的修复代码（修复后）组成。这种结构为研究缺陷预测和修复方法提供了清晰的对比依据。为进一步优化数据集的实验实用性，每对代码片段被拆分为两个独立的实例，并标注为“存在缺陷”或“修复完成”。经过此过程，数据集总共生成了 594 078 条标注代码片段。

为确保实验结果的可靠性和科学性，这些代码片段被划分为训练集、验证集和测试集，比例为 8∶1∶1。其中，训练集包含 475 232 条代码片段，验证集和测试集分别包含 59 404 条和 59 442 条代码片段。这种划分策略为模型的全面训练、性能验证和泛化能力测试提供了充足的数据支持。

4.5 参数设置

实验在 Ubuntu 16.04 操作系统上进行，设备配置包括 256 GB 内存和 4 个 Intel E5-2620 v4 CPU。实验中使用的神经网络模型，包括 CodeBERT 和 CodeT5+，均在单张 Nvidia Titan RTX 显卡上进行训练和推理。

CodeBERT 和 CodeT5+ 均在训练数据集上进行了微调。在训练过程中，通过早停机制，根据验证集的表现选择性能最佳的模型。

GPT-4o 模型通过 OpenAI 官方提供的 API 接口进行调用，确保模型返回的结果是符合要求的 JSON 格式。这种格式化的返回方式有助于保证模型输出的数据结构化，便于后续的解析与处理。特别是在进行代码分析和缺陷检测任务时，结构化的 JSON 输出不仅提升了结果的可读性，还增强了自动化处理流程的效率。

在 API 请求中，采用了如下提示词（Prompt）来指导 GPT-4o 输出符合预期结构的结果：你是一个可以进行代码分析并检测 bug 的工具。返回的结果应为 JSON 格式，包括一个‘result’字段，值为 0 或 1，分别表示无 bug 或有 bug。在此提示词中，system 角色的内容指引模型进行代码分析并返回结构化的 JSON 格式结果。通过这种方式，可以确保模型返回合法的 JSON 对象，进而有效地进行后续的结果解析和利用。

5 实验结果

5.1 RQ1：DefectGen 在生成用于缺陷预测的有效测试输入方面的表现如何？

为回答 RQ1，分析了测试输入生成方法在缺陷预测模型

中的有效性。在实验中,测试数据集中每条测试数据被输入到 DefectGen,并对条件语句进行策略性修改。在此问题的分析中,评估了生成测试输入的数量和质量,实验结果如表 1 所列。

表 1 生成测试输入的数量

Table 1 Number of generated inputs

	代码片段	测试输入	每个代码片段	准确率/%
DefectGen	18327	**265901**	**14.51**	**100**
DPTester	18327	222112	12.12	99

5.1.1 生成测试数量

对于每条测试数据,最多生成 30 个变异体。在 59442 条测试数据中,共有 18327 条代码片段包含缺陷并被标记为无缺陷。基于这些代码片段,DefectGen 成功生成了 265901 个测试输入,用于后续实验,相当于每条代码片段平均生成 14.51 个测试输入,在相同的条件下生成的测试数量是 DPTester 的 1.2 倍。

5.1.2 生成测试质量

为评估所生成测试输入的有效性,从生成的测试集随机选取了 100 个变异体代码片段,并邀请 3 位具有相关编程与软件测试经验的专家进行独立人工评估。评估重点集中在以下两个方面:1)生成的代码是否保持了基本的语法正确性和可编译性;2)生成的代码中所引入的缺陷是否一定是错误的。结果显示,这 100 个变异体均被评估人员一致认定为适用于缺陷预测模型的测试输入,在此条件下实现了 100%的评估通过率。与已有的 DPTester 方法相比,我们并未观察到仅通过插入 if 语句进行调试而无法真正引入有效缺陷的情况,这说明本文方法在缺陷有效性与输入质量上具有更高的准确度和适用性。

实验结果表明,测试输入生成方法具有较高的可靠性。结果进一步验证了该方法在生成高质量测试用例以评估缺陷预测模型准确性方面的有效性。

基于这些结果,共生成了 265901 个测试输入(其来源于 18327 条原始代码片段),并将其用于进一步研究缺陷预测模型在准确检测引入缺陷方面的能力。

RQ1 总结:DefectGen 成功生成了 265901 个测试输入,用于进一步实验。平均而言,每条代码片段生成了 14.51 个测试输入,这一数字是 DPTester 的 1.2 倍。在生成测试输入方面,DefectGen 的准确率接近 100%,这验证了其在生成高质量测试用例以支持缺陷预测模型测试中的有效性。

5.2 RQ2:DefectGen 在检测缺陷预测模型问题方面的效果如何?

为回答 RQ2,本文使用了由 RQ1 生成的测试输入。这些测试输入是通过系统性地注入缺陷生成的,总计 265901 条。该数据集用于评估缺陷预测模型在缺陷检测方面的能力。

5.2.1 发现问题的数量

表 2 列出了 DefectGen 在 CodeT5+,CodeBERT 和 GPT-4o 中检测到的问题数量。实验结果表明,DefectGen 在 CodeT5+ 上检测到 163762 个问题,在 CodeBERT 上检测到 207025 个问题,而在 GPT-4o 上检测到 79238 个问题。这一结果揭示了现有模型在缺陷预测中的问题。相较于 DPTester 方法,本文方法能够检测出更多问题,同时展现出更高的可靠性。

表 2 检测到的问题数量

Table 2 Number of issues detected

	模型	预测有误	百分比/%
DefectGen	CodeT5+	**163762**	**61.57**
	CodeBERT	**207025**	**77.89**
	GPT-4o	**79238**	**29.80**
DPTester	CodeT5+	126122	56.78
	CodeBERT	156126	70.31
	GPT-4o	51536	23.20

通过进一步分析发现,CodeBERT 发现的问题数量(207025)高于 CodeT5+(163762),而 GPT-4o 的报告问题数量(79238)则介于两者之间。这表明在处理修改引入的缺陷时,模型的灵敏度可能存在差异。这一差异表明了模型在识别特定类型缺陷或处理语法变更时的能力差异。

5.2.2 行为不一致预测

DefectGen 假设在引入缺陷后,代码片段应被分类为有缺陷。为验证此假设,我们统计了模型预测与该假设不一致的情况,即测试输入与其原始代码被预测为不同状态的次数。表 3 列出了行为不一致预测的数量,其中 CodeT5+,CodeBERT 和 GPT-4o 分别为 132758,193934 和 62220。这表明模型在检测因代码修改引入的缺陷时存在显著困难。通过对比 DPTester 的方法,发现检测到的行为不一致数量有所提升。DPTester 原论文中未进行 GPT-4o 的实验。本研究复现了 DPTester 的方法,并进行了相同的实验。

表 3 缺陷预测模型的不一致预测数量

Table 3 Inconsistent predictions of defect prediction models

	模型	不一致预测数量
DefectGen	CodeT5+	**132758**
	CodeBERT	**193934**
	GPT-4o	**62220**
DPTester	CodeT5+	126122
	CodeBERT	156126
	GPT-4o	53341

CodeBERT 的行为不一致预测数量高达 193934,这表明模型在预测某些缺陷类型上虽然表现良好,但在识别更复杂的缺陷注入或隐藏缺陷时仍存在明显不足。这反映了当前缺陷预测模型在检测代码修改引入的缺陷方面的局限性。

RQ2 总结:评估结果显示,DefectGen 对缺陷预测模型的精准度构成了挑战。在检测因语句修改而引入的缺陷方面,CodeBERT 检测到 207025 个问题,CodeT5+ 检测到 163762 个问题,GPT-4o 检测到 79238 个问题。行为不一致预测数量的增加(CodeBERT 为 193934,CodeT5+ 为 132758,GPT-4o 为 62220)进一步突显了现有模型在处理代码修改引入缺陷时的局限性。研究结果表明,与 DPTester 相比,本文方法在缺陷模型检测能力上表现更优。

5.3 RQ3:DefectGen 在测试缺陷预测模型时的效率如何?

为回答 RQ3,评估了 DefectGen 在测试输入生成和问题检测阶段的效率。效率在工具集成到现有开发工作流中至关重要,特别是在自动化测试和持续集成环境中,时间约束是一个重要考量因素。表 4 提供了这些流程所需时间的详细分解。

表 4 DefectGen 的执行效率

Table 4 Efficiency of DefectGen

(s)

DefectGen	整体执行时间	每个测试执行时间
测试输入生成	947.840	0.003
缺陷预测	5573.738	0.02

5.3.1 测试生成效率

DefectGen 生成测试输入的总时间为 947.840 s。按每条测试输入计算，平均时间为 0.003 s。这表明 DefectGen 在测试输入生成方面具有较高的效率，能够快速生成大量测试用例，为进一步评估提供支持。

5.3.2 问题检测效率

在问题检测阶段，DefectGen 的总耗时为 5573.738 s，平均而言，每次检测需要 0.02 s。尽管与测试输入生成阶段相比，该阶段的总时间更长，但其单次检测的时间效率仍保持在较低水平。这种效率反映了 DefectGen 不仅能够及时发现潜在缺陷，还能够以快速的方式完成，这对于将该工具集成到持续集成/持续部署(CI/CD)管道中至关重要。

此外，通过并行处理可以显著提高 DefectGen 的效率。利用并发执行技术，同时进行测试输入生成和问题检测，可以进一步缩短总耗时。并行化方法能够利用现代多核处理器的能力，同时生成和评估多个测试用例，从而显著缩短执行时间。这种优化策略在云计算环境或需要多线程处理的大规模软件开发项目中尤为重要，为满足大规模软件开发需求提供了可扩展的解决方案。

RQ3 总结：DefectGen 的效率分析表明，其能够高速生成测试输入和检测问题，每条测试输入的平均生成时间为 0.003 s，每次问题检测的平均时间为 0.02 s。这些指标突出了 DefectGen 在自动化测试环境中无缝集成的潜力，尤其是在时间效率至关重要的情况下。尽管问题检测阶段耗时较长，但总体效率支持了 DefectGen 在增强缺陷预测和软件质量保障流程中的实际应用价值。

6 有效性威胁分析

6.1 外部有效性威胁

外部有效性威胁可能存在于本文使用的模型和数据集之中。本研究聚焦于基于 Java 的代码片段，并使用 CodeBERT 和 CodeT5+ 模型。此选择可能会限制研究结果在其他编程语言和缺陷预测模型中的适用性。

为缓解这一威胁，本文选择了性能最优的模型，并从 Recoder 数据集中精心筛选了多样化且具有代表性的代码片段，以涵盖广泛的真实缺陷场景。同时，鼓励未来研究探索更多编程语言和模型，以验证并扩展本文的研究结果。

6.2 内部有效性威胁

内部有效性威胁主要可能出现在 DefectGen 的实现与执行，以及数据的处理与分析过程中。其中一个潜在的风险来源于代码实现中的错误，这些错误可能会影响缺陷预测模型的结果。

为缓解这一威胁，本文采用了两项关键策略。

1)使用作者提供的模型

为确保缺陷预测分析的完整性，直接使用了 CodeBERT 和 CodeT5+ 作者在 Hugging Face 平台[HuggingFace (2024)]上提供的预训练模型。此方法保证了模型的使用与其预期用途和优化配置一致，从而降低了因实现相关的不准确性对实验结果产生影响的可能性。此外，本文还使用了 OpenAI 提供的 GPT-4o 模型，作为最新的大语言模型之一，进一步增强了对复杂缺陷的预测能力，确保了实验在更广泛模型的评估下具有较强的代表性和全面性。

2)遵循成熟的解析工具

在测试输入的生成和操作中，本文依赖于 Tree-sitter，这是一种健壮且广泛使用的解析器，已在 GitHub 上发布。Tree-sitter 将代码解析为抽象语法树(AST)方面的可靠性和高效性，确保了本文在修改条件语句和生成测试输入方面的操作既准确又符合软件分析领域的最佳实践。

结束语 本文在 DPTester 方法的基础上进行了重要改进，提出了 DefectGen 方法。通过引入多种缺陷生成策略，如改变条件逻辑、破坏循环结构和随机修改数值等，DefectGen 增加了缺陷的多样性和复杂性。同时，评估了最新的大语言模型 GPT-4o，以测试其在缺陷预测方面的性能和局限性。实验结果表明，DefectGen 能更全面地模拟实际开发中的复杂缺陷，并为现有缺陷预测模型提供了更加严苛的测试场景。

参 考 文 献

[1] CHEN J, HU K, YU Y, et al. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction [C]// 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). 2020: 578-589.

[2] WANG S, LIU T, TAN L. Automatically learning semantic features for defect prediction[C]// Proceedings of the 38th International Conference on Software Engineering. New York: Association for Computing Machinery, 2016: 297-308.

[3] LIANG H, YU Y, JIANG L, et al. Seml: A Semantic LSTM Model for Software Defect Prediction[J]. IEEE Access, 2019, 7: 83812-83824.

[4] GIRAY G, BENNIN K E, KÖKSAL Ö, et al. On the use of deep learning in software defect prediction[J]. Journal of Systems and Software, 2023, 195: 111537.

[5] FENTON N E, NEIL M. A critique of software defect prediction models[J]. IEEE Transactions on Software Engineering, 1999, 25(5): 675-689.

[6] CARSON J S. Model verification and validation [C] // Proceedings of the Winter Simulation Conference: 2002: 52-58.

[7] XU F, SUN Z. Defect-Introducing Defect Prediction Testing [C]// 2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C). 2024: 401-410.

[8] ZHU Q, SUN Z, XIAO Y, et al. A syntax-guided edit decoder for neural program repair[C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: Association for Computing Machinery, 2021: 341-353.

[9] YEFET N, ALON U, YAHAV E. Adversarial examples for models of code[J]. Proc. ACM Program. Lang., 2020, 4(OOPSLA): 162: 1-162: 30.

[10] FENG Z, GUO D, TANG D, et al. CodeBERT: A Pre-Trained

Model for Programming and Natural Languages[C]// Findings of the Association for Computational Linguistics (EMNLP 2020). Association for Computational Linguistics, 2020: 1536-1547.

[11] WANG Y, WANG W, JOTY S, et al. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation[C]// MOENS M F, HUANG X, SPECIA L, et al. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021: 8696-8708.

[12] WANG Y, LE H, GOTMARE A, et al. CodeT5+: Open Code Large Language Models for Code Understanding and Generation [C]// Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Singapore: Association for Computational Linguistics, 2023: 1069-1088.

[13] ZHANG H, LI Z, LI G, et al. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models [C]// Proceedings of the AAAI Conference on Artificial Intelligence. 2020: 1169-1176.

[14] POUR M V, LI Z, MA L, et al. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding [C]// 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). 2021: 36-46.

[15] HENKEL J, RAMAKRISHNAN G, WANG Z, et al. Semantic Robustness of Models of Source Code[C]// 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2022: 526-537.

[16] JHA A, REDDY C K. CodeAttack: Code-Based Adversarial Attacks for Pre-trained Programming Language Models[C]// Proceedings of the AAAI Conference on Artificial Intelligence. 2023: 14892-14900.

[17] TIAN Z, CHEN J, JIN Z. Code Difference Guided Adversarial Example Generation for Deep Code Models [C]// 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023: 850-862.

[18] LI Z, WANG C, LIU Z, et al. CCTEST: Testing and Repairing Code Completion Systems[C]// 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 2023: 1238-1250.

[19] FENG Z, GUO D, TANG D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages[C]// Findings of the Association for Computational Linguistics (EMNLP 2020). Association for Computational Linguistics, 2020: 1536-1547.

GUO Liwei, born in 1998, postgraduate. His main research interests include software engineering and software testing.

LIU Yong, born in 1984, Ph.D, professor, master supervisor, is a member of CCF(No. 48965M). His main research interests include software engineering and software testing.