软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3093−3114 [doi: 10.13328/j.cnki.jos.007113]

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

基于规则与学习的变异技术对比研究*

贡志豪[1,2], 陈逸洲[1,2], 陈俊洁[3], 郝 丹[1,2]

[1](高可信软件技术教育部重点实验室(北京大学), 北京 100871)

[2](北京大学 计算机学院, 北京 100871)

[3](天津大学 智能与计算学部, 天津 300350)

通信作者: 郝丹, E-mail: haodan@pku.edu.cn

摘 要: 变异测试是一种有效的软件测试技术, 通过生成变异体来模拟软件缺陷, 帮助提升现有测试套件的缺陷检测能力. 变异体的质量对于变异测试的有效性具有显著影响. 传统的变异测试方法通常采用人工设计的基于语法规则的变异算子生成变异体, 并已取得一定的研究成果. 近年来, 许多研究开始结合深度学习技术, 通过学习开源项目历史代码生成变异体. 目前, 该新方法在变异体生成方面取得了初步的成果. 基于语法规则和基于学习的两种变异技术, 其机理不同, 但其目标均是通过生成变异体来提高测试套件的缺陷检测能力, 因此, 全面比较这两种变异技术对于变异测试及其下游任务至关重要. 针对这一问题, 设计实现了一项针对基于语法规则和基于学习的变异技术的实证研究, 旨在了解不同机理的变异技术在变异测试任务上的性能以及生成的变异体在程序语义上的差异性. 具体地, 以 Defect4J v1.2.0 数据集为实验对象, 比较以 MAJOR 和 PIT 为代表的基于语法规则的变异技术和以 DeepMutation、μBERT 和 LEAM 为代表的基于深度学习的变异技术. 实验结果表明: 基于规则与学习的变异技术均可有效支持变异测试实践, 但 MAJOR 的测试效果最优, 能够检测 85.4%的真实缺陷. 在语义表示上, MAJOR 具有最强的语义代表能力, 基于其构造的测试套件能够杀死其余变异技术生成的超过 95%占比的变异体. 在缺陷表征上, 两类技术均具有独特性.

关键词: 变异测试; 变异分析; 实证研究; 缺陷检测

中图法分类号: TP311

中文引用格式: 贡志豪, 陈逸洲, 陈俊洁, 郝丹. 基于规则与学习的变异技术对比研究. 软件学报, 2024, 35(7): 3093−3114. http://www.jos.org.cn/1000-9825/7113.htm

英文引用格式: Gong ZH, Chen YZ, Chen JJ, Hao D. Comparison Research on Rule-based and Learning-based Mutation Techniques. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3093−3114 (in Chinese). http://www.jos.org.cn/1000-9825/7113.htm

Comparison Research on Rule-based and Learning-based Mutation Techniques

GONG Zhi-Hao[1,2], CHEN Yi-Zhou[1,2], CHEN Jun-Jie[3], HAO Dan[1,2]

[1](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[2](School of Computer Science, Peking University, Beijing 100871, China)

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

Abstract: Mutation testing is an effective software testing technique. It helps improve the defect detection capability of an existing test suite by generating mutants that simulate software defects. The quality of mutants has a significant impact on the effectiveness of mutation testing. The traditional mutation testing approach typically employs manually designed syntactic rule-based mutation operators to generate mutants, and has achieved some academic success. In recent years, many studies have started to incorporate deep learning

* 基金项目: 国家重点研发计划(2022YFB4501902); 国家自然科学基金(62232001)
本文由“面向复杂软件的缺陷检测与修复技术”专题特约编辑张路教授、刘辉教授、姜佳君副研究员、王博博士推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-09

techniques to generate mutants by learning historical code from open source projects. This new approach has achieved preliminary progress in mutant generation. A comprehensive comparison of the two mutation techniques, i.e. rule-based and learning-based, which have different mechanisms but both aim to improve the defect detection capability of the test suite by mutation, is crucial for mutation testing and its downstream tasks. To handle the problem, this study designs and implements an empirical study of rule-based and learning-based mutation techniques, aiming to understand the performance of mutation techniques with different mechanisms on the task of mutation testing, as well as the variability of the generated mutants in terms of program semantics. Specifically, this study uses the Defect4J v1.2.0 dataset to compare the syntactic rule-based mutation techniques represented by MAJOR and PIT with the deep learning-based mutation techniques represented by DeepMutation, μBERT, and LEAM. The experimental results show that both rule-based and learning-based mutation techniques can effectively support mutation testing practices, but MAJOR has the best testing performance and is able to detect 85.4% of real defects. In terms of semantic representation, MAJOR has the strongest semantic representation capability, and its constructed test suite is able to kill more than 95% of the mutants generated by other mutation techniques. In terms of defect representation, both types of techniques are unique.

Key words: mutation testing; mutation analysis; empirical study; defect detection

1 引 言

变异测试是一种有效的软件测试[1]技术, 可用于评估测试套件识别软件缺陷(或者称为软件故障)的能力. 变异测试通过在原程序中注入人为设计的语法错误, 产生一组称为“变异体”的衍生版本, 用于模拟软件中潜在的缺陷, 并可评估测试套件的测试充分性. 根据测试结果, 变异测试还能够揭露现有测试套件的不足, 指导测试用例设计并提升测试套件的可靠性.

变异体的质量对于变异测试的有效性具有非常显著的影响. 过去工作通常采用一系列预先定义的、基于语法的变异规则(称为变异算子)进行程序变异, 典型的技术有 Jester[2]、MuJava[3]、Jumble[4]、Javalanche[5]、MAJOR[6]和 PIT[7]. 这些基于规则的变异技术已在变异测试[8]及其衍生应用[9−11]中取得一定研究成果. 近年来, 许多工作结合深度学习技术, 通过学习开源项目历史代码生成变异体, 典型的技术有 DeepMutation(简称 DM)[12]、μBERT[13]和 LEAM[14]. 目前, 该新方法在变异体生成方面取得了令人鼓舞的结果.

然而, 我们注意到, 基于程序语法规则变异和利用神经网络隐式推理是两类截然不同的变异机理. 这两类变异机理制导的变异体不仅在语法形态上存在显著差异, 且有可能在程序语义上呈现不同的特性, 最终影响变异测试的效果. 遗憾的是, 尚缺乏实证研究比较基于规则和基于学习这两大类变异技术在性能上的差异性和正交性.

为了填补这一研究空白, 本文选择了最具代表性的两种基于语法规则的变异技术(MAJOR 和 PIT)和 3 种最先进的基于深度学习的变异技术(DeepMutation、μBERT 和 LEAM)进行全面比对分析. 与以往工作不同, 本文不仅关注了它们在变异测试任务上的整体性能, 还从程序语义的角度研究了其生成变异体的语义特性, 包括互相代表性与模拟真实缺陷的能力, 并发现它们之间存在语义正交性, 为来研究提供了新的见解. 因此, 本文旨在通过比较两类不同机理的变异技术, 从定量和定性两个维度评估它们的性能和语义特性, 为改进和提升现有变异技术提供了新的方向.

具体地, 本文以 Defect4J v1.2.0 数据集为实验对象, 比较 MAJOR 和 PIT 为代表的基于语法规则的变异技术和以 DeepMutation、μBERT 和 LEAM 为代表的基于深度学习的变异技术. 结果表明: 基于规则与学习的技术均可有效支持变异测试实践, 但 MAJOR 的测试效果最优, 能够检测 85.4%的真实缺陷. 在语义表示上, MAJOR 具有最强的语义代表能力, 基于其构造的测试套件能够杀死其余变异技术生成的超过 95%占比的变异体. 在缺陷表征上, 两类技术均具有独特性. 例如: 基于规则的变异技术对特定的代码元素具有更稳定的变异能力, 而基于学习的变异技术能够结合上下文语义对程序作更灵活的变异.

本文的主要贡献如下:

- 开展较为全面的对照实验. 调研并检验 5 种基于 Java 的变异技术的性能, 包括 2 种最广泛使用的基于规则的变异技术与 3 种最先进的基于学习的变异技术.
- 在大规模数据集上深入分析深度学习技术与传统技术生成的变异体在程序语义上的差异性与正交性.

本文第 2 节介绍变异测试的背景知识与相关工作. 第 3 节通过两个真实的缺陷阐明研究动机. 第 4 节描述具体的实验设计. 第 5 节展示并分析实验结果. 第 6 节对实验结果进行讨论. 第 7 节讨论对未来工作的可能影响. 第 8 节总结全文并展望未来工作.

2 背景与相关工作

2.1 变异测试基本概念

变异测试(mutation testing)也称为基于变异的测试(mutation-based testing)[15], 它通过在原程序中注入人为设计的语法错误来模拟软件缺陷, 根据测试套件在这些“人工”错误上的执行结果, 评估测试套件的有效性、揭示其不足并针对性地改进.

变异测试中, 待测试程序称为原程序(original program), 测试用例集合也称为测试套件(test suite). 在原程序中注入人为设计的语法错误的过程称为变异(mutation), 变异原程序得到的衍生程序称为变异体(mutant). 传统变异测试通常采用一系列预先定义的语法变换规则, 或者称为变异算子(mutation operator), 对原程序中特定代码元素变异. 利用变异算子进行一次变异得到的程序称为一阶变异体(first order mutant), 进行二次及以上变异得到的程序称为高阶变异体(higher order mutant).

基于原程序 P 生成变异体集 M 后, 开发者执行测试套件 T 并分析测试用例执行情况. 具体地, 对于某个变异体 m, 如果 T 中存在测试用例 t, 其执行结果在 m 与 P 上不同(即能区分 m 与 P 的行为), 那么称测试用例 t 能够杀死(kill)变异体 m, 或者称变异体 m 被测试用例 t 杀死; 否则, 则称变异体 m 为存活变异体(surviving mutant). 在存活变异体中, 有些变异体虽然在语法形态上与原程序存在差异, 但在功能上与原程序完全等价, 这些与原程序语义等价的变异体称为等价变异体(equivalent mutant).

在基于变异的充分性准则下, 通常以非等价变异体为测试目标, 并采用变异分数(mutation score)[16]指标评估现有测试套件的充分性. 变异分数的定义如公式(1)所示, 数值上等于测试套件杀死的变异体数量与所有非等价变异体的比值.

$$MS=\frac{|Kill(M,T)|}{|M.all|-|M.equiv|}\times 100\% \tag{1}$$

其中, $M.all$ 代表全体变异体集, $M.equiv$ 代表等价变异体集, $Kill(M,T)$代表 T 在 M 上杀死的变异体集. 当测试套件能杀死所有非等价变异体时, 其取得 100%的变异分数, 此时认为该测试套件是一个变异充分的测试套件(mutation adequate test suite).

2.2 变异技术研究

随着变异测试的发展, 现有文献提出许多不同思想指导的变异体生成技术, 简称为变异技术(mutation technique)[14]. 这些变异技术曾用以支持变异测试的各项研究[10,16−23]与实践[11,24,25]. 过去, 关于变异测试的研究大多基于优秀程序员假设[26]与耦合效应假设[27−29], 因此, 传统的变异技术大多遵循程序语法, 采用预先定义的、人工设计的变异算子进行变异, 进而生成形式简单、符合语法的变异体. 典型的基于语法规则的变异技术包括 Jester[2]、MuJava[3]、Jumble[4]、Javalanche[5]、MAJOR[6]和 PIT[7]等.

研究表明, 基于简单语法规则变换产生的变异体不能非常有效地表示真实缺陷[30,31]. 为了提升传统变异技术生成变异体的质量, 研究人员提出了许多新颖的变异技术. Brown 等人[32]首先提出从开源项目的历史提交中挖掘变异模式用以指导变异体生成. 具体地, Brown 等人[32]从开源社区 GitHub 中的 C 语言项目中使用自动化手段挖掘“野生变异体(wild-caught mutant)”, 即实际开发代码中的代码变动模式, 用于弥补经典的变异技术生成变异体的局限性. Beller 等人[25]提出了 Mutation Monkey, 它能半自动化地从历史变更中挖掘变换模式并归纳为变异算子. Khanfir 等人[33]提出了基于信息检索的缺陷注入技术 IBIR, 通过收集缺陷报告的自然语言特征与代码语义特征预测缺陷注入位置, 再通过修复模型 TBar[34]的逆向操作进行程序变异.

在数据驱动的条件下, 深度学习在计算机科学的许多领域都展现了不俗的表现. 目前也有一些工作尝试

利用神经网络进行代码变异. 受 Brown 等人[32]的“野生变异体”启发, Tufano 等人[12]提出了 DeepMutation, 利用预训练的神经网络机器翻译模型学习从缺陷程序到正确程序的修复模式. 为了提升神经网络变异程序的语法正确性并生成质量更高的高阶变异体, Tian 等人[14]提出了 LEAM, 利用一个语法指导的、基于编码器解码器架构的 Transformer 进行代码变异. 由于大型预训练语言模型在语义理解与文本生成上的卓越表现, Renzo 等人[13]提出了μBERT, 通过 Mask 程序中某一位置的 Token 并调用预训练语言模型 CodeBERT[35]对程序进行变异.

2.3 变异测试实证研究

变异测试是一种测试充分性准则(adequacy criterion). 已有实证研究主要围绕其作为充分性准则的有效性展开, 并发现以变异体为测试目标设计测试用例的实践方式优于其他结构化充分性准则(例如语句覆盖、代码块覆盖、分支覆盖和路径等结构化充分性准则)[30,36−41]. Just 等人[30]发现: 变异和语句覆盖均与缺陷检测相关, 其中, 使用变异体作为测试目标具有更高的相关性. Chekham 等人[36]通过实验发现: 基于强变异(strong mutation)的测试准则与缺陷检测直接具有强相关性, 而基于语句覆盖、分支覆盖和弱变异这 3 种充分性准则与缺陷检测之间只具有较弱的相关性. Papadakis 等人[37]使用统计学相关性指标计算了变异分数、测试套件大小与缺陷检测之间的相关性, 认为在忽略测试套件大小的条件下, 变异分数与缺陷检测之间存在较强的相关性; 考虑测试套件大小后, 两者之间之间仍然存在中等至微弱的相关性. 此外, 他们发现: 当测试套件具有较高的变异分数时, 可以显著地提升缺陷检测的效果. Chen 等人[5]考虑数据不平衡问题, 在改进 Papadakis 等人在文献[37]中的实验条件后, 重新比较了变异与语句覆盖这两种常见的充分性准则的有效性. 他们发现: 在基于语句覆盖的测试准则达到充分后, 再采取基于变异的测试准则, 能够得到更好的缺陷检测效果.

此外, 一些工作也对现有变异技术展开比对研究. Kintis 等人[42]在 Defects4J 1.0 上评估了 4 种传统变异技术(MuJava、MAJOR、PIT 和 PITrv)的缺陷检测能力, 并通过人工分析评价其优缺点. Gopinath 等人[40]在若干 Java 项目上基于多种指标(例如变异分数、最小变异体集合等)评估了 3 种传统工具(Judy、MAJOR 和 PIT)的效果, 发现这些工具在测试性能具有较大的方差, 难以判断孰优孰劣. Ojdanic 等人[43]在 Defects4J 2.0 上使用 4 种变异技术(PIT、DeepMutation、μBERT 和 IBIR)探究了变异体与原程序之间的语法变化与语义变化的相关性. Tian 等人[14]在 Defects4J 1.5 上比较了 4 种变异技术(MAJOR、PIT、DeepMutation 和 LEAM)在 3 个软工任务(变异测试、测试用例排序和缺陷定位)上的性能. Ojdanic 等人[44]在 Defects4J 2.0 上利用基于学习的变异体选择策略比较了 4 种变异技术(μBERT、IBIR、PIT 和 PITrv)的缺陷检测能力.

目前, 越来越多的工作结合深度学习技术, 利用神经网络自动化地推理生成变异体程序. 然而, 我们注意到, 基于程序语法规则变异和利用神经网络隐式推理是两种截然不同的变异机理. 这两类不同的变异机理不仅会对变异体的语法形态带来显著影响, 且有可能使变异体的语义呈现不同的特性, 最终影响变异测试的效果. 遗憾的是, 目前仍然缺乏全面的实证研究比较基于规则和基于学习这两大类变异技术在性能上的差异性和正交性.

为了填补这一研究空白, 本文选择了两种基于语法规则的变异技术(MAJOR 和 PIT)以及 3 种基于深度学习的变异技术(DeepMutation、μBERT 和 LEAM)进行全面地比较和分析. 本文不仅评估了不同变异技术在测试性能上的优劣(即缺陷检测效果以及变异开销等), 还面向这两类技术从程序语义角度出发, 定量和定性地研究其生成变异体的语义特性, 发现这两类技术生成的变异体存在一定的正交性, 可以结合使用,提升变异测试整体效果. 这一研究角度为未来这两类技术的发展提供了有价值的洞察.

综上, 本文从多个维度对基于规则的与基于学习的两类变异技术展开实证研究, 不仅评估了其在变异测试任务的整体性能, 还借助动态程序语义分析其生成变异体的语义特性, 为研究人员改进和提升现有变异技术提供新的见解.

简言之, 本文工作将从如下 3 个方面开展.

1) 比对基于规则和基于学习两类变异技术在变异测试任务上的有效性与性能;

2) 探究这两类变异技术生成变异体的语义替补性;

3) 分析这两类变异技术生成的变异体模拟缺陷行为的语义独特性.

3 研究动机

本节基于 Defects4J (版本 1.2.0)的两个例子, 介绍基于规则和基于学习的变异技术生成的变异体的差别. 以其为例, 说明两类技术在变异机理上的差异, 以及变异体的形态和语义的特性. 由于空间限制, 示例只展示一部分与缺陷相关的代码. 我们使用绿色高亮修正代码, 红色高亮缺陷代码.

图 1 展示了 Time-10 的缺陷与变异代码. Time-10 是 Defects4J 数据集 Joda-Time 项目的第 10 个真实缺陷. 在该例子中, 开发者在 chrono.set 调用方法的第 2 个参数中传递了错误数值 0L, 其补丁是将 chrono.set 调用方法第 2 个参数修改为 START_1972 (数值上等于 2L×365L×86400L×1000L). 在变异过程中, MAJOR 的 EVR (expression value replacement)变异算子将 START_1972 的赋值从 2L×365L×86400L×1000L 变异为 0L, 从而复现了该缺陷. 在实验中, 基于学习的变异技术均无法复现该缺陷. 该例子说明: 经过精心设计的变异算子对于特定语法元素具有较为鲁棒的变异能力, 能够模拟开发过程中常规的程序缺陷.

缺陷ID: **Time-10** 缺陷所在类: **BaseSingleFieldPeriod.Java**

```
@@ -49,7 +49,6 @@ public abstract class BaseSingleFieldPeriod
- private static final long START_1972 = 2L * 365L * 86400L * 1000L;


@@ -102,7 +101,7 @@ public abstract class BaseSingleFieldPeriod
- int[] values = chrono.get(zeroInstance, chrono.set(start,START_1972), chrono.set(end, START_1972));
+ int[] values = chrono.get(zeroInstance, chrono.set(start, 0L), chrono.set(end, 0L));
```

MAJOR

EVR: 2L * 365L * 86400L * 1000L |==> 0L

图 1 Time-10 缺陷分析

图 2 展示了 Chart-8 的缺陷与变异代码. Chart-8 是 Defects4J 数据集 JFreeChart 项目的第 8 个真实缺陷.

缺陷ID: **Chart-8** 缺陷所在类: **Week.Java**

```
@@ -172,7 +172,7 @@ public class Week extends RegularTimePeriod implements Serializable {
    public Week(Date time, TimeZone zone) {
        // defer argument checking...
-       this(time, zone, Locale.getDefault());
+       this(time, RegularTimePeriod.DEFAULT_TIME_ZONE, Locale.getDefault());
```

LEAM

```
- this(time);
+ this(time, zone, Locale.getDefault());
```

图 2 Chart-8 缺陷分析

在该例子中, 开发者错误地将默认时区变量 *RegularTimePeriod.DEFAULT_TIME_ZONE* 作为 this 构造器(也就是 Week 类)的第 2 个参数传入, 其补丁是将默认时区变量替换成变量 zone. 在实验中, 该缺陷仅被基于学习的变异技术 LEAM 识别. 在变异过程中, LEAM 将语句 *this*(*time*,*zone*,*Locale.getDefault*(·))变异为 *this*(*time*),即实现了构造器重载操作.

Week 类中存在 *Week*(*Date time*)构造器, 其内部又会调用 *Week*(*time*,*RegularTimePeriod.DEFAULT_TIME_ZONE*,*Locale.getDefault*(·))从而触发缺陷. 因此, LEAM 的这一变异体成功复现缺陷 Chart-8. 由于缺乏特定的

变异算子, 基于规则的变异技术均无法复现该缺陷. 该例子说明: 基于学习的变异技术可以生成形式更为多样、自然的变异体, 能够帮助模拟程序中的潜在缺陷.

由于设计思路不同, 不同变异机理的技术生成的变异体在语法形态上存在一定的差异. 具体地, 传统的变异技术(例如 MAJOR)通过预先设计的语法变换规则(变异算子)对特定语法元素进行变异操作, 而基于学习的变异技术则基于深度学习模型隐式地推理复杂的变异规则并生成变异体. 因此, 在语法形态上, 基于规则的变异技术生成的变异体相对有迹可循, 遵从语法变换规则; 而基于学习的变异技术生成的变异体相对自然多样, 基于训练资料和程序上下文语义推理而得. 然而, 不同技术在变异机理上的差异不仅会影响变异体的语法形态, 还有可能导致变异体在语义层面出现差异, 从而影响变异测试的效果. 本文旨在深入探究基于规则和基于学习的变异技术在生成变异体的语义差异性.

4 实验设计

本文的总体实验流程如图 3 所示. 在第 4.1 节中, 介绍本文关注的 3 个研究问题. 在第 4.2 节中, 介绍实验选择的变异技术以及选择原因. 在第 4.3 节中, 介绍实验中使用的测试框架以及变异所需的实验对象. 在第 4.4 节中, 对变异技术生成的变异体做定量分析. 在第 4.5 节中, 介绍实验所需控制的外部变量. 在第 4.6 节中, 介绍实验的环境设置. 在第 4.7 节中, 介绍实验评估所需的统计学指标.

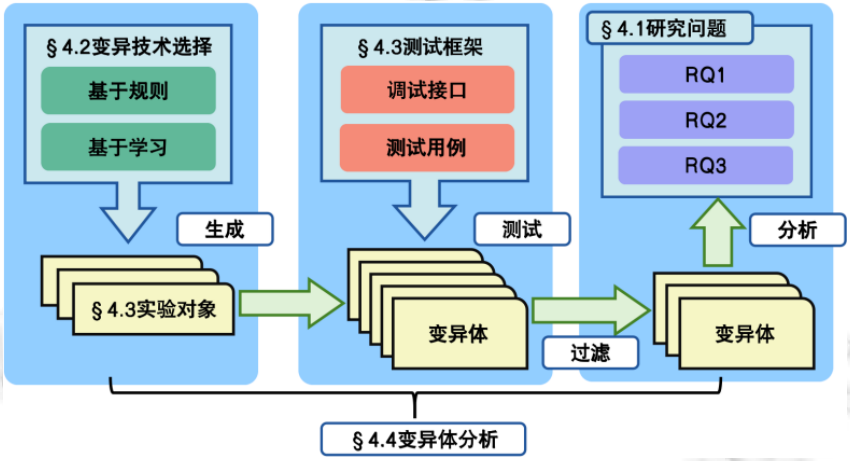


图 3 实验流程总体框架

4.1 研究问题

由于变异机理不同, 基于规则和基于学习的变异技术生成的变异体在语法形态上存在差异, 并可能对其程序语义带来未知影响. 为了全面比对两类技术生成变异体的差异性, 本文探究 3 个研究问题.

- RQ1: 两类变异技术对于提升测试套件缺陷检测能力的有效性?

第 1 个问题探究变异技术在变异测试任务上的整体性能. 为此, 以不同变异技术生成的非等价变异体为测试目标, 指导测试用例设计构造相应的测试套件, 统计其在 Defects4J 上的缺陷检测效果.

- RQ2: 两类变异技术生成的变异体的语义替补性?

第 2 个问题探究变异技术生成的变异体在语义上的替补关系. 为此, 以不同变异技术生成的变异体为测试目标, 指导测试套件的构造,并比较该测试套件在其余变异技术生成变异体上的测试效果.

- RQ3: 两类变异技术生成的变异体表征缺陷的独特性?

第 3 个问题探究变异技术生成的变异体在表征缺陷上的独特能力. 为此, 分析不同变异技术生成变异体的对于真实缺陷的模拟效果, 并研究不同机理的变异体的缺陷表征能力.

4.2 变异技术选择

为了评估不同机理的变异方式对于变异测试的影响, 考虑现有文献中两类主流的变异技术.

具体地, 选取 MAJOR[6]和 PIT[7]为代表的基于语法规则的变异技术和以 DeepMutation (简称 DM)[12]、μBERT[13]和 LEAM[14]为代表的基于深度学习的变异技术. 表 1 从多个维度概括了这些变异技术的特点, 下面简要介绍这 5 种变异技术的机理以及相应的实验设置.

表 1 变异技术多维分析

技术名称	编程语言	变异方式	变异阶数	变异算子/模型	训练数据集
MAJOR	Java	源码	1	9 种变异算子	–
PIT	Java	字节码	1	100 多种变异算子	–
DeepMutation (DM)	Java	源码	高	NMT-based encoder-decoder	fix-bug 数据集
μBERT	Java	源码	1	CodeBERT	多语言语料库
LEAM	Java	源码	高	syntax-guided encoder-decoder	fix-bug 数据集

- MAJOR

MAJOR 是一种源码级变异技术, 通过操作程序的抽象语法树(abstract syntax tree)进行变异. 目前, MAJOR 集成了 9 种变异算子, 分别为 AOR (arithmetic operator replacement)、COR (conditional operator replacement)、EVR (expression value replacement)、LOR (logical operator replacement)、LVR (literal value replacement)、ORU (operator replacement unary)、ROR (relational operator replacement)、SOR (shift operator replacement)和 STD (statement deletion). 在变异过程中, MAJOR 每次只选择 1 种变异算子进行变异操作, 生成一阶变异体. 实验中采用 MAJOR 1.3.4 开源版本. 为了比较的全面性, 开启 MAJOR1.3.4 支持的全部 9 种变异算子.

- PIT

PIT 是一种字节码层面的变异技术, 通过对 Java 字节码进行变异, 从而避免编译开销. 目前, PIT 对常用的变异算子进行了拓展, 总共实现了 29 类超过 120 种任务特定的变异算子[7]. 在变异过程中, PIT 也只对原程序进行一阶变异. 实验采用 PIT 1.7.4 版本. 为了比较的全面性, 开启 PIT 1.7.4 支持的超过 100 种变异算子.

- DeepMutation (DM)

DeepMutation[12]是一个基于神经网络机器翻译模型(neural machine translation)的变异技术, 它通过学习开源 Java 项目中的缺陷修复提交直接对程序进行变异. 实验采用文献[45]开源的 DeepMutation. 根据文献描述, 对原程序中每个方法只生成一个变异体.

- μBERT

μBERT[13]是一个基于预训练语言模型(CodeBERT[35])的变异技术. μBERT 的核心是 CodeBERT, 一个在包括 Java 在内的 6 种程序语言、超过 6 百万程序上进行预训练的大型语言模型. 在变异过程中, μBERT 通过词法解析原程序, 选择程序中某个 Token 进行 Mask, 并交由 CodeBERT 预测候选答案. 根据预测, μBERT 替换 Mask 处的 Token 得到变异体. 实验采用文献[13]开源的μBERT. 根据文献描述, 采用μBERT 默认配置, 对原程序中每一行代码语句都进行所有可行变异.

- LEAM

LEAM[14]是一个语法指导的、基于编码器解码器架构(syntax-guided encoder-decoder architecture)的变异技术. 为了更好地理解程序上下文与结构信息, LEAM 将程序表示成一棵抽象语法树, 并编码其语义与结构信息; 为了提高生成程序的语法正确性, LEAM 引入了人为拓展的 Java 语法规则. 变异过程中, LEAM 首先使用预测模块预测变异位置, 然后使用变异模块进行程序变异. LEAM 还采取 Beam Search 算法, 确保生成变异体的多样性. 实验采用文献[34]开源的 LEAM. 根据论文描述, 实验设置 BeamSearch 大小为 64.

4.3 测试框架与实验对象

实验采用 Defects4J[46]变异测试框架, 版本号为 1.2.0. Defects4J 提供了一系列封装好的变异测试接口, 帮

助研究人员测试 Java 程序并获取执行结果. 同时, Defects4J 提供了数百个从开源项目中挖掘的真实缺陷, 这些缺陷被广泛用于软件测试[8]、缺陷定位[9,23]和程序修复领域[11]. Defects4J 对每个缺陷都做了最小化处理, 即剔除了修复版本(fixed version)与缺陷版本(buggy version)中与当前缺陷无关的变更代码, 确保两个版本中的修改类仅涉及当前缺陷的变更信息. Defects4J 为每个实验对象都提供了一个开发者测试套件(developer test suite), 其中包含至少 1 个能在缺陷版本上触发缺陷的测试用例(triggering test).

表 2 描述了实验涉及项目的基本信息, 其中, 列“项目 ID”表示项目标识符, 列“项目名称”和列“缺陷数目”分别表示项目全名以及包含的缺陷总数, 列“测试用例数”和列“代码行数”表示该项目首个版本的测试用例个数与代码行数. 现阶段, 实验选择了 Defects4J 中 6 个项目(Chart、Closure、Lang、Math、Mockito 和 Time)、总计 395 个真实缺陷进行实验.

表 2 Defects4J 实验对象基本信息

项目 ID	项目名称	缺陷数目	测试用例数	代码行数	分析缺陷
Chart	JFreeChart	26	2 193	152k	26
Closure	Google Closure compiler	133	7 911	261k	80
Lang	Apache commons-lang	65	2 191	60k	52
Math	Apache commons-math	106	4 378	171k	90
Mockito	Mockito	38	1 379	57k	35
Time	Joda-Time	27	4 042	83k	23
总计	–	395	–	–	306

4.4 变异体分析

根据现有文献[14,37]的做法, 将 Defects4J 中每个研究对象的修复版本视为正确提交, 缺陷版本视为包含一个缺陷的提交. 首先, 在修复版本中的修改类上生成变异体; 然后, 基于变异分析构造测试套件; 最后, 在缺陷版本上进行测试. 如果测试套件包含触发缺陷的测试用例, 则认为成功检测该缺陷.

总体上, 实验流程可分为生成、测试、过滤和分析.

- 在生成阶段, 基于不同变异技术, 在每个研究对象修复版本的修改类上生成变异体.
- 在测试阶段, 执行不同变异技术生成的变异体并获取其执行结果. 具体地, 先对变异体进行编译, 丢弃编译失败的变异体, 再使用开发者测试套件对通过编译的变异体测试. 实验设置超时时间为 5 min, 最终收集每个变异体的测试结果.
- 在过滤阶段, 根据动态执行结果过滤掉重复变异体、等价变异体以及其他不可用变异体(程序执行中发生超时、内存错误等异常的变异体). 由于识别等价变异体是一个不可判定的问题[47], 目前尚无非常有效的识别方法. 因此, 根据文献[34]的做法, 直接丢弃所有存活变异体.
- 最终, 将过滤后得到的变异体用于实验分析.

由于系统环境以及变异技术实现问题, 并非所有变异技术均能在每个实验对象上生成变异体, 也并非所有实验对象均可用于实验. 我们尽了最大的努力调试系统环境以及变异技术, 但仍然存在一些不可用的实验对象. 为了实验的公平性, 实验只在变异技术都能成功生成变异体的实验对象上进行. 其中, DM 技术无法在 Closure 全体实验对象上进行变异, 因此, Closure 上的实验对象基于其余 4 种变异技术的可行域选择. 最终采用的实验对象(称为分析缺陷)如表 2 第 6 列“分析缺陷”所示, 总计包括 306 个缺陷. 为了消除实验随机性,所有实验都进行了 20 次重复实验.

表 3 展示了不同变异技术生成(生成集)、通过编译(可编译集)和用于实验(分析集)的变异体数量. 平均而言, 在每一个文件上, MAJOR、PIT、DM、μBERT 和 LEAM 分别生成 550.7、2006.6、16.4、3044.9 和 1121.3 个变异体, 通过编译的有 530.8、1983.8、6.7、820.1 和 408.4 个变异体, 编译率为 96.4%、98.9%、41.1%、26.9% 和 36.4%. 总体上, 基于规则的变异技术具有相对较高(>96%)的编译通过率, 而基于学习的变异技术编译通过率较低(26.9%–41.1%). 通过过滤, 最终在每个实验对象上, MAJOR、PIT、DM、μBERT 和 LEAM 用于分析的变异体平均数目为 360.1、1078.3、5.0、601.5 和 299.4.

表 3 Defects4J 实验对象基本信息

变异体集合	变异技术	最小	平均	最大
生成集	MAJOR	2.0	550.7	3 628.0
	PIT	12.0	2 006.6	13 822.0
	DM	1.0	16.4	110.0
	μBERT	15.0	3 044.9	20 345.0
	LEAM	8.0	1 121.3	8 717.0
可编译集	MAJOR	0.0	530.8	3 510.0
	PIT	12.0	1 983.8	13 426.0
	DM	0.0	6.7	35.0
	μBERT	0.0	820.1	6 164.0
	LEAM	0.0	408.4	6 638.0
分析集	MAJOR	0.0	360.1	2 673.0
	PIT	0.0	1 078.3	11 188.0
	DM	0.0	5.0	35.0
	μBERT	0.0	601.5	3 242.0
	LEAM	0.0	299.4	2 329.0

4.5 变量控制

- 变异体数量

如表 3 所示, 不同变异技术在变异体数量上存在较大差异. 虽然生成变异数量是变异技术的内在特性之一, 但为了比较的公平性, 仍然尝试控制该变量. 具体地, 实验分别在变异体的原始集合和控制数量后的集合上进行实验. 首先在每个实验对象上计算不同变异技术分析集大小的最小值 *mutant_min_num*; 然后通过随机采样从每种变异技术的分析集中采样 *mutant_min_num* 个变异体; 最终通过 20 次重复实验求取平均结果.

由表 3 可知, DM 的变异体数量远少于其余变异技术. 实验表明, 以 DM 为基准采样大小为 *mutant_min_num* 的分析集严重低估变异技术的有效性. 因此, 控制实验考虑 All、Control w/ DM 与 Control w/o DM 这 3 种设置, 分别代表使用全部分析集、考虑 DM 时控制变异体数量和不考虑 DM 时控制变异体数量.

- 测试套件大小

现有文献[36,42]表明, 测试套件的大小对于测试套件的有效性具有潜在影响. 因此, 实验中尝试控制测试套件大小这一变量. 具体地, 首先在每个实验对象上计算不同变异技术对应的测试套件大小的最大值 *test_max_num*; 然后, 基于 *test_max_num* 的固定百分比(例如 10%), 通过随机采样的方式从每个变异技术的测试套件中采样子集; 最终, 基于测试套件子集进行实验.

4.6 环境设置

实验的硬件环境为一台操作系统版本为 Ubuntu 18.04.4 LTS 的服务器, 其处理器为 80 Intel(R) Xeon(R) Gold 5218R CPU@2.10 GHz、显卡为 4 NVIDIA GeForce RTX 3090 GPU、内存大小为 256 GB.

实验的软件环境包括 Apache Maven v3.6.0, openjdk v1.8.0_362 与 PyTorch v1.12.1.

4.7 统计学指标

- 威尔康森符号秩检验[48]

威尔康森符号秩检验(Wilcoxon signed-rank test)是一种非参数检验(nonparametric test)方法, 它无须了解样本总体分布即可进行统计推断. 威尔康森符号秩检验非常适用于在两种不同条件下对于相同对象进行多次观测的实验场景. 实验中, 在置信度为 0.05 的情况下, 使用威尔康森符号秩检验以比较两种变异技术间的性能是否存在显著性差异.

- Vargha Delaney Effect Size $\hat{A}_{12}$ [49]

在比较两种技术某一性能指标时, 即便度量数据存在统计学意义上的显著性差异, 也并不一定代表这两种技术的性能具有明显差别. 因此, 采用 $\hat{A}_{12}$ 统计指标从数值上量化某一技术优越于另一技术的程度. 如果 $\hat{A}_{12}$ 的数值为 0.5, 那么代表这两种技术的性能表现相同(在 50%的情况下, 前者优于后者). 如果 $\hat{A}_{12}$ 的数值大

于 0.5, 那么代表前一种技术比后一种技术表现优越, 反之亦然. 通常, 当 $\hat{A}_{12}$ 的数值大于 0.71 时候, 可以认为前一种技术明显优于后一种技术; 当 $\hat{A}_{12}$ 的数值小于 0.29 时, 可以认为后一种技术明显优于前一种技术.

5 实验分析

5.1 RQ1

- 评估指标

真实缺陷检测率(real fault detection rate). 用于度量测试套件检测程序缺陷的有效性. 具体地, 在缺陷程序上执行测试套件, 如果测试套件中存在执行失败的测试用例, 就认为该测试套件具有缺陷检测能力.

变异体识别与缺陷检测相关性(correlation between mutant killing and real fault detection). 用于近似地度量基于变异的充分性准则与缺陷检测之间的相关性. 考虑到变异分数是连续变量, 而缺陷检测是二分变量,借鉴文献[38]的做法, 采用点二列相关性系数(point-biserial correlation coefficient)度量变异分数与缺陷检测二者之间的相关程度.

- 实验流程

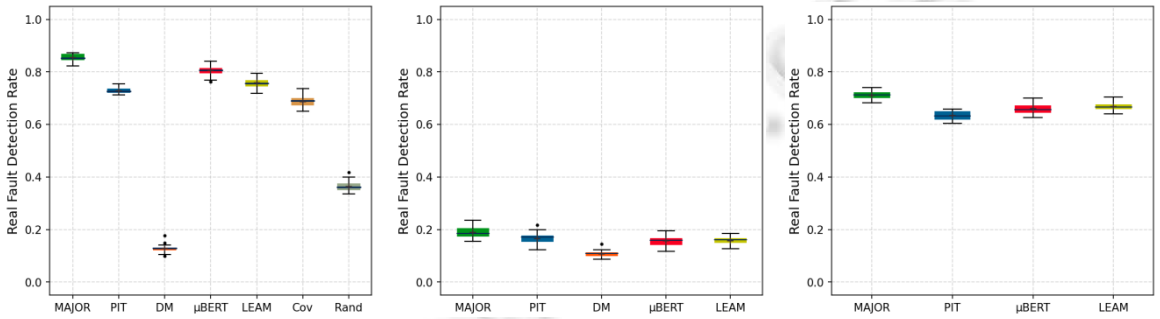
评估不同变异技术的真实缺陷检测率: 1) 在每个实验对象上, 使用不同变异技术的分析集, 采用贪心策略构建“最小化的测试套件(minimal test suite)”; 2) 在缺陷版本上执行该“最小化的测试套件”, 并判断缺陷检测结果; 3) 统计每种变异技术在所有实验对象上的平均缺陷检测率, 并使用统计学指标比较不同变异技术在缺陷检测效果上的显著性差异.

评估不同变异技术的变异体识别与缺陷检测相关性: 1) 通过随机采样从原始测试套件中构造若干大小相同的测试子集; 2) 执行测试子集并计算其变异分数; 3) 计算点二列相关性系数. 以 1%、10%、20%、30%、40%和 50%的比例, 从原有测试套件中随机采样 100 次进行实验分析.

上述流程充分借鉴了文献[26,50,51]的做法. 此外, 为了更全面地研究不同变异技术对于测试充分性的提升, 实验实现了两个基准模型, 即基于覆盖率的与基于随机采样的测试用例选择策略, 称为 Coverage (Cov)和 Random (Rand). Coverage 以未覆盖的代码语句为测试需求, Random 则通过无放回随机采样的方式从原始测试套件中选择测试用例.

- 结果分析

图 4 汇总了基于不同技术构造的测试套件的平均缺陷检测率, 横坐标表示罗列了不同技术的名称, 纵坐标表示缺陷检测率的平均值. 注意, 图 4 没有包含 Closure 的统计结果.



(a) All (b) Control w/ DM (c) Control w/o DM

图 4 基于不同变异技术的真实缺陷检测率

图 4(a)展示了在不控制变异体数量条件下的平均缺陷检测率. MAJOR、PIT、DM、μBERT 和 LEAM 的检测率分别为 0.854、0.729、0.129、0.804 和 0.758, Coverage 和 Random 的检测率分别为 0.688 和 0.364. 除了 DM 之外, MAJOR、PIT、μBERT 和 LEAM 这 4 种技术的平均缺陷检测率均高于 Coverage 和 Random.

图 4(b)展示了 Control w/ DM 条件下的平均缺陷检测率. MAJOR、PIT、DM、μBERT 和 LEAM 的表现分别为 0.190、0.165、0.108、0.158 和 0.157. 此时, Random 的表现为 0.276, 高于所有变异技术. 注意到: DM 对于每个方法仅生成 1 个变异体, 导致其分析集较小. 该结果表明, 过少的变异体会极大地降低变异测试的有效性. 由于 DM 的性能显著低于其余技术, 因此在后续的实验中, 不再针对性地讨论 DM 技术.

图 4(c)展示了 Control w/o DM 条件下的平均缺陷检测率. MAJOR、PIT、μBERT 和 LEAM 的表现分别为 0.711、0.634、0.660 和 0.668, 此时, Random 的表现为 0.331.

为了实验全面性, 我们也分析了包含 Closure 时的平均缺陷检测率: 在不控制变异体数量时, MAJOR、PIT、μBERT 和 LEAM 的结果为 0.855、0.718、0.797 和 0.741; 在控制变异体数量时, MAJOR、PIT、μBERT 和 LEAM 的结果为 0.680、0.601、0.633 和 0.657.

总体上, MAJOR 具有最高缺陷检测率, 能够检测 85.4%的真实缺陷; 其次为μBERT(80.4%); 而 PIT (72.9%)和 LEAM(75.8%)则具有相似效果. DM 技术由于生成的变异体数量较少, 仅检测到 12.9%的缺陷. 除了 DM 之外, 基于其余 4 种变异技术构造测试套件的缺陷检测能力均优于 Coverage(68.8%)和 Random(36.4%). 这一结果表明: 基于规则与基于学习的变异技术均可应用于变异测试, 帮助开发者提升测试套件的有效性.

表 4 通过威尔康森符号秩检验与 Vargha Delaney Effect Size 这两个统计学指标进一步展示不同变异技术在故障检测任务上效果的显著性差异. 其中, Sig.表示是否具有显著性差异, $\hat{A}_{12}$ 表示 Vargha Delaney Effect Size. 由表可知: MAJOR 在缺陷检测效果上显著地优于其余 4 种技术, DM 则在效果是显著地劣于其余 4 种技术; 在 All 与 Control w/o DM 的条件下, μBERT 显著优于 PIT. 除此之外, μBERT 与 LEAM 之间、LEAM 与 PIT 之间, 在性能上不存在显著性差异.

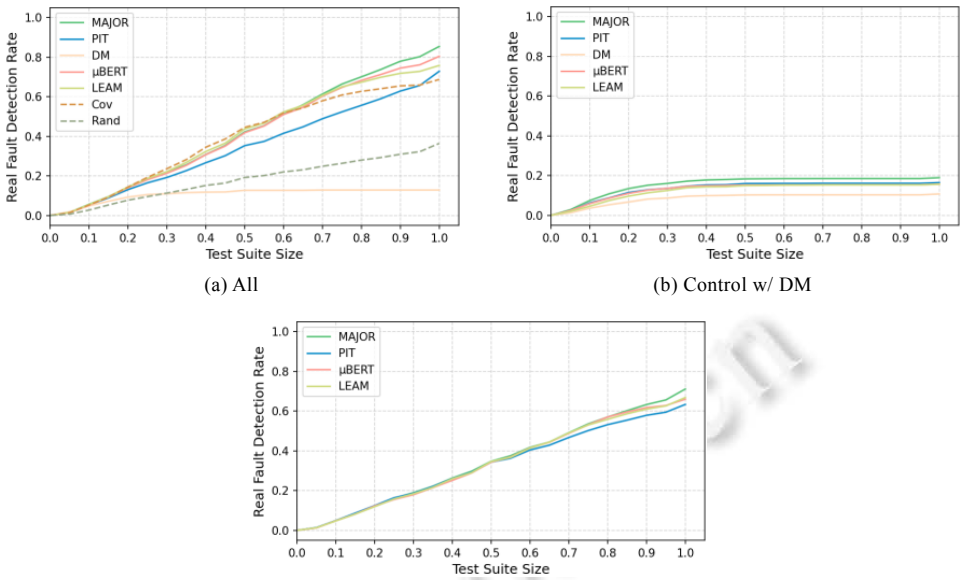
表 4 基于统计学指标的变异技术故障检测效果显著性差异比较

变异技术	All		Control w/ DM		Control w/o DM	
	Sig.	$\hat{A}_{12}$	Sig.	$\hat{A}_{12}$	Sig.	$\hat{A}_{12}$
MAJOR-PIT	√	0.574	√	0.535	√	0.567
MAJOR-DM	√	0.914	√	0.652	–	–
MAJOR-μBERT	√	0.533	√	0.537	√	0.542
MAJOR-LEAM	√	0.561	√	0.536	√	0.531
PIT-DM	√	0.828	√	0.620	–	–
PIT-μBERT	√	0.456	??	0.503	√	0.471
PIT-LEAM	×	0.470	×	0.505	×	0.467
DM-μBERT	√	0.102	√	0.385	–	–
DM-LEAM	√	0.116	√	0.377	–	–
μBERT-LEAM	×	0.534	×	0.499	×	0.494

图 5 汇总了在控制测试套件大小时, 基于不同技术构造的测试套件的平均缺陷检测率, 其中, 横坐标表示在构造 Minimal Test Suite 过程中测试套件的相对大小(以 *test_max_num* 为基准), 纵坐标表示平均缺陷检测率.

图 5(a)展示了不控制变异体数量时的平均缺陷检测率. 当 Test Suite Size 小于 0.6 时, MAJOR、μBERT、LEAM 与 Cov 效果差异不大; 当 Test Suite Size 为 0.6 时, 其检测效果分别为 0.514、0.510、0.523 与 0.518; 当 Test Suite Size 等于 0.7 时, 变异技术开始优于 Cov (例如 MAJOR 为 0.619, 而 COV 为 0.580). 这表明, 基于语句覆盖的测试准则比基于变异的测试准则更早失去效力[38]. 注意到 PIT 在 Test Suite Size 小于 0.95 时均低于 Cov, 只有当 Test Suite Size 接近 1.0 时才具有更高的检测能力(此时, PIT 为 0.729, Cov 为 0.688). 这是因为 PIT 存在许多实验性的变异算子, 设计变异算子的目的是实验研究, 而非实际应用. 因此, PIT 的变异体中存在一些非常规变异体, 对变异测试的有效性存在潜在影响.

图 5(b)与图 5(c)展示了在 Control w/ DM 与 Control w/o DM 条件下的平均缺陷检测率. 在 Control w/o DM 条件下, Test Suite Size 小于 0.7 时, 不同技术之间的效果差异并不明显(0.404–0.418); 随着 Test Suite Size 的增加, MAJOR 展示出最高的缺陷检测效果(0.711), μBERT 和 LEAM 的性能相似(0.634 和 0.668). 在包括 Closure 时, 我们也在实验中观测到一致的趋势.



(a) All (b) Control w/ DM

(c) Control w/o DM

图 5 控制测试套件大小条件下, 基于不同变异技术的真实缺陷检测率

表 5 展示了在控制测试套件大小的条件下, 不同变异技术对应的变异分数与缺陷检测的相关性. 当测试套件大小为 1%时, MAJOR、PIT、DM、μBERT 和 LEAM 的相关性分数较低, 分别为 0.178、0.174、0.082、0.163 和 0.119, 除了 MAJOR 与μBERT 以外, 其余技术的相关性指标均低于 Coverage 技术(0.155). 随着测试套件大小的增加, 不同的变异技术的相关性具有一定的提升. 例如: 测试套件大小为 20%时, MAJOR 从 0.178 提升到 0.201. 考虑到 DM 的特殊性, 我们只讨论 All 与 Control w/o DM 的情况. 结果表明: MAJOR 具有相对较高的相关性分数(0.201/0.191); PIT (0.198/0.171)和μBERT (0.188/0.183)效果相似; 而 LEAM 的相关性分数相对较低(0.142/0.137). 我们分析, 这是因为 MAJOR 在变异算子实施过程中实现了去冗余[48], 一定程度上保证了测试的有效性与效率.

表 5 变异分数与缺陷检测之间的相关性

变异体数量	技术	测试套件大小					
		1%	10%	20%	30%	40%	50%
All	MAJOR	**0.178**	**0.185**	**0.201**	**0.200**	**0.197**	**0.197**
	PIT	0.174	0.184	0.198	0.197	0.193	0.193
	DM	0.082	0.059	0.069	0.075	0.070	0.063
	μBERT	0.163	0.176	0.188	0.185	0.186	0.185
	LEAM	0.119	0.123	0.142	0.140	0.136	0.145
	Coverage	0.155	0.170	0.168	0.155	0.150	0.149
Control w/ DM	MAJOR	0.115	0.113	0.117	0.119	0.115	0.116
	PIT	**0.118**	**0.119**	**0.125**	**0.128**	**0.135**	**0.125**
	DM	0.069	0.048	0.056	0.070	0.065	0.053
	μBERT	0.105	0.103	0.084	0.088	0.089	0.080
	LEAM	0.778	0.539	0.069	0.077	0.074	0.077
Control w/o DM	MAJOR	**0.174**	0.177	**0.191**	**0.197**	**0.193**	**0.193**
	PIT	0.161	0.165	0.171	0.175	0.175	0.175
	μBERT	0.161	**0.179**	0.183	0.182	0.183	0.178
	LEAM	0.116	0.124	0.137	0.145	0.141	0.148

- RQ1 总结

总体上, 除了 DM 之外, 不同类别的变异技术均可以提升测试套件的缺陷检测能力, 但在性能上存在一定的差异. 特别地, 虽然 MAJOR 是传统的基于规则的变异技术, 但是基于其构造的测试套件却具有最好的缺陷检测效果. MAJOR 能检测实验对象中 85.4%的真实缺陷, 不仅优于传统技术 PIT (72.9%), 甚至比两种基于

深度学习的变异技术μBERT (80.4%)和 LEAM (75.8%)分别有 5.0%和 9.6%的提升.

5.2 RQ2

- 评估指标

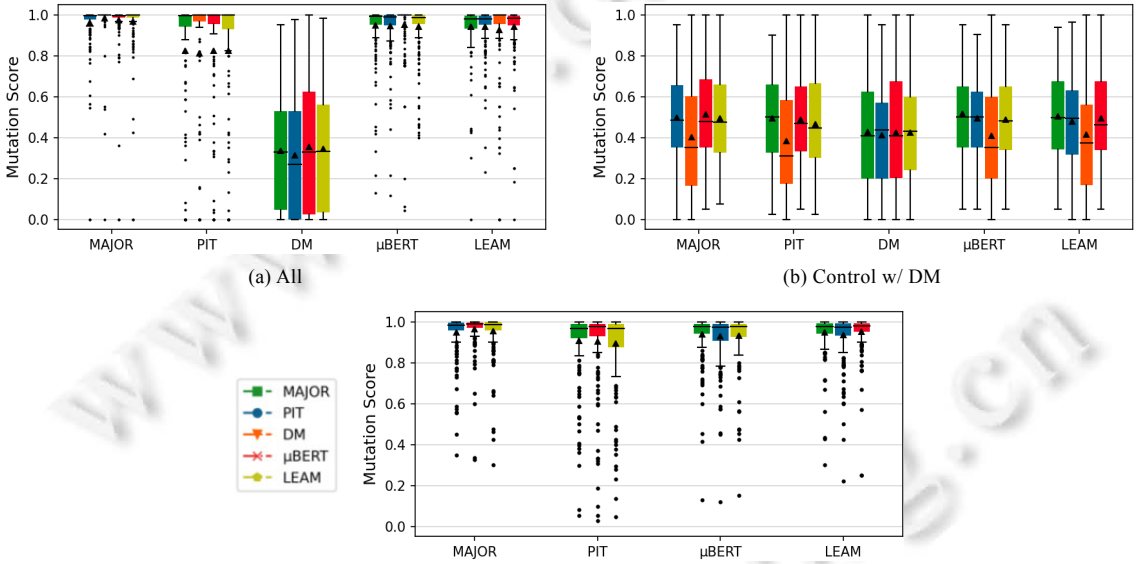
代表性(representativeness). 用于度量不同变异技术生成的变异体集合之间的语义代表能力. 具体地, 首先, 基于一种变异技术生成的变异体构造“最小化的测试套件”; 然后, 在另一技术生成的变异体集上执行该“最小化的测试套件”并计算变异分数.

- 实验流程

借鉴现有文献的做法, 评估不同变异技术之间的代表性, 从而分析不同变异技术生成变异体集合的替补关系: 1) 在每个实验对象上, 基于不同的变异技术的分析集构建“最小化的测试套件”; 2) 在另一技术的分析集上执行该测试套件, 并计算该测试套件的变异分数.

- 结果分析

图 6 展示了不同变异技术之间的相互代表性, 横坐标表示 5 种变异技术, 纵坐标表示变异分数. 例如:横轴为 MAJOR 与其第 1 竖列 PIT, 表示基于 MAJOR 构造的 Minimal Test Suite 在 PIT 的分析集上取得了 0.959 的变异分数.



(a) All (b) Control w/ DM

(c) Control w/o DM

图 6 不同变异技术之间的相互代表能力

在不控制变异体数量时, MAJOR 对于 PIT、DM、μBERT 和 LEAM 的语义代表能力为 0.959、0.983、0.973 和 0.969; PIT 对于 MAJOR、DM、μBERT 和 LEAM 的语义代表能力为 0.825、0.810、0.823 和 0.825; DM 对于 MAJOR、PIT、μBERT 和 LEAM 的语义代表能力为 0.336、0.312、0.355 和 0.344; μBERT 对于 MAJOR、PIT、DM 和 LEAM 的语义代表能力为 0.947、0.944、0.951 和 0.943; LEAM 对于 MAJOR、PIT、DM 和μBERT 的语义代表能力为 0.941、0.932、0.926 和 0.942.

在 Control w/o DM 条件下, MAJOR 对于 PIT、μBERT 和 LEAM 的语义代表能力为 0.949、0.966、0.953 和 0.907; PIT 对于 MAJOR、μBERT 和 LEAM 的语义代表能力为 0.907、0.906 和 0.895; μBERT 对于 MAJOR、PITT 和 LEAM 的语义代表能力为 0.932、0.921 和 0.927; LEAM 对于 MAJOR、PIT 和μBERT 的语义代表能力为 0.948、0.936 和 0.952. 总体上, MAJOR 生成的变异体具有最高的语义代表能力, 基于 MAJOR 构造的测试套件在其余变异技术对应的变异体上均能取得大于 0.950 的变异分数. μBERT (0.943–0.951)和 LEAM

(0.929–0.942)的语义代表能力相近.

注意到: 不同技术之间的变异分数均值小于 1.0, 即基于一种技术构造的测试套件不能完全杀死另一技术生成的变异体集, 这表明变异技术之间不存在严格的取代关系. 为了进一步分析基于学习的变异技术与基于规则的变异技术之间代表能力的互补关系, 我们从变异算子的角度, 分析基于学习的变异技术对于特定变异算子的代表能力.

表 6 和表 7 分别展示了两种基于学习的变异技术对于 MAJOR 和 PIT 变异算子的代表能力, 行 Original 表示 MAJOR/PIT 变异算子的原始比重, μBERT/LEAM 分别表示在执行基于μBERT/LEAM 构造测试套件后, 存活变异体中变异算子的分布. 由于 PIT 包含较多的变异算子, 因此我们只统计了原始比重最高的 8 种, 分别为 UOI (unary operator insertion)、CRCR (constant replacement mutator)、ROR (relational operator replacement mutator)、ABS (negation mutator)、RC (remove conditionals)、AOR (arithmetic operator replacement mutator)、NVMC (non void method calls)和 IC (inline constant).

表 6 基于学习的变异技术对于 MAJOR 的代表能力(%)

变异技术	变异算子								
	AOR	COR	EVR	LOR	LVR	ORU	ROR	SOR	STD
Original	20.2	10.9	8.9	0.4	23.7	0.6	22.1	0.1	13.1
μBERT	7.2 (↓)	9.8	14.5 (↑)	0.4	25	0.8	18.8 (↓)	0.2	23.3 (↑)
LEAM	14.1 (↓)	8.0 (↓)	10.1 (↑)	1.8	35.7 (↑)	1.1	20.9	0	8.3 (↓)

表 7 基于学习的变异技术对于 PIT 的代表能力(%)

变异技术	变异算子								
	UOI	CRCR	ROR	ABS	RC	AOR	NVMC	IC	Other
Original	25.8	16.6	12.6	6.4	6.4	5.4	5.5	3.4	18.4
μBERT	18.6 (↓)	17.4	14.5	4.8	9.3 (↑)	2.4 (↓)	7.5 (↑)	3.6	21.9
LEAM	23.4 (↓)	18.5	11.9	6.0	7.2	3.9	5.9	3.7	19.5

μBERT 对于 MAJOR-AOR、MAJOR-ROR、PIT-UOI、PIT-AOR 的代表能力较强, 分别降低了其 8%、2.3%、7.2%和 3.0%的比重. 这表明: μBERT 对程序中单个 Token 进行 Mask 的策略可以很好地模拟现有 AOR (arithmetic operator replacement)、ROR (relational operator replacement)和 UOI (unary operator insertion)算子, 即对运算符替换操作具有较优的代表能力. 我们认为, 这是因为μBERT 使用的 CodeBERT 大语言模型(large language model)具有优越的语义和语法理解能力, 能够结合程序的上下文信息进行变异. 然而, μBERT 对于 EVR (expression value replacement)、STD (statement deletion)、RC (remove conditional)的代表能力较弱, 使其在存活变异体中比原始比重高出 5.6%、10.2%和 2.9%的比重. 我们认为, μBERT 单一 Token 替换的变异策略对于涉及多个 Token 的“粗粒度”变异算子不具备较高的代表能力.

LEAM 对于 MAJOR-AOR、MAJOR-COR、MAJOR-STD 和 PIT-UOI 的代表能力较强, 分别降低了其 6.1%、2.9%、4.8%和 2.4%的比重. 这是因为 LEAM 通过学习历史提交, 能够学习到常规的变异策略, 包括“粗粒度”的语句删除变异算子. 利用上下文信息, LEAM 可以将学习到的变异模式应用到程序中潜在的缺陷位置. 注意到: LEAM 无法很好地代表 EVR 与 LVR, 这表明 LEAM 对于数值替换变异的代表能力较弱.

- RQ2 总结

综上, 传统的变异技术 MAJOR 生成的变异体具有最强的语义代表能力, 其对应的测试套件能够杀死其余技术分析集中超过 95%的变异体. 这表明, 传统的变异算子仍具有较强的代表能力. 同时, 我们发现: 基于学习的变异技术由于具备上下文语义理解能力, 可以增强现有部分变异算子的有效性. 因此, 我们认为, 现有的两类变异技术的语义代表能力可以相互补充.

5.3 RQ3

- 评估指标

语义相似性(semantics similarity). 用于评估两个程序行为之间的相似性. 语法变动可能会导致变异体的

行为功能与原程序发生不同. 根据现有文献[43]的做法, 基于测试用例的执行结果(通过或未通过)来表征程序语义. 具体地, 基于缺陷定位领域的 Ochiai 公式量化程序之间的语义相似性. 如公式(2)所示, 给定原程序 $P1$、$P2$ 以及两者对应的执行失败的测试集 FTS_1、FTS_2, 根据公式计算即可得到两者的语义相似性:

$$\text{Ochiai}(P1,P2)=\frac{|fTS_1\cap fTS_2|}{\sqrt{|fTS_1|\cdot|fTS_2|}} \tag{2}$$

故障耦合(fault coupling). 用于评估变异体与真实故障行为之间的耦合关系. 给定原程序 P、变异体 m 与测试套件 T, 变异体 m 与故障 f 具有耦合关系, 当且仅当:

1) 存在测试用例 $t\in T$, t 能够杀死变异体 m;

2) 对于任何杀死变异体 m 的测试用例 t, 均能够检测故障 f.

- 实验流程

首先, 使用语义相似性比较不同变异技术生成的变异体与真实缺陷之间的语义相似性. 具体地, 在每个研究对象上计算生成变异体与真实缺陷的语义相似性.

为了探究变异体与真实缺陷的耦合关系, 实验统计了与每一种变异技术耦合的真实缺陷集合, 并分析不同技术对应的缺陷集合之间的交并关系.

- 结果分析

图 7 展示了不同变异技术生成的变异体与真实缺陷的最大语义相似性, 横坐标罗列了 5 种变异技术的名称, 纵坐标表示每个实验对象上在变异体与缺陷的最大语义相似性. 平均来看, MAJOR、PIT、DM、μBERT 和 LEAM 与缺陷的最大相似性分别为 0.766、0.651、0.148、0.731 和 0.664. 除了 DM 之外, 其余 4 种变异技术生成的变异体集中存在变异体可以模拟缺陷的语义, 其中, MAJOR (0.766)和μBERT (0.731)生成的变异体与真实缺陷在语义上具有较高的相似性. 在包括 Closure 数据集时, MAJOR、PIT、μBERT 和 LEAM 与缺陷的最大语义相似性分别 0.755、0.641、0.722 和 0.650.

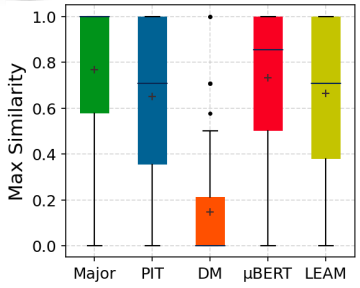


图 7 变异体与缺陷的最大语义相似性

实验发现: 虽然 LEAM 理论上可以生成形式更为复杂的高阶变异体, 但是其生成的变异体在与缺陷的语义相似性上不如 MAJOR 和μBERT. 与 LEAM 不同的是, MAJOR 和μBERT 生成的变异体均可以看作是一阶变异体(其中, MAJOR 基于某种变异算子进行变异、μBERT 则通过调用 CodeBERT 对单个 Token 进行替换). 注意到: 多数情况下, 真实缺陷会涉及若干行代码改动, 其与原程序的语法差异往往要大于一阶变异体. 因此, 我们认为: 与真实缺陷语义相近的变异体不一定在语法形态上与之相近, 使用 MAJOR 和μBERT 生成的一阶变异体仍然能有效地模拟真实缺陷的语义行为.

表 8 展示了与不同变异技术生成变异体具有耦合关系的缺陷情况. 在不考虑 Clousre/考虑 Closure、使用全部变异体的条件下, MAJOR、PIT、DM、μBERT 和 LEAM 生成的变异体分别与 173/236、146/203、16/–、157/216 和 142/195 个真实缺陷相互耦合, 故障耦合率分别为 0.765/0.771、0.646/0.663、0.071/–、0.695/0.706 和 0.628/0.637. 在不考虑 Clousre、Control w/ DM 的条件下, MAJOR、PIT、DM、μBERT 和 LEAM 的平均故障耦合率为 0.128、0.106、0.061、0.105 与 0.095. 在考虑 Closure、Control w/o DM 的条件下, MAJOR、PIT、μBERT 和 LEAM 的平均故障耦合率为 0.598、0.508、0.541 和 0.551. 实验表明: MAJOR 具有最高的故障耦合

率(0.765), μBERT 次之(0.695), PIT 和 LEAM 的表现相近(0.646 与 0.628); 在控制了变异体数量后, MAJOR 生成的变异体仍然具有最高的耦合率(0.598), 而 LEAM(0.551)则仅次于 MAJOR. 这表明: MAJOR 生成变异体的质量最高;而控制变异体数量的情况下, LEAM 生成变异体的质量也高于 PIT、DM、μBERT.

表 8 不同技术的故障耦合率

变异技术	故障耦合率		
	All	Control w/ DM	Control w/o DM
MAJOR	0.765 (0.771)	0.128	0.598
PIT	0.646 (0.663)	0.106	0.508
DM	0.071 (–)	0.061	–
μBERT	0.695 (0.706)	0.105	0.541
LEAM	0.628 (0.637)	0.095	0.551

图 8 以韦恩图的形式展示了与不同变异技术生成的变异体具有耦合关系的缺陷集合情况. 注意到: DM 效果最差, 无法与绝大多数缺陷耦合. 因此, 我们忽略 DM, 讨论图 8(b). 在 306 个研究对象上, 83.7%的缺陷(256/306)能与至少 1 种技术生成的变异体具有耦合关系, 即大多数缺陷能与现有技术生成的变异体耦合. 81.0%的缺陷(248/306)能与 MAJOR 与 PIT 基于规则的变异技术耦合, 78.8%的缺陷(241/306)能与μBERT 和 LEAM 基于学习的变异技术耦合. 两类不同思想指导的变异技术具有相近的故障耦合率, 这表明这两类技术生成的变异体均可以模拟真实缺陷的异常行为, 指导开发者提升测试套件的有效性.

(a) 不包括 Closure (b) 包括 Closure

图 8 不同变异技术生成变异体的缺陷耦合情况

实验发现: 存在 6 个缺陷仅与 MAJOR 耦合(Closure-17、Closure-111、Mockito-7、Mockito-16、Mockito-17 与 Time-10), 存在 4 个缺陷仅与 PIT 耦合(Closure-21、Closure-22、Mockito-10 与 Math-88), 存在 5 个缺陷仅与μBERT 耦合(Closure-16、Math-23、Math-30、Math-87 与 Mockito-30), 存在 3 个缺陷仅与 LEAM 耦合(Chart-8、Closure-22、Math-34). 存在 15 个缺陷仅与 MAJOR 与 PIT 基于规则的变异技术耦合, 存在 8 个缺陷仅与μBERT 与 LEAM 基于学习的变异技术耦合.

本文的前两位作者审阅了实验中涉及的真实缺陷及其耦合变异体的源代码, 讨论并分析了基于规则和基于学习的两类变异技术生成变异体的特性. 总的来说, 基于规则的变异技术对特定代码元素的变异能力更为稳定, 并且精心设计的变异算子通常具有较强的程序语义修改能力; 基于学习的变异技术具有更强的上下文语义感知能力, 能够生成更符合语境的变异体, 且生成的变异体在语法形态上更为多样. 基于我们的观测与分析, 我们希望未来的变异测试工作在设计和选择变异体时可以得到参考和启发.

对于基于规则的变异技术, 我们的分析如下.

- 代码元素变异的稳定性

基于规则的变异技术机械地匹配语法变换规则进行变异操作, 因此, 对于特定的代码元素基于规则的变

异技术能够保证更为稳定的变异覆盖率.

如图 9(a)所示, 在缺陷 Mocktio-7 中, 开发者遗漏了语句 registerTypeParametersOn(new TypeVariable[·]{typeVariable}). MAJOR 技术通过 STD 变异算子复现了该缺陷. 值得注意的是: 审阅过程中, 我们发现 LEAM 同样具有语句删除能力, 然而其黑盒的推理并未在该位置进行语句删除. 因此我们认为, 基于语法规则的变异算子对于特定代码元素具有更稳健的变异能力.

缺陷ID: **Mockito-7** 缺陷所在类: **GenericMetadataSupport.Java**

```
@@ -376,7 +376,6 @@ public abstract class GenericMetadataSupport {
            for (Type type : typeVariable.getBounds()) {
                registerTypeVariablesOn(type);
            }
-           registerTypeParametersOn(new TypeVariable[] { typeVariable });
            registerTypeVariablesOn(getActualTypeArgumentFor(typeVariable));
        }
```

MAJOR

STD: registerTypeParametersOn(new TypeVariable[]{typeVariable}) |==> <NO-OP>

(a) 缺陷 Mockito-7

缺陷ID: **Chart-26** 缺陷所在类: **Axis.Java**

```
@@ -1189,13 +1189,11 @@ public abstract class Axis implements Cloneable, Serializable {
        }
        if (plotState != null && hotspot != null) {
            ChartRenderingInfo owner = plotState.getOwner();
-           if (owner != null) {
                EntityCollection entities = owner.getEntityCollection();
                if (entities != null) {
                    entities.add(new AxisLabelEntity(this, hotspot,
                            this.labelToolTip, this.labelURL));
                }
-           }
        }
```

PIT

COR: owner != null |==> true

(b) 缺陷 Chart-26

缺陷ID: **Math-30** 缺陷所在类: **MannWhitneyUTest.Java**

```
@@ -170,7 +170,7 @@ public class MannWhitneyUTest {
                                    final int n2)
        throws ConvergenceException, MaxCountExceededException {
-       final double n1n2prod = n1 * n2;
+       final int n1n2prod = n1 * n2;
```

μBERT

double |==> int

(c) 缺陷 Math-30

缺陷ID: **Math-34** 缺陷所在类: **ListPopulation.Java**

```
@@ -206,6 +206,6 @@ public abstract class ListPopulation implements Population {
    public Iterator<Chromosome> iterator() {
-       return getChromosomes().iterator();
+       return chromosomes.iterator();
```

LEAM

- return this.chromosomes;

+ return Collections.unmodifiableList(chromosomes);

(d) 缺陷 Math-34

图 9 耦合缺陷分析

- 程序语义变动的有效性

经过长期实践探索, 部分变异算子已证明具有较高的程序语义变换能力. 例如: 条件运算符替换算子(ROR)可以将条件判断替换成确定的布尔值, 改变程序的控制流, 从而影响程序语义. 如图 9(b)所示, 在缺陷 Chart-26 中, PIT 将条件判断 *owner*!=*null* 替换为 true, 影响了程序的语义并帮助揭示真实缺陷的异常行为.

对于基于学习的变异技术,我们的分析如下.

- 上下文语义理解的敏感性

基于学习的变异技术利用深度学习模型进行变异, 因此, 基于学习的变异技术能够敏感地捕获程序上下文语义信息进行变异.

如图 9(c)所示, 在缺陷 Math-30 中, 上下文中的函数签名 *private double calculateAsymptoticPValue*(*final double Umin*,*final int n*1,*final int n*2)表明, 形参 *n*1 与 *n*2 是 *int* 类型. 因此在变异过程中, μBERT 基于 CodeBERT 推理将变量 *n*1*n*2*prod* 的类型从 *double* 变异为 *int*, 从而复现缺陷.

- 语法变换的多样性

基于学习的变异技术以海量的训练资料作为支持, 能够学习训练数据中出现的复杂多样的变异模式. 如图 9(d)所示, 在缺陷 Math-34 中, 直接访问 *chromosomes* 的 *iterator* 会导致程序异常. LEAM 通过变异 *getChromosomes* 方法, 在方法内部将复杂的方法调用 *Collections.unmodifiableList*(*chromosomes*)变异为 *this*.*chromosomes*, 从而复现缺陷.

- RQ3 总结

综上, 我们发现: 传统的变异技术对于特定代码元素的变异更为稳定, 且其集成的变异算子大多经过精心设计, 具有较强的程序语义变动能力; 而基于学习的变异技术具有更强的上下文理解能力, 能够结合语境生成更为自然的变异体, 且其生成的变异体具有更加多样的语法变换形式, 可以表征多样的缺陷模式. 因此, 我们认为: 基于规则和基于学习两类变异机理具有正交的缺陷表征能力, 未来的研究可以结合两类变异机理生成语义更为丰富的变异体.

6 讨 论

6.1 变异技术可靠性

变异体生成工具作为一类面向软件开发者的测试软件, 需要满足软件的可靠性, 即需要持续地提供程序变异功能. 在第 4.4 节中, 由于实验环境、技术实现等因素, 所选用的变异技术无法在 Defects4j v1.2.0 的全部实验对象上进行变异与评估. 为了更为全面地评价不同变异技术的能力, 我们进一步分析变异技术能够成功变异的实验对象数量, 这里称为变异技术可靠性.

表 9 展示了不同变异技术能够成功生成变异体的实验对象数量(称为分析缺陷数), 其中, ∩表示 5 种变异技术的分析缺陷数的交集. 总体上, MAJOR、PIT、DM、μBERT 和 LEAM 对应的分析缺陷数分别为 395、351、254、351、357. 实验发现: MAJOR 具有最佳的可靠性, 能够成功变异 Defects4j v1.2.0 中全体实验对象; PIT、μBERT 和 LEAM 的变异能力相近, 能够变异 351–357 个实验对象; DM 由于代码实现问题, 无法变异 Closure 项目, 因此仅能变异 254 个实验对象, 其可靠性最差.

表 9 不同变异技术的分析缺陷数

技术名称	项目名称					
	Chart	Closure	Lang	Math	Mockito	Time
MAJOR	26	133	65	106	38	27
PIT	26	101	65	97	35	27
DM	26	–	64	102	38	24
μBERT	26	109	57	98	38	23
LEAM	26	112	54	100	38	27
∩	26	80	52	90	35	23

注意到 MAJOR 作为 Defects4j v1.2.0 内嵌的变异测试引擎, 因此其天然地在 Defects4J 上具有较好的兼容

性与可靠性. 然而, 通过审阅其余变异技术(即 PIT、DM、μBERT 和 LEAM)的开源实现, 我们发现现有变异技术在代码实现上均存在一定的缺陷. 因此, 我们建议变异技术的研究与开发人员需要更加关注变异测试软件的可靠性, 以持续地为工业实践与学术研究提供技术保证.

6.2 变异测试开销

为了充分评估不同变异技术生成变异体的效率, 我们记录了它们在每个实验对象上生成变异体的时间. 注意: 对于同一实验对象, 不同变异技术生成的变异体数量往往不同. 为了比较的公平性, 我们基于采用的 306 个实验对象上计算了每种变异技术生成单个变异体的平均时间. 具体地, MAJOR、PIT、DM、μBERT 和 LEAM 的平均生成成本为 0.063 s、0.007 s、0.812 s、1.120 s 和 0.920 s. 结果表明: MAJOR 和 PIT 这两种基于规则的变异技术具有更高的生成效率(小于 0.01 s), 而 DM、μBERT 和 LEAM 这 3 种基于学习的变异技术在生成变异体时需要相对更高的时间开销(0.812 s–1.120 s). 这一结果表明: 基于学习的变异技术在推理和生成过程中需要更高的时间开销, 而基于规则的变异技术开销较低.

6.3 有效性威胁

本文的内部有效性威胁主要来自技术实现以及实验实施. 为了缓解威胁, 实验使用的变异技术均采用其官方或开源实现, 并在第 4.6 节所描述的系统环境下进行调试; 此外, 本文的前两位作者细致地检查了实验代码, 确保逻辑正确、实验数据完整.

本文的构建威胁主要来自实验采用的评估指标. 为了缓解威胁, 实验采用先前工作广泛使用的指标.

本文的外部有效性威胁主要来自实验对象程序及其测试套件. 为了缓解威胁, 实验采用软件测试领域广泛使用的基准数据集 Defects4J(包括其缺陷和测试套件). 值得注意的是: Defects4J 是基于 Java 的开源测试框架与缺陷数据集, 尽管其在软件工程研究中具有极为广泛的应用, 但是基于 Defects4J 的变异测试实验仍然不能够轻易地将实验结论推广到其他程序语言, 甚至无法全面囊括基于 Java 的变异测试各种应用场景, 这也是本文研究的局限性. 在未来, 我们会拓展实验场景, 在更多的程序语言(例如 C/C++与 Python)上比对不同机理的变异技术的测试效果, 从而得到更为一般的结论. 此外, 由于实验中仅用 Defects4J 唯一一个开源基准数据集, 实验结论存在一定的过拟合风险, 未来工作应该在更为广泛的数据集上进行比对研究.

7 对未来工作的可能影响

根据本文实验, 我们发现基于规则和基于学习的变异技术均能够有效地支持变异测试, 两者生成的变异体在语义代表性上具有一定的正交性, 可以相互补充, 从而表征更多的真实缺陷. 本文的实证研究对于未来工作可能有以下 3 个方面的启发.

1) 编译通过率: 现有的 3 种 State-of-the-art 的基于学习的变异技术 DM、μBERT 和 LEAM 生成变异体的编译通过率分别为 41.1%、26.9%和 36.4%, 数值上远低于基于规则的变异技术 MAJOR (96.4%)和 PIT (98.9%). 未来工作应该关注如何提升基于学习的变异技术的编译通过率, 提高其生成变异体的可用性.
2) 变异体生成数量: 实验中, MAJOR、PIT、DM、μBERT 和 LEAM 在每个实验对象上生成的变异体的平均数为 360.1、1 078.3、5.0、601.5 和 299.4. 除了 DM 之外的其余变异体生成技术仍然生成了数量较多的变异体, 导致实验的执行开销较高. 未来工作可以关注如何更有效地预测和选择变异体生成位置, 或者如何结合程序上下文选择更符合语境的变异体, 从而降低变异测试的计算开销.
3) 高阶变异体: 有价值的高阶变异体可以有效耦合部分细微缺陷, 降低变异测试开销, 提升变异测试的有效性[17]. 现有技术 DM 和 LEAM 虽然从理论上可以生成较为复杂的高阶变异体, 但实验结果显示, 其变异测试性能仍然具有提升空间. 未来工作可以探索如何基于语法规则以及深度学习技术生成更有价值的高阶变异体, 推动高阶变异测试的发展.

8 总 结

我们对基于规则和基于学习的变异技术展开全面地比较研究, 并探究这两类不同机理的变异技术生成变异体的语义差异性. 具体地, 我们有如下发现.

- 在变异测试有效性上, 两类变异技术均能有效地支持变异测试, 但在性能上存在一定的差异. 其中, MAJOR 对于测试套件的提升效果最好, 其对应的测试套件能检测 85.4%的真实缺陷; PIT、μBERT 和 LEAM 分别能检测 72.9%、80.4%和 75.8%的真实缺陷.
- 在语义替补性上, MAJOR 生成的变异体具有最强的语义代表能力, 基于 MAJOR 构造的测试套件能杀死其余技术分析集中超过 95%的变异体; 同时发现, 基于学习的变异技术具备上下文理解能力, 可以增强现有部分变异算子的语义代表能力. 因此我们认为, 现有的两类变异技术的语义代表能力可以相互补充.
- 在缺陷表征的独特性上, 我们认为: 基于规则的变异技术具有更稳健的代码元素变异能力, 且部分变异算子具有较高的缺陷揭示能力; 基于学习的变异技术则具有更敏感的上下文语义理解能力, 且能生成形式更为复杂的变异体, 补充现有基于规则的变异技术的机械性. 因此我们认为, 未来研究可以结合两种变异机理用以生成更为多样的变异体.

本文在大规模数据集上分析基于规则与基于学习两类不同机理的变异技术在变异体语义上的差异.在未来, 我们计划将基于语法规则变换的变异模式与基于深度学习推理的变异模式结合, 提出一种稳健的、基于混合模式的、能够根据上下文语义动态推理变异模式的变异技术, 以更好地支持变异测试研究. 我们认为: 深度学习技术可以借鉴程序的语法规则, 充分学习程序分析相关的领域知识, 例如类型系统(type system),从而提高模型生成变异体的语法正确性和可编译率. 此外, 大语言模型的涌现, 为基于学习的变异技术提供了新的内核, 可以辅助生成更高质量的变异. 我们认为: 这一方向对于改进现有变异技术具有十分重要的意义, 有助于克服当前技术的局限性, 提高变异测试的效率和效果.

References:

[1] Ammann P, Offutt J. Introduction to Software Testing. Cambridge University Press, 2016.

[2] Moore I. Jester-a JUnit test tester. In: Proc. of the 2nd XP. 2001. 84−87.

[3] Ma YS, Offutt J, Kwon YR. MuJava: An automated class mutation system. Software Testing, Verification and Reliability, 2005, 15(2): 97−133.

[4] Irvine SA, Pavlinic T, Trigg L, *et al*. Jumble Java byte code to measure the effectiveness of unit tests. In: Proc. of the Testing: Academic and Industrial Conf. on Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007. 169−175.

[5] Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering. 2009. 297−298.

[6] Just R. The major mutation framework: Efficient and scalable mutation analysis for Java. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. 2014. 433−436.

[7] Coles H, Laurent T, Henard C, *et al*. PIT: A practical mutation testing tool for Java. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. 2016. 449−452.

[8] Papadakis M, Kintis M, Zhang J, *et al*. Mutation testing advances: An analysis and survey. Advances in Computers, 2019, 112: 275−378.

[9] Papadakis M, Le Traon Y. Metallaxis-FL: Mutation-based fault localization. Software Testing, Verification and Reliability, 2015, 25(5−7): 605−628.

[10] Zhang J, Wang Z, Zhang L, *et al*. Predictive mutation testing. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. 2016. 342−353.

[11] Ghanbari A, Benton S, Zhang L. Practical program repair via bytecode mutation. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2019. 19−30.

[12] Tufano M, Watson C, Bavota G, *et al*. Learning how to mutate source code from bug-fixes. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2019. 301−312.

[13] Degiovanni R, Papadakis M. μBERT: Mutation testing using pre-trained language models. In: Proc. of the 2022 IEEE Int'l Conf. on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2022. 160−169.

[14] Tian Z, Chen J, Zhu Q, *et al*. Learning to construct better mutation faults. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2022. 1−13.

[15] Amalfitano D, Paiva ACR, Inquel A, *et al*. How do Java mutation tools differ? Communications of the ACM, 2022, 65(12): 74−89.

[16] Wang Z, Yan M, Liu S, Chen JJ, Zhang DD, Wu Z, Chen X. Survey on testing of deep neural networks. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1255−1275 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.jos.005951]

[17] Jia Y, Harman M. Higher order mutation testing. Information and Software Technology, 2009, 51(10): 1379−1393.

[18] Gopinath R, Jensen C, Groce A. The theory of composite faults. In: Proc. of the 2017 IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). IEEE, 2017. 47−57.

[19] Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. In: Mutation Testing for the New Century. 2001. 34−44.

[20] Ammann P, Delamaro ME, Offutt J. Establishing theoretical minimal sets of mutants. In: Proc. of the 7th IEEE Int'l Conf. on Software Testing, Verification and Validation. IEEE, 2014. 21−30.

[21] Zhang L, Marinov D, Zhang L, *et al*. Regression mutation testing. In: Proc. of the 2012 Int'l Symp. on Software Testing and Analysis. 2012. 331−341.

[22] Wang B, Xiong Y, Shi Y, *et al*. Faster mutation analysis via equivalence modulo states. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2017. 295−306.

[23] Yao YW, Jiang SJ, Bo LL. Survey of mutation-based fault location. Computer Engineering and Applications, 2019, 55(20): 1−12 (in Chinese with English abstract).

[24] Zhang J, Zhu M, Hao D, *et al*. An empirical study on the scalability of selective mutation testing. In: Proc. of the 25th IEEE Int'l Symp. on Software Reliability Engineering. IEEE, 2014. 277−287.

[25] Beller M, Wong CP, Bader J, *et al*. What it would take to use mutation testing in industry—A study at Facebook. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021. 268−277.

[26] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. Computer, 1978, 11(4): 34−41.

[27] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans. on Software Engineering, 2010, 37(5): 649−678.

[28] Offutt AJ. Investigations of the software testing coupling effect. ACM Trans. on Software Engineering and Methodology (TOSEM), 1992, 1(1): 5−20.

[29] Offutt A. The coupling effect: Fact or fiction. ACM SIGSOFT Software Engineering Notes, 1989, 14(8): 131−140.

[30] Just R, Jalali D, Inozemtseva L, *et al*. Are mutants a valid substitute for real faults in software testing? In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. 2014. 654−665.

[31] Gopinath R, Jensen C, Groce A. Mutations: How close are they to real faults? In: Proc. of the 25th IEEE Int'l Symp. on Software Reliability Engineering. IEEE, 2014. 189−200.

[32] Brown DB, Vaughn M, Liblit B, *et al*. The care and feeding of wild-caught mutants. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. 2017. 511−522.

[33] Khanfir A, Koyuncu A, Papadakis M, *et al*. IBIR: Bug-report-driven fault injection. ACM Trans. on Software Engineering and Methodology, 2023, 32(2): 1−31.

[34] Liu K, Koyuncu A, Kim D, *et al*. TBar: Revisiting template-based automated program repair. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. 2019. 31−42.

[35] Feng Z, Guo D, Tang D, *et al*. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the Findings of the Association for Computational Linguistics (EMNLP 2020). 2020. 1536−1547.

[36] Chekam TT, Papadakis M, Le Traon Y, *et al*. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). IEEE, 2017. 597−608.

[37] Papadakis M, Shin D, Yoo S, *et al*. Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In: Proc. of the 40th Int'l Conf. on Software Engineering. 2018. 537−548.

[38] Chen YT, Gopinath R, Tadakamalla A, *et al*. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2020. 237−249.

[39] Li N, Praphamontripong U, Offutt J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Proc. of the 2009 Int'l Conf. on Software Testing, Verification, and Validation Workshops. IEEE, 2009. 220−229.

[40] Gopinath R, Jensen C, Groce A. Code coverage for suite evaluation by developers. In: Proc. of the 36th Int'l Conf. on Software Engineering. 2014. 72−82.

[41] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proc. of the 36th Int'l Conf. on Software Engineering. 2014. 435−445.

[42] Kintis M, Papadakis M, Papadopoulos A, *et al*. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. Empirical Software Engineering, 2018, 23: 2426−2463.

[43] Ojdanic M, Garg A, Khanfir A, *et al*. Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. IEEE Trans. on Software Engineering, 2023, 49(7): 3922−3938.

[44] Ojdanic M, Khanfir A, Garg A, *et al*. On comparing mutation testing tools through learning-based mutant selection. In: Proc. of the 4th ACM/IEEE Int'l Conf. on Automation of Software Test (AST 2023). 2023.

[45] Tufano M, Kimko J, Wang S, *et al*. Deepmutation: A neural mutation tool. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering: Companion Proceedings. 2020. 29−32.

[46] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. 2014. 437−440.

[47] Papadakis M, Henard C, Harman M, *et al*. Threats to the validity of mutation-based test assessment. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. 2016. 354−365.

[48] Wilcoxon F. Individual comparisons of grouped data by ranking methods. Journal of Economic Entomology, 1946, 39(2): 269−270.

[49] Vargha A, Delaney HD. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics, 2000, 25(2): 101−132.

[50] Offutt J. A mutation carol: Past, present and future. Information and Software Technology, 2011, 53(10): 1098−1107.

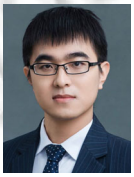
[51] Papadakis M, Malevris N. Automatic mutation test case generation via dynamic symbolic execution. In: Proc. of the 21st IEEE Int'l Symp. on Software Reliability Engineering. IEEE, 2010. 121−130.

附中文参考文献:

[16] 王赞, 闫明, 刘爽, 陈俊洁, 张栋迪, 吴卓, 陈翔. 深度神经网络测试研究综述. 软件学报, 2020, 31(5): 1255−1275. http://www.jos.org.cn/1000-9825/5951.htm [doi: 10.13328/j.cnki.jos.005951]

[23] 姚毅文, 姜淑娟, 薄莉莉. 基于变异测试的错误定位研究进展. 计算机工程与应用, 2019, 55(20): 1−12.

贡志豪(2000－), 男, 博士生, CCF 学生会员, 主要研究领域为软件工程.



陈俊洁(1992－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为软件分析与测试.

陈逸洲(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为软件测试, 区块链安全和深度学习.

郝丹(1979－), 女, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件测试.