利用预训练语言模型实现高效变异测试

Ahmed Khanfir、RenzoDegiovanni、Mike Papadakis 和 Yves Le Traon，SnT，卢森堡大学，卢森堡

摘要—变异测试是一种成熟的基于故障的测试技术。它通过向待测程序中注入故障，并要求开发者编写能够揭示这些故障的测试来实现。此类测试有望揭示大量故障——即与注入故障耦合的故障——因此被认为十分重要。为此，变异测试应注入既“自然”（即开发者容易理解）又强大（揭示故障的可能性高）的故障。为实现这一目标，我们提出使用预训练生成式语言模型（即 CodeBERT），它能够生成类似开发者编写的代码，其行为与目标代码相似但并不完全相同。这意味着模型能够注入自然故障，从而为开展变异测试提供机会。我们通过实现 μBERT来实践这一构想；这是一种使用 CodeBert 执行变异测试的技术，并使用 689 个故障程序版本对其进行了实证评估。结果表明，μBERT 的故障揭示能力高于最先进的变异测试工具 PiTest，生成的测试的故障检测潜力最高比 PiTest 高 17%。此外，我们发现 μBERT 可以作为 PiTest 的补充，能够检测出 PiTest 漏掉的 47 个错误；与此同时，PiTest 也能发现 μBERT 漏掉的 13 个错误。

索引词—故障 注入、变异测试、预训练语言 模型

arXiv:2301.03543v1 [cs.SE] 9 Jan 2023

1 引言

变异测试旨在通过简单的语法转换来播种故障 [19]。这些转换也称为变异算子，通常依据目标编程语言的语法所制定的规则构造 [8]，，例如将一个算术运算符替换为另一个运算符，比如将 + 替换为 -。遗憾的是，这类技术会生成许多变异体（播种的故障），其中许多都“不自然”，即不符合开发者的编码方式，因此开发者会认为它们不切实际 [11]。与此同时，基于语法的故障播种无法捕捉其所作用的代码片段的语义，从而导致大量琐碎或实用价值低的故障 [46]。

为了解决上述问题，我们提出利用大规模代码来生成自然的变异体。因此，我们旨在引入符合程序员所遵循的隐含规则、规范和编码惯例的修改，并借助预训练语言模型的能力，捕捉代码及其写作方式的内在分布；这种分布是模型通过在大规模代码上进行预训练而学得的。

为此，我们依赖 CodeBERT [22]，这是一种自然语言-编程语言双模态语言模型，训练数据超过 640 万个程序。更确切地说，我们使用其掩码语言建模（MLM）功能：给定一个包含掩码令牌的代码序列，该功能会预测可替换该令牌的候选项，使其与序列上下文的匹配度最高。这一点很重要，因为这些预测结果不像传统变异测试那样遵循预先设定的固定模式，而是会调整以尽可能适应目标代码。例如，给定一个序列 int a = 1;，我们将其掩码版本 int a =<mask>; 作为输入，然后 CodeBERT 默认会按照可能性得分从高到低给出 5 个预测结果：0、1、b、2 和 10。由于这些词元最有可能与代码上下文相匹配，我们认为，用这些预测结果替换掩码令牌会生成“自然”的变异体。

具体而言，我们提出了 μBERT，这是一种变异测试方法，使用预训练语言模型（CodeBERT）通过掩码并替换词元来生成变异体，旨在形成自然的变异体。μBERT 遍历程序语句并修改其中的词元。具体来说，μBERT 按以下步骤进行：（1）一次选择并掩码一个词元；（2）将掩码后的序列输入 CodeBERT，并获得预测结果；（3）用预测的词元替换被掩码的词元，从而创建变异体；（4）丢弃无法编译、重复以及等价的变异体（与原始代码在语法上相同的变异体）。

近期研究 [32] 表明，一些真实故障只有通过使用复杂模式才能捕获，即需要多于一次词元变异的模式。为应对这类情况，μBERT 配备了加性变异，即添加代码的变异（而非删除或修改代码）。例如，考虑一个布尔表达式e_1（通常出现在 if、do、while 和 return 语句中），μBERT 通过添加一个新条件 e_2,对其进行变异，从而生成新条件 $e_1||e_2$（或 $e_1\&\&e_2$），随后由 CodeBERT 对其进行掩码并补全。例如，给定条件 if(a == b)，μBERT 生成新条件 if(a== b || a > 0)，该条件经过掩码后生成 if(a ==b ||b > 0)。

我们实现了 μBERT，并评估其发挥变异测试主要作用的能力，即引导测试以发现故障。因此，我们使用 Defects4J 中的 689 个故障对其进行评估，并评估 μBERT 的有效性和

• *A. Khanfir*、*R. Degiovanni*、*M. Papadakis* 和 *Y. Le Traon* 就职于卢森堡大学，卢森堡。

揭示这些故障的成本效益。结果表明，就故障揭示而言，μBERT 非常有效，平均能够发现 84% 的故障。这意味着 μBERT 变异体能够有效覆盖真实缺陷所导致的故障行为。更重要的是，与传统变异测试技术 PiTest 相比，这种方法的有效性和成本效益明显更高；我们在评估中将 PiTest 用作基准。具体而言，我们考虑了 PiTest 的三种不同配置，它们使用不同的变异算子集（DEFAULT、ALL 和 RV）。事实上，能够杀死 μBERT 所有变异体的测试套件，平均发现的缺陷比为杀死 PiTest 引入的所有变异体而生成的测试套件多 5.5% 至 33%。此外，即使分析相同数量的变异体，μBERT 生成的测试套件平均发现的缺陷也比 PiTest 多 6% 至 16%。这些结果颇具前景，支持将 μBERT 用作测试生成方法和评估准则，以取代所考察的变异测试技术。

我们还研究了基于条件播种的变异对 μBERT 故障检测能力的影响。我们发现，与仅针对直接 CodeBERT 预测生成的变异体而设计的测试套件相比，旨在杀死两类 μBERT 变异体（分别由 1）直接 CodeBERT 预测，以及 2）条件播种与 CodeBERT 预测相结合生成）的测试套件，平均能够多发现 9% 以上的缺陷（1）。

总的来说，我们的主要贡献如下：

- 我们提出了 μBERT，这是首个采用预训练语言模型的变异测试方法。它利用模型在大型代码语料库上预训练时所获取的代码知识，以及捕捉程序上下文的能力，生成“自然”的变异体。
- 我们提出了新的加性变异：在目标代码现有的条件表达式中播种新的条件，然后对其词元进行掩码，并用模型预测结果替换。
- 我们提供了实证证据，表明 μBERT 变异体能够引导测试实现更高的故障检测能力，在有效性和成本效益方面优于最先进的技术（即 PiTest）。在实证研究中，我们还验证了采用新的加性变异模式的优势，即有助于提升编写具备更高故障揭示能力的测试套件时的有效性和成本效益。

2 背景

2.1 变异测试

变异分析 [47] 是一种测试充分性准则，它通过变异体来表示测试需求。变异体是通过对原始程序进行轻微的语法修改而获得的。例如，表达式 `x > 0` 可以通过将关系运算符 `>` 替换为 `<`，变异为 `x < 0`。随后，这些变异体将用于评估测试套件在检测相应代码修改方面的有效性和全面性。

1. 编写或生成测试以杀死（揭示）变异体。如果用于杀死某个错误的变异体的测试也能揭示该错误，那么变异测试方法就揭示了该错误。

如果一个测试用例能够在变异体与原始程序之间产生可区分的可观察输出，那么它就能检测出该变异体。如果某个变异体被测试用例或测试套件检测到，就称其为已杀死；否则，称其为存活或幸存。有些变异体在功能上与原始程序等价，因此无法被杀死。变异分数用于衡量测试套件的充分性，计算方法是已杀死的变异体数量与生成的变异体总数之比。

2.2 生成式语言模型

深度学习方法的进展催生了用于代码生成的新型语言模型 [1], [4], [15], [22]。这些模型基于包含多个项目的大型语料库进行训练，从而获得了丰富的代码知识，使其能够准确地向开发者预测源代码。在这些预训练模型中，CodeBERT [22], 是一种近期推出的语言模型，由微软向研究人员开放使用。

CodeBERT 是一种自然语言-编程语言双模态预训练语言模型（Natural Language Programming Language），支持代码搜索、代码文档生成等多种应用。与大多数大型预训练模型（如 BERT [20],）一样，CodeBERT 的开发采用了多层Transformer [55] 架构。它使用从 GitHub 上超过 640 万个项目中收集的大型语料库进行训练，涵盖包括 Java 在内的 6 种不同编程语言。该模型以跨模态方式进行训练，使用双模态自然语言-编程语言数据（输入数据由源代码及其相关文档组成），以及单模态数据（每个输入中仅包含自然语言或编程语言序列）。这样，模型便能够同时提供编程语言和自然语言-编程语言两类功能。训练采用基于替换词元检测的混合目标函数。

μBERT 在其工作流程中集成了 CodeBERT 的掩码语言建模（MLM）功能 [2]，以生成“自然”的变异体。CodeBERT 掩码语言建模流程以一段代码序列作为输入，最多包含 512 个词元，其中一个词元被标记为 <mask>，模型会根据从其余词元中捕获的上下文预测该词元的值。CodeBERT 默认会为每个词元提供 5 个预测结果，我们从中选取不准确且可编译的预测代码作为变异体。

3 方法

我们提出了 μBERT，这是一种基于生成式语言模型的变异测试方法，其步骤如图 1 所示。给定输入源代码，μBERT 利用 CodeBERT 对代码的理解，以及捕获程序上下文的能力，生成“自然”的变异，即与开发者可能犯下的错误相似的变异。为此，μBERT 按以下六个步骤执行：

1) 首先，它会提取需要进行变异的相关位置（AST [2] 节点）。
2) 然后，它会屏蔽已识别节点中的词元，并为每个选定的词元创建一个屏蔽版本。

2. AST：抽象语法树。

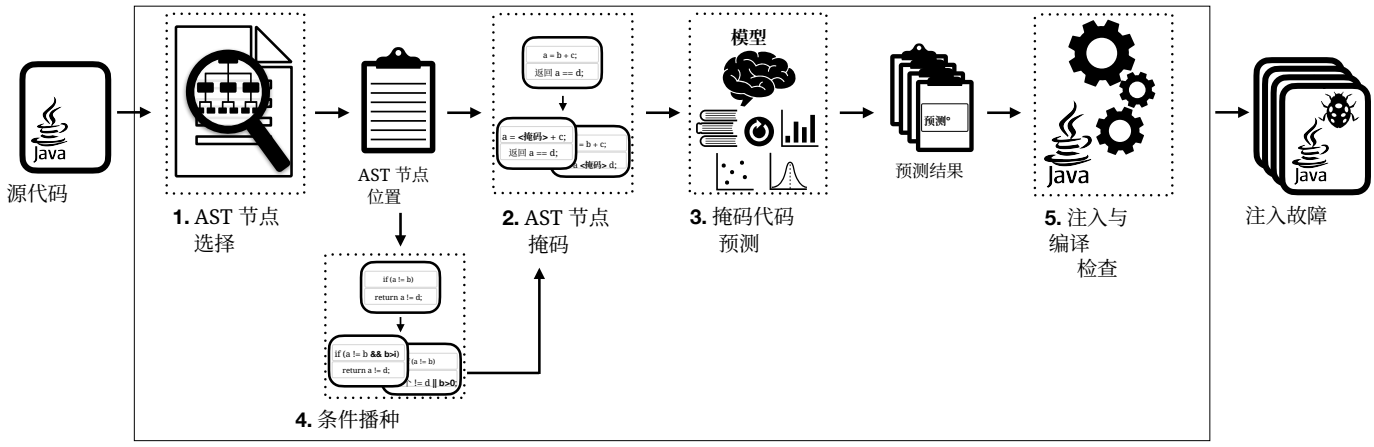


图 1： μBERT 工作流程：(1) 解析作为输入提供的 Java 代码，并提取待变异的表达式；(2) 通过掩码感兴趣的词元并调用 CodeBERT，创建简单替换变异体；(3) 通过用 CodeBERT 预测结果替换掩码令牌，生成变异体；(4) 通过 a) 条件播种、b) 掩码词元，然后 c) 用 CodeBERT 预测结果替换，生成复杂变异体；最后，(5) 丢弃无法编译以及语法相同的变异体。

3) 接着，它会调用 CodeBERT，为这些被屏蔽的词元预测替换内容。4) 除了步骤 (3) 生成的变异体外， μBERT 还实现了一些条件种子式的加性变异，这些变异会修改多个词元。具体而言，它会修改控制流中的条件表达式（通常出现在 if、do、while 和 return 语句中），通过添加一个新条件来扩展原始条件，并使用逻辑运算符 && 或 || 将两者组合。然后，按照相同的步骤 (2) 和 (3) 对新的条件表达式进行变异——屏蔽词元，并用 CodeBERT 的预测结果替换被屏蔽的词元。

5) 最后，该方法会丢弃重复的预测结果、会生成与原始代码相似代码的预测结果，以及无法编译的预测结果，并将剩余结果作为变异体输出，这些变异体来自目标代码中的不同位置。更具体地说，该方法会以随机顺序遍历各条语句，并在每次迭代中每行输出一个变异体，直到达到所需的变异体数量，或输出所有变异体为止。

3.1 AST 节点选择

μBERT 会解析输入源代码的抽象语法树，并选取更可能实现程序规范的代码行，同时排除导入语句和声明语句，例如声明类、方法、属性等的语句。这样一来，该方法便将变异集中在程序的业务逻辑部分，并排除那些可能对程序行为影响较小的代码行。随后，它会从这些语句中选出相关的待变异节点，即运算符、操作数、方法调用和变量等，并排除特定于语言的元素，例如分隔符和流程控制结构，即分号、括号、if、else 等。表1总结了 μBERT 的目标 AST 节点类型，并列出相应的表达式示例和生成的变异体。我们将这些称为 μBERT 提供的传统变异，在我们的评估中记为 μBERT_{conv} ，这类变异已在该方法的初步版本 μ 中介绍。

我们的评估，此前已在该方法的初步版本 [18] 中介绍。

3.2 词元屏蔽

在此步骤中，我们逐一屏蔽选定的节点，为每个感兴趣的节点生成一个源代码的屏蔽版本。这意味着，每个屏蔽版本都包含原始代码，其中一个节点被移除，并由占位符 <mask> 替代。

这样， μBERT 就能在同一程序位置生成多个变异体。例如，对于像 res = a + b 这样的赋值表达式， μBERT 将从以下屏蔽序列中生成（可能多达 25 个）变异体：

- **<掩码>** = 一个 + 乙
- 结果 **<掩码>**= a + b
- 结果 = **<掩码>** + b
- 结果 = a **<掩码>** b
- 结果 = a + **<掩码>**

3.3 CodeBERT-MLM 预测

μBERT 调用 CodeBERT 来预测掩码节点的替换内容。为此，它会将每个掩码版本进行词元化，转换为词元向量，然后将其裁剪为符合模型允许的最大长度（512）的子集，并统计掩码令牌及其周围的代码词元。接下来，我们的方法将这些向量输入 CodeBERT 掩码语言模型，以预测掩码令牌最可能的替换内容。我们的直觉是，伴随掩码占位符的代码部分越大，CodeBERT 就越能够捕捉代码上下文，因此其预测结果也会越有意义。该步骤最终会为每个掩码令牌生成五个预测结果。

3.4 条件播种

μBERT 通过将条件播种与 CodeBERT 的预测能力相结合来生成二阶变异体。为此，我们的方法会修改控制流中的条件和

表1： μBERT传统变异体示例，适用于该方法的初步版本 [18], , 记为μBERT$_{conv}$。

AST节点	表达式	掩码表达式	变异体示例
字面量	结果 + 10	结果 + **<掩码>**	结果 + **0**
标识符	结果 + 10	**<掩码>** + 10	**一个** + 10
二元表达式	a && b	**一** **<掩码>**二	a **|** b
一元表达式	--a	**<掩码>**一个	**++**a
赋值语句	总和 += 当前	总和 **<掩码>**=当前	总和 **−**=当前
对象字段	node.next	node.**<掩码>**	**节点。上一页**
方法调用	list.add(node)	list.**<掩码>**(node)	**列表。推入**(节点)
数组访问	arr[索引 + 1]	arr[**<掩码>**]	arr[**索引**]
静态类型引用	Math.random() * 10	**<遮罩>**.random() * 10	**随机**.random() * 10

return 语句，包括 if、do、while 和 return 条件表达式。对于这些语句中的每一种，它首先会在原始条件的基础上扩展一个新条件，两者之间用逻辑运算符 && 或 || 分隔；同时会尝试两种顺序（原始条件在前或新条件在前），并分别尝试是否添加否定（!）。

接下来，将所有替代条件逐一插入原始代码中，形成多个条件种子代码版本，并将其作为输入传递给步骤（2）；在步骤（2）中，这些版本的词元会被掩码，然后在步骤（3）中，逐一传递给 Code-BERT，以预测各自对应的掩码词元的最佳替代项。

种子条件通过两种方式创建：

3.4.1 使用同一类中的现有条件

要变异给定的条件——if、do、while 和 return 条件表达式——我们会收集同一类中存在的所有其他条件，然后使用逻辑运算符将每个条件与目标条件组合起来。

具体而言，令 Exp_t 表示要变异的条件表达式，$S_E = \{Exp_0, ..., Exp_n\}$ 表示同一类中出现的其他条件表达式集合，但不包括空值检查表达式（即 var == null）。为 Exp_t 生成的备选替换条件由以下组合构成：

- Exp_t op neg Exp_i and
- Exp_i op neg Exp_t,

其中，op 是二元逻辑运算符，取值为 { &&、||}；neg 可以是否定运算符 !，也可以为空；Exp_i 是来自 S_E 的一个条件。

3.4.2 使用同一类中的现有变量

当待变异的目标 if 条件表达式包含变量（包括字段）时，我们会将这些变量与同一类中类型相同的其他变量组合，创建新的附加条件。然后使用逻辑运算符，将每个新创建的条件分别与原始条件组合。

具体而言，设 Exp_t 是一个包含一组变量 S_{vt} 的待变异条件表达式。对于 S_{vt} 中的每个变量 var_t，我们加载同一类中出现的、类型与 var_t 相同的其他变量集合 T，然后生成以下新条件：

- Exp_t op (var_t rel_{op} var_i)，以及
- (var_t rel_{op} var_i) op Exp_t,

其中，op 是一个二元逻辑运算符，取值为 {&&、||}、rel_{op}；rel_{op} 是适用于类型 T 的关系运算符，而 var_i 是来自 S_v 的变量。

3.5 变异体筛选

在此步骤中，我们的方法首先丢弃正确的预测和重复的预测；这些冗余预测与原始代码完全相同。随后，它遍历各条语句，并在每次迭代中按行选择一个可编译的预测，同时丢弃不可编译的预测。选出所有一阶变异体（通过单次令牌替换生成）后，我们的方法以相同的迭代方式继续选择二阶变异体（通过条件播种与一次令牌替换的组合生成）。 μBERT 持续迭代，直到达到所需的变异体数量，或输出所有变异体。

4 研究问题

我们首先分析加性变异（又称条件植入变异）对旨在杀死 μBERT 变异体的测试套件的故障检测能力有何提升。因此，我们提出以下问题：

RQ1 （μBERT 加性变异）加性变异对旨在杀死BERT 变异体的测试套件的故障检测能力有何附加价值?

故障检测能力?

为回答这一问题，我们使用 μBERT 生成两组变异体：1）第一组采用所有可能的变异，我们将其称为 μBERT；2）第二组仅采用传统的 μBERT 变异——这是我们初步实现的一部分 [18], , 但不包括加性变异——我们将其称为 μBERT$_{conv}$。随后，我们评估为杀死各组变异体而选取的测试套件的故障检测能力。

这一问题的答案提供了证据，表明加性变异能够提升μBERT 的故障检测能力。不过，为了评估其整体性能，我们将其与最先进的（SOA）变异测试，尤其是 PiTest [17],进行比较。因此，我们提出以下问题：

RQ2 (故障检测) μBERT 与当前最先进方法相比如何——在故障检测方面，变异测试的效果如何?

为回答这一问题，我们使用 PiTest 的最新版本 [17], 在与 RQ1 相同的目标项目上生成变异体。由于我们关注的是比较方法，而不是比较工具的实现，因此排除 PiTest 未能正确运行或未生成任何变异体的受试对象。这样，通过确保两种方法所统计的研究受试对象完全相同，我们获得了公平的比较基础（更多详情见第 5 节）。然后，我们比较测试套件的故障检测能力

这些测试套件被选定为杀死每种方法生成的相同数量的变异体。

最后，我们对使用 μBERT生成的一些变异体进行定性分析，并提出以下问题：

RQ3（定性分析） μBERT生成的变异体是否不同于传统变异测试算子生成的变异体?

为回答这一问题，我们展示由 μBERT生成、能够帮助检测出PiTest未能发现的故障的变异体。此外，我们还通过缩小传递给模型的掩码代码规模并重新运行模型，进而讨论捕获程序上下文对 μBERT运行的重要性，并将生成的变异体示例与原始设置下获得的示例进行比较。

5 实验设置

5.1 数据集与基准

为评估 μBERT的故障检测能力，我们使用软件工程研究领域常用数据集Defects4J [29]v2.0.0中的真实缺陷。在该基准中，每个受试缺陷都提供一份含错误版本的源代码及其对应的修复版本，并配有一个在修复版本上通过、且至少有一项测试在含错误版本上失败的测试套件。该数据集包含800多个缺陷；在报告时，我们排除了存在问题的缺陷，例如修订版本标识符错误、无法编译、存在执行问题，或在修复版本上有测试失败的缺陷。接下来，我们在其余缺陷的修复版本中，针对受缺陷影响的对应类运行 μBERT和PiTest，并排除没有任何工具生成变异体的缺陷。最终， μBERT覆盖了689个缺陷，PitTest覆盖了457个缺陷。由于我们关注的是比较方法，而非工具的实现，并希望排除与环境（例如各技术支持的Java和JUnit版本等）或数据集的局限与不足相关的潜在威胁，因此每项比较研究都使用仅包含所有待比较方法均覆盖的缺陷的数据集：回答RQ1时为689个缺陷，回答RQ2和RQ3时为457个缺陷。

5.2 实验过程

为评估基于条件播种的变异在故障揭示方面的互补价值和附加价值（对 RQ1 的回答），我们分别在不包含和包含这些额外变异体的情况下运行我们的方法——分别命名为μBERT 和 μBERT$_{conv}$—— 从而在数据集程序的修复版本上生成所有可能的变异体。接下来，我们比较为杀死每组中的变异体而生成的测试套件的平均有效性；这些测试套件分别由 μBERT 和 μBERT$_{conv}$诱导生成。

在验证所提出的基于条件播种的变异体所带来的附加价值后，我们将其性能与现有技术变异测试进行比较（回答 RQ2 和研究问题3）。我们使用 PiTest [17],这一稳定成熟的 Java 变异测试工具，因为它比其他工具更擅长发现缺陷 [33] ，并且截至目前，它是研究人员和从业者最常用的工具之一 [47], [52], 。该工具提供了不同的配置，可通过排除或纳入变异算子来调整生成的变异体及其总体成本，以适应目标用户。我们使用了以下三种配置：

- Pit-all (ALL)，它统计当前版本[3]中所有可用的变异算子。
- Pit-default (DEFAULTS)，其变异算子经过选择，形成一个稳定且具成本效益的算子子集，通过生成更少但更相关的变异体来实现这一点。
- Pit-rv-all (ALL)，这是一个版本[4] ，包含 Pit-all 的变异算子以及额外的实验性 [7]变异算子，这些算子可供研究使用。

为比较不同的方法，我们评估它们在实现变异测试主要目的之一——即引导测试以获得更高的故障检测能力——方面的有效性和成本效益。因此，我们模拟一个变异测试用例场景，其中开发者/测试人员选择变异体并编写测试来杀死它们 [13], [34]。

我们在 Defects4J 提供的修复版本和测试套件上运行每种方法，然后收集变异体及其测试执行结果：变异体是否被杀死（即导致测试套件中的至少一个测试失败），若是，则是被哪些测试杀死。接下来，我们假设未被杀死的变异体是等价的或无关紧要的，并据此解释开发者为何没有编写测试来杀死它们。然后，我们模拟开发者测试修复版本的场景：在这种情况下，1）没有任何测试；2）因此所有变异体都没有杀死它们的测试；3）开发者也不知道哪些变异体是等价的。通过这种方式，我们可以重现开发者以下工作流程：

1）选择并分析一个变异体；2）然后要么（a）如果该变异体是等价的（未被实际测试套件中的测试杀死），就将其从变异体集合中剔除，要么（b）编写一个测试来杀死它（从实际能够杀死该变异体的测试中选择一个）；3）随后剔除所有被该测试杀死的变异体；4）对剩余变异体重复上述过程，直至全部分析完毕。

如果由开发者编写（选取）的测试所组成的测试套件中，至少有一个测试能够揭示某个错误（即在含错误版本上执行时会失败的测试），我们就称该错误被某种变异测试技术发现。

我们以分析的变异体数量来表示测试成本，因此，将发现错误所需的工作量视为编写出第一个揭示错误的测试之前所分析的变异体数量。为了在不同方法之间建立统一的比较基准，并考虑生成的变异体数量不同，我们将模拟运行至达到相同的最大工作量（待分析的变异体最大数量）；该上限设为其中一种对比方法杀死所有变异体所需的最低成本。在评估研究中，所有对比方法均采用相同的变异体选择策略：按随机顺序遍历各行，并在每次迭代中为每行选择一个任意变异体。为了降低过程随机性对结果的影响

3.PitTest 的 GitHub 仓库中提供的版本 1.9.4（分支=master，仓库=https://github.com/hcoles/pitest.git，修订 ID=17e1eecf） 4.PitTest 的 GitHub 仓库中提供的版本 1.7.4（分支=master，仓库=https://github.com/hcoles/pitest.git，修订 ID=2ec1178a）

（包括变异体和测试的选择），我们对每次模拟运行 100 次，然后针对每个目标缺陷和所考察的方法，分别计算结果的平均值。最后，我们汇总所有目标缺陷的这些平均值，并将其归一化为全局百分比，以变异体分析数量衡量每单位投入精力所实现的故障检测率。

最后，为回答 RQ3，我们根据 RQ2 的结果，选取一些使 μBERT 能够独自发现缺陷（任何版本的 PiTest 都未发现）的示例变异体。然后，我们通过分析这些变异体与修复版本在行为上的差异，以及它们与有缺陷版本的相似性，讨论 μBERT 变异体带来的附加价值。

5.3 实现

我们按照第 3 节所述实现了 μBERT 的方法：使用 Spoon [51] 和 Jdt [21] 库解析并提取与业务逻辑相关的 AST 节点，并应用条件播种变异算子。为预测掩码词元，我们使用了 CodeBERT-nt [3], [31], 提出的方法，该方法采用 CodeBERT 掩码语言建模（MLM）功能 [2], [22]。

我们在 https://github. com/Ahmedfir/mBERTa 提供了该方法的实现及其评估的复现包。

6 实验结果

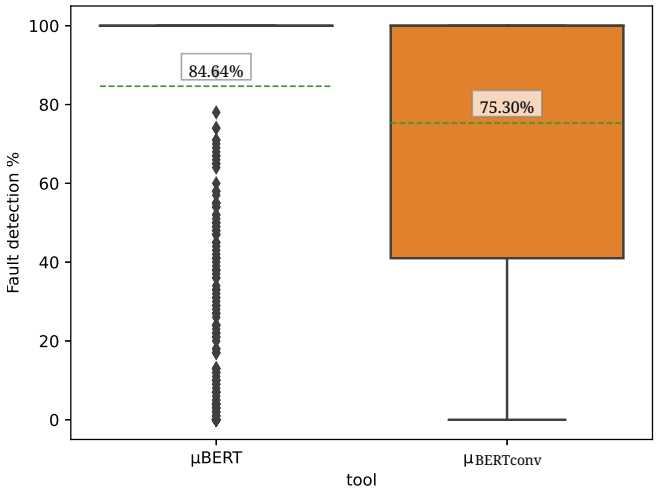
6.1 RQ1： μBERT 加性变异

为回答这个问题，我们比较了用于杀死由 μBERT生成的变异体的测试套件的故障检测有效性，这些变异体分别是在加入和未加入加性变异的情况下生成的，分别记作 μBERT和 μBERT$_{conv}$。图2展示了在 μBERT变异体中加入加性变异体后，故障检测能力的提升。事实上，与μBERT$_{conv}$相比， μBERT的故障检测能力平均提高了9%以上，平均达到84.64%。我们还可以看到，除离群值外，大多数缺陷在所有情况下都能被发现。此外，单独考察这些缺陷时，我们发现 μBERT比 μBERT$_{conv}$多发现了20个缺陷（故障检测率为 $> 0\%$）；若只考虑故障检测率超过90%时发现的缺陷，则多发现了70个。这证实了加性模式能够生成有意义的变异体，从而确保始终或在大多数情况下检测到某些缺陷，同时也能更好地体现以其他方式无法检测的新型故障。

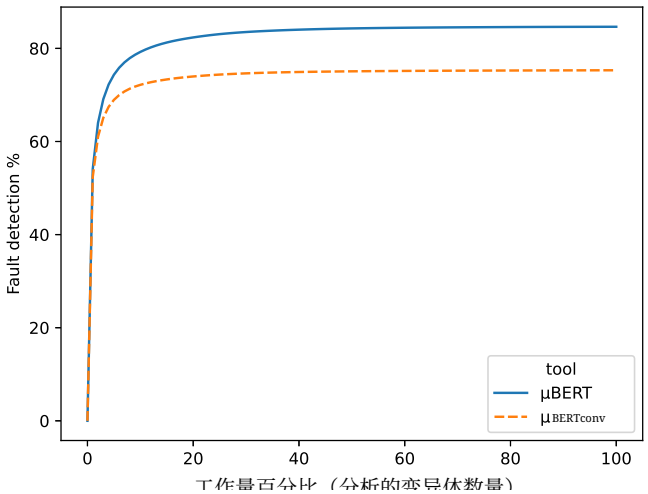
为检验加性模式带来的故障检测优势是否显著，我们对图2a中的数据进行了统计检验（Wilcoxon 配对检验），以验证以下假设：“由

μBERT **优于**另一种 μBERT$_{conv}$ ”。极小的p值 5.92e-21（$\ll 0.05$）表明差异具有显著性，这意味着这种故障检测能力的提升不太可能是偶然发生的。ˆ差异大小也证实了同样的优势，A_{12}值为0.5827（> 0.5），这表明在大多数情况下， μBERT生成的测试套件具有更高的故障检测能力。

接下来，我们比较μBERT 和 μBERT$_{conv}$ 在分析相同数量的变异体时的故障检测性能，并在图3中展示它们的平均故障



(a) 有效性：每个受试对象的平均故障检测数。

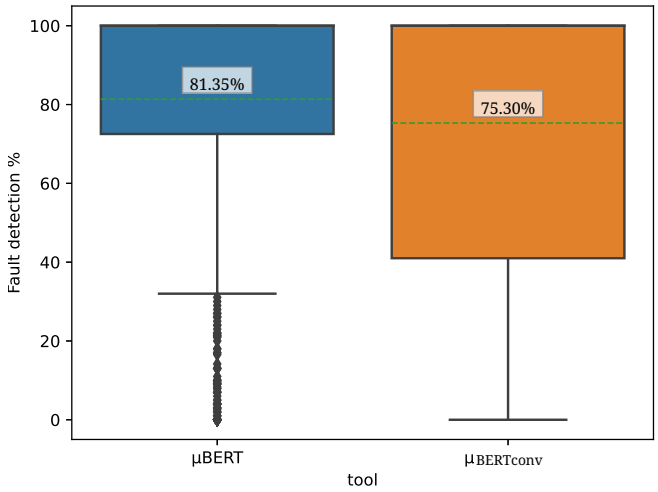


(b) 成本效益：按分析的变异体数量计算的故障检测能力。

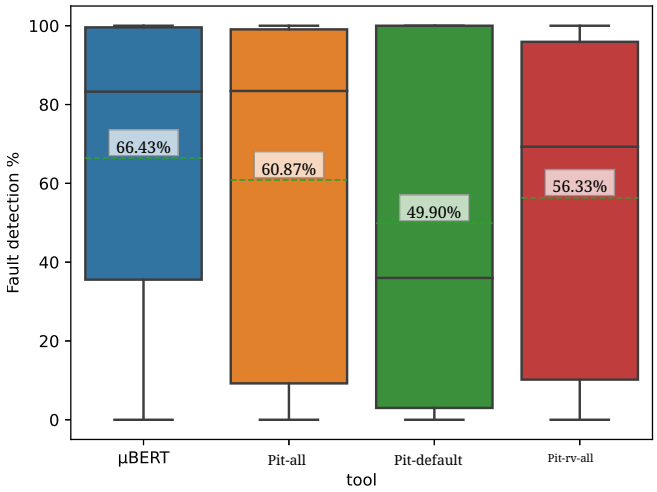
图 2：使用加性模式时故障检测性能的提升。比较 μBERT与μBERT$_{conv}$在检测旨在杀死**所有**生成变异体的测试套件方面的表现。

检测有效性和按分析变异体数量衡量的成本效益。子图3a的箱线图显示，即使投入与 μBERT$_{conv}$ 相同的工作量， μBERT仍保持相似的优势，平均故障检测率高出 6.05%，最高达到 81.35%。从子图3b的折线图可以看出，两种方法在最大成本的 ($\leq\approx 40\%$) 处均实现了相当的故障检测率 ($\approx 70\%$)。在成本较高时， μBERT$_{conv}$的曲线缓慢上升，直到工作量达到 $\approx 60\%$ 时趋于平稳；而 μBERT的曲线则持续上升，朝着更高的故障检测率迈进，即使工作量已达到固定最大工作量的$\approx 100\%$ 。

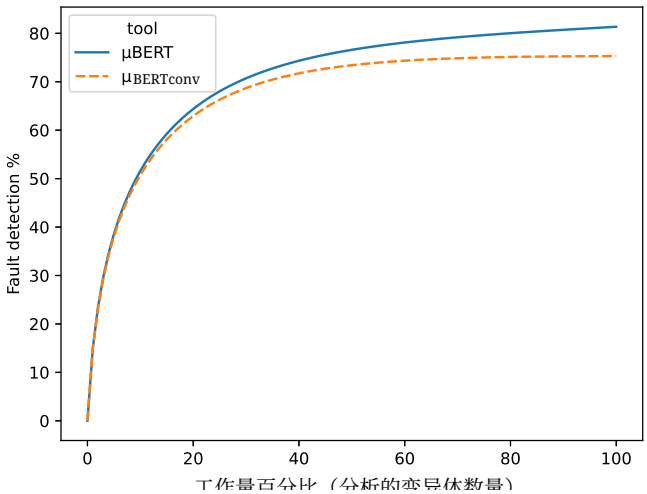
为验证这些发现，我们针对子图3a中的数据重新进行了相同的统计检验，发现μBERT的表现显著优于 μBERT^{conv} ，其ˆp值仅为1.15e-19，A_{12} 值为0.5711。



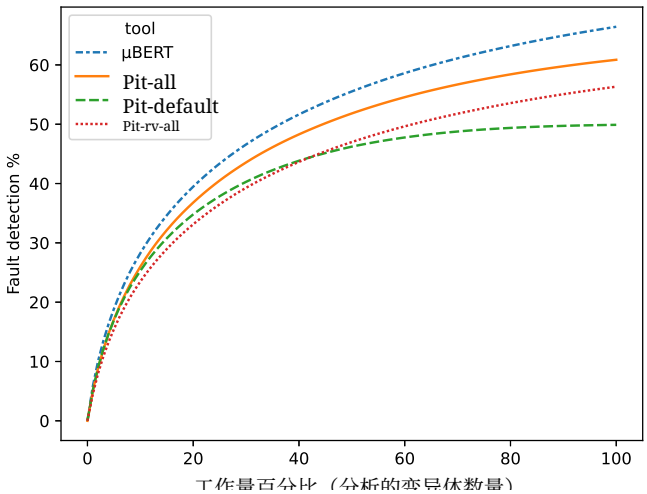
（a）有效性：每个受试对象的平均故障检测数。



(a) 有效性：每个受试对象的平均故障检测数。



（b）成本效益：按已分析的变异体数量计算的故障检测情况。



(b) 成本效益：按分析的变异体数量衡量故障检测。

图3： μBERT与μBERT$_{conv}$的故障检测对比，**工作量相同**：最大工作量限制为分析其中任一方法的所有变异体所需的最小工作量；在大多数情况下，这一方法是 μBERT$_{conv}$。

图4： μBERT与PiTest的故障检测对比，**工作量相同**：最大工作量限制为分析其中任一方法的所有变异体所需的最小工作量；在大多数情况下，这一方法是Pit-default。

6.2 RQ2：与PiTest的故障检测对比

为回答这一研究问题，我们将数据集缩减为 μBERT覆盖的错误，以及所考虑的PitTest方法的3个版本：包含PiTest默认变异算子的“Pit-default”、包含所有PiTest算子（包括默认算子）的“Pit-all”，以及除“Pit-all”中的算子外还包含实验性算子 [7] 的“Pit-rv-all”。接着，我们进行与RQ1相同的研究，根据为杀死这些方法生成的变异体而编写的测试套件的故障检测能力，比较所考虑方法的有效性和成本效益。为了公平比较，我们在分析变异体的工作量相同的条件下比较这些方法；该工作量等于其中一种方法杀死所有变异体所需的最低平均工作量（在大多数情况下，这是Pit-default所需的工作量）。由于我们关注的是比较变异测试方法，而非变异体选择策略，因此我们使用相同的单变异体选择策略运行模拟对所有技术按行随机抽样变异体（见第5.2小节）。

图4b显示，在投入较少工作量（$\leq\approx$ 5%）时，所有方法的故障检测得分都相近（$\approx$ 10%）。然而，随着投入更多工作量，差异变得更加明显： μBERT 的表现优于 PiTest 的所有版本；其故障检测得分平均比 Pit-default 高 16.53%，比 Pit-rv-all 高 10.10%，比 Pit-all 高 5.56%（见图4a）。

为验证这些结果，我们执行了与 RQ1 中相同的统计检验，以检验假设：“μBERT 的故障检测能力优于其他方法”。表 2a 和表 2b 的第一行展示了计算得到的 Wilcoxon 配对检验 p值ˆ以及 Vargha 和 Delaney A_{12} 值。结果表明，μBERT 相较于所考察的 SOA方法具有显著优势，p值低于 0.05。此外，μBERT 的ˆA_{12} 值高于 0.5，这证实了引导

表 2：对各受试对象（每个错误）的测试套件平均故障检测率进行配对统计检验；这些测试套件旨在杀死由各方法生成的**相同数量的变异体**（图4a的数据）。

（a）在每个数据集主题上计算 Wilcoxon 配对检验 p值，将第一列中的方法（A1）与其他方法（A2）进行比较。p值小于 0.05 表示（A1）的平均故障检测率显著高于（A2）。

p值	Pit-rv-all	Pit-default	Pit-all
μBERT	7.78e-11	1.18e-12	3.32e-02
Pit-all	1.54e-22	8.87e-06	–
Pit-default	9.55e-01	–	–

ˆ(b) 在每个数据集受试对象上计算的 Vargha 和 Delaney A_{12} 值，用于比较第一列中的每种方法 (A1) ˆ与其他方法 (A2)。A_{12} 值高于 0.5 表示在大多数情况下，(A1) 的平均故障检测效果高于 (A2)。

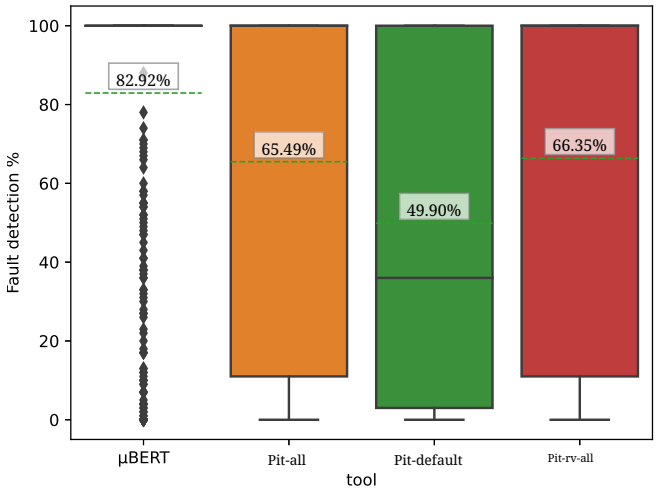
$\hat{A}_{12}$	Pit-rv-all	Pit-default	Pit-all
μBERT	0.6488	0.5514	0.5066
Pit-all	0.7210	0.4956	–
Pit-default	0.5449	–	–

使用 μBERT 变异体而非 SOA 技术生成的变异体进行测试，在大多数情况下能够取得相当或更高的故障检测ˆ率。事实上，Pit-all 与 μBERT 之间的 A_{12} 差异很小（0.5066），表明两种方法在某些受试对象上的表现相近或更好，而在其他受试对象上的表现则较差。

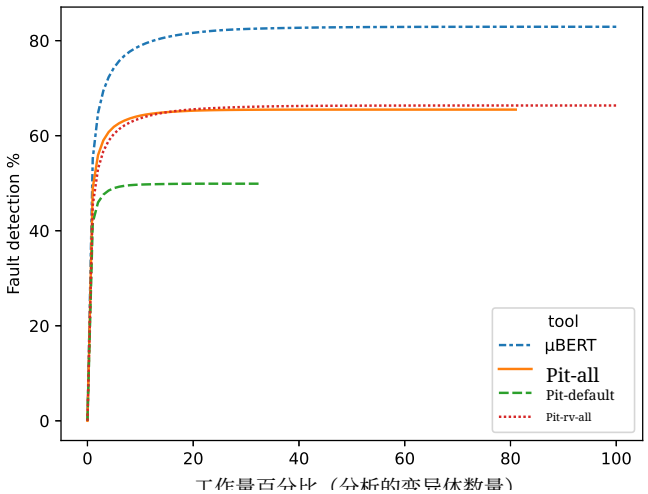
我们还从子图 4b 中注意到，Pit-default 在工作量达到约 60% 时便趋于平稳，而其他工具仍在持续提升，这表明它们能够实现更高的故障检测能力，但代价也更高。将图 4 的子图 (a) 和 (b) 与图 2 进行比较时，这一点尤为明显：μBERT 的平均故障检测率远低于其在 RQ1 中的表现——约为 66%，而非 84%。这直接是因为 Pit-default 生成的变异体少于其他方法，从而大幅限制了变异活动的最大工作量，进而降低了多数情况下的故障检测率。事实上，如图 5 所示，所有方法在投入更多工作量时都能取得更高的故障检测百分比，平均而言，Pit-all 为 ≈65%，Pit-rv-all 为 ≈66%，μBERT 为 ≈83%。与 RQ1 相比，μBERT 的平均故障检测率略微下降了 1.72%（RQ2 中为 82.92%，而 RQ1 中为 84.64%）；我们将此归因于两个 RQ 所采用的数据集不同。

在表3中，我们展示了Â12以及根据子图5a和线图数据计算出的p值。结果证实，μBERT在测试套件的故障检测能力方面显著优于最先进的变异测试；这些测试套件旨在杀死各方法生成的所有变异体。

接下来，我们将关注点转向在投入相同工作量时，各种方法能够和无法揭示的特定缺陷集合。因此，我们根据图4a中的模拟结果，将每个缺陷与能够揭示它的工具对应起来，并在图6中展示相应的维恩图。



(a) 有效性：每个受试对象的平均故障检测数。



(b) 成本效益：根据分析的变异体数量衡量故障检测。

图 5：比较 μBERT 和 PiTest 在测试套件故障检测方面的表现；这些测试套件旨在杀死**所有** 生成的变异体。

从子图 6a 中互不相交的集合可以看出，与所考虑的 SOA 基线相比，使用 μBERT 具有明显优势：在投入相同工作量时，它不仅能发现这些基线所发现的大多数缺陷，还能独家发现 47 个缺陷。具体而言，μBERT 分别比 Pit-all、Pit-default 和 Pit-rv-all 多发现 52、77 和 52 个缺陷，而它们各自发现了 μBERT 未能发现的 13、10 和 13 个缺陷。

这印证了这样一个事实：μBERT 引入的变异体比 SOA 变异技术所引入的变异体代表了更多真实缺陷，也促使我们探究这些方法最终能否互为补充。当考虑各方法在至少 90% 的模拟中发现的缺陷之间的重叠情况时，这一观察更为明显（子图 6b）。我们注意到，这些方法的表现相当，其中 Pit-all 和 Pit-default 的结果尤为突出：它们分别发现了 19 个和 21 个具有如此高故障检测率的缺陷，而子图 6a 中观察到的数量为 0。尽管如此，μBERT 相较于所考虑的基准仍保持同样的优势，

表 3：对每个受试对象的缺陷进行配对统计检验，比较各方法生成的**所有变异体**的测试套件平均故障检测率（图 5a 的数据）。

(a) 在每个数据集主题上计算 Wilcoxon 配对检验的 p值，将第一列中的方法（A1）与其他方法（A2）进行比较。p值小于 0.05 表示（A1）的平均故障检测率显著高于（A2）。

p值	Pit-rv-all	Pit-default	Pit-all
μBERT	2.49e-13	2.14e-33	1.47e-14
Pit-all	4.71e-01	2.76e-23	–
Pit-default	1.00e+00	–	–

ˆ(b) 在每个数据集受试对象上计算的 Vargha 和 Delaney$\hat{A}_{12}$ 值，用于比较第一列中的每种方法 (A1) ˆ与其他方法 (A2)。$\hat{A}_{12}$ 值高于 0.5 表示在大多数情况下，(A1) 的平均故障检测率高于 (A2)。

$\hat{A}_{12}$	Pit-rv-all	Pit-default	Pit-all
μBERT	0.6028	0.7123	0.6061
Pit-all	0.5077	0.6400	–
Pit-default	0.3676	–	–

另外，单独来看，它还发现了42个缺陷。它发现的缺陷数量也分别比Pit-all、Pit-default和Pit-rv-all多50、63和69个，而它们各自发现了μBERT遗漏的59、58和27个缺陷。

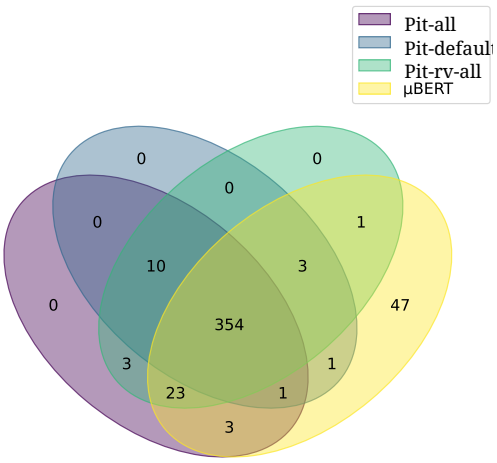
6.3 研究问题3： μBERT变异体的定性分析

为了回答这一研究问题，我们调查了 μBERT生成的变异体。这些变异体诱导生成的测试套件能够发现其他方法未能检测到的缺陷，即所考虑的SOA方法未能检测到的缺陷（见图6）。这意味着，这些变异体会破坏与目标真实含错误版本相似的测试。

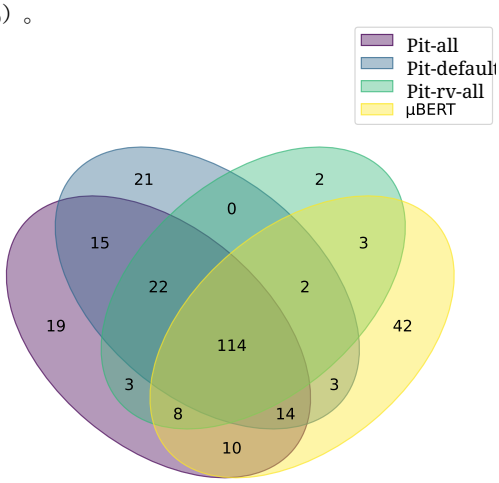
作为一个简单的错误示例（只需一次修改即可修复），我们考察 Defects4J 中的 Lang-49，并研究由μBERT 生成、且有助于生成揭示该错误的测试的变异体。该错误影响 org.apache.commons.lang.math.Fraction 类的 reduce() 方法的运行结果。该方法在可能的情况下返回一个新的约分分数实例，否则返回同一个实例。该错误是由于未正确实现某个特定边界情况所致：在分子为0 的分数实例上调用该方法。表 4 展示了由 μBERT 生成、并有助于揭示该错误的变异体示例。每个变异体都通过对比修复版本与 μBERT 变异版本的差异来表示。

可以看到， μBERT 能够生成可通过应用传统的基于模式的变异方式得到的变异体，即，变异体1 将一个关系运算符（==）替换为另一个（>），而变异体2 将一个整数操作数（0) 与另一个（1）。

此外，它还提出了更复杂的变异；若不了解抽象语法树或所分析程序的上下文，这些变异几乎不可能实现。例如，它可以通过将条件返回语句改为返回当前 Fraction 实例（this）来生成变异体 4，因为该实例与方法的返回类型相符。同样，为了



（a）每 100 次运行中至少发现一次的故障（故障检测 $> 0\%$）。



（b）在超过 90% 的运行中发现的故障（故障检测$\geq 90\%$）。

图 6：分析相同数量的变异体（工作量相同）时，用于杀死由 μBERT 和 PiTest 版本生成的变异体的测试套件所发现的故障数量。

生成变异体 5 时，它将类 Fraction 的当前实例替换为同一类型的一个现有实例（ONE），使该语句返回对象 ONE 或对象 ZERO。

为生成更复杂的变异体， μBERT 会先进行条件播种，再应用标记掩码和 Code-BERT 预测，例如在 return 语句的原始条件中添加 || (分子 ==other.分子)，从而生成变异体 8；或者在 if 语句的原始条件中添加 || !(分子== Integer.MIN_VALUE)，从而生成变异体3。

为进一步研究模型捕获的代码上下文对生成的变异体的影响，我们在数据集中选取了 5 个受试对象，重新运行了 μBERT，并将周围词元的最大数量设为 10（而非 512）。随后，我们手动比较了在相同位置诱导出的变异体与默认设置生成的变异体。根据结果，我们观察到可编译预测的数量明显减少，这表明模型缺少代码上下文信息时，整体性能会下降。具体而言，我们注意到

表4： μBERT 生成的、帮助发现 Defects4J 中 Lang-49 错误的变异体示例。

变异体1：**替换二元运算符**

```
@@ org.apache.commons.lang.math.Fraction:466 @@
-if(分子 == 0) {
+if(分子 > 0) {
```

变异体2：**替换字面量实现**

```
@@ org.apache.commons.lang.math.Fraction:466 @@
-if(分子 == 0) {
+if(分子 == 1) {
```

变异体3：**在 if 语句中添加一个条件**

```
@@ org.apache.commons.lang.math.Fraction:466 @@
-if(分子 == 0) {
+如果((分子 == 0)
+ ||!(分子==Integer.MIN VALUE)) {
```

变异体 4：**替换一个条件**

```
@@ org.apache.commons.lang.math.Fraction:467 @@
-return equals(ZERO) ? this : ZERO;
+return this;
```

变异体5：**将此访问替换为另一个对象**

```
@@ org.apache.commons.lang.math.Fraction:467 @@
-return equals(ZERO) ? this : ZERO;
+ return 等于(ZERO) ? ONE: ZERO;
```

变异体 6：**替换方法参数**

```
@@ org.apache.commons.lang.math.分数:469 @@
int最大公约数 =最大公约数(
-数学.绝对值(分子), 分母);
+数学.绝对值(分子), 1);
```

变异体 7:**替换变量**

```
@@ org.apache.commons.lang.math.Fraction:473 @@
-return Fraction.getFraction(分子/gcd,
+return Fraction.getFraction(分子/3,
      denominator/gcd);
```

变异体 8：**向 return 语句添加一个条件**

```
@@ org.apache.commons.lang.math.Fraction:840 @@
返回(getNumerator() ==other.getNumerator()
-  && getDenominator() ==other.getDenominator());
+  && getDenominator() ==other.getDenominator()))
+  ||(分子 ==other.分子);
```

模型无法生成程序特定变异体，例如将一个对象替换为另一个对象，或将一个方法调用替换为另一个。在表5中，我们列举了一些示例变异体；这些变异体有助于发现每个受研究对象中的错误（会触发与原始错误相同的测试），但当周围词元的最大数量限制为 10 时， μBERT 无法生成它们。

7 有效性威胁

一项外部效度威胁涉及我们的实证评估所得发现和结果的普遍性。为降低这一威胁，我们使用了来自热门开源项目的大量真实缺陷及其相关开发者测试套件，这些数据由一个成熟且独立构建的基准（即 Defects4J [29]）提供。尽管如此，我们承认，针对不同领域的项目，结果可能有所不同。

其他威胁可能源于我们评估变异测试方法故障检测能力的方式：我们根据这些方法通过开发者／测试人员模拟来指导测试的能力进行评估，并假设当前测试套件是完备的，且未被杀死的变异体均为等价变异体。尽管我们承认，在真实场景中情况未必如此，但我们认为这一流程足以评估我们的方法，尤其考虑到 Defects4J 提供的测试套件相对强大。此外，为减轻所考察方法之间的比较威胁，我们在所有研究中始终一致地采用相同的测试套件、设置和模拟假设。

我们所选择的比较基线可能会对研究结果的有效性构成其他威胁。尽管近期已经提出了不同的故障播种方法，PiTest 仍然是 Java 程序最成熟、最稳定的变异测试工具之一，因此适合作为评估我们工作成果的比较基线。此外，我们还将结果与 PiTest 所提出的不同配置生成的变异体所得结果进行了比较，从而将研究范围扩展到 PiTest 后者所面向的不同受众。不过，我们承认，在考虑其他技术时，结果可能会发生变化，并认为评估不同变异测试技术的有效性和成本效益超出了本文的作用域。

其他构念效度威胁可能源于将分析的变异体数量作为度量指标，用以衡量发现故障所需的工作量。除了该度量指标已被文献广泛采用这一事实 [9], [34], [47], 之外，我们认为它直观且能代表测试人员分析变异体、将其舍弃或编写测试以杀死变异体时所付出的总体人工工作量。尽管这是文献中的标准做法，我们也承认，这种度量方式没有考虑不同变异体之间的成本差异，而是为所有变异体赋予相同的成本。这只是因为我们不知道分析每个特定变异体或编写每个特定测试所需的具体工作量。此外，我们的成本效益结果可能受到该度量指标未涵盖的成本影响，例如 CodeBERT（ μBERT 的关键组件）的执行或开发工作量，或 PiTest 不同版本中的模式集和执行增强所产生的成本。尽管如此，我们仍尝试缓解由不同工具的实现所引发的任何重大威胁；也就是说，我们将数据集对象缩减为每种方法至少生成一个变异体的对象，并将方法之间的任何实现差异视为当前范围之外的问题。

8 相关工作

自20世纪70年代以来，变异测试一直是多项研究工作的重点 [57]。研究结果证明，人工故障可用于多种软件工程应用，例如测试 [47], 调试 [37],[48], 评估容错 [42], 风险分析 [16], [56]以及可信性评估 [10]。

尽管相关研究由来已久，如何生成相关变异体仍是一个悬而未决的问题。大多数相关研究都聚焦于故障

表5： μBERT 生成的变异体示例，这些变异体有助于发现 Defects4J 中的缺陷；当周围词元的最大数量限制为 10 时，则无法生成这些变异体。

变异体1（JacksonCore-4）：**替换一个方法调用**
@@ com . f a s t e r x m l . j a c k s o n . c o r e . u t i l . TextBuffer : 515 @@ – 取消共享（1）； + e x p a n d (1) ;
变异体2（Closure-26）：**替换一个对象**
@@ com . g o o g l e . j a v a s c r i p t . j s c o m p . ProcessCommonJSModules : 89 @@ – . r e p l a c e A l l (P a t t e r n . q u o t e (文件 . 分隔符) , 模块名称分隔符) – + . r e p l a c e A l l (P a t t e r n . q u o t e (文件名) , 模块名称分隔符) –
变异体3 (Closure-35)：**替换方法调用**
@@ com . g o o g l e . j a v a s c r i p t . j s c o m p . 类型推断 : 1092 @@ – 作用域 = 遍历子节点 (n , 作用域); + 作用域 = 遍历（n, 作用域）；
变异体 4（Lang-27）：**替换方法调用**

@@ org . apache . commons . lang3 . math . 数字工具类 : 526 @@
– i f (! (f . i s I n f i n i t e ( ) || (f . f l o a t V a l u e ( ) == 0 . 0 F && ! a l l Z e r o s))) {
+ i f (! (f . i s I n f i n i t e ( ) || (f . r o u n d ( ) == 0 . 0 F && ! a l l Z e r o s))) {
// 也有 *"f.floatValue()" to "f.scale()"*

变异体5 (Math-64)：**替换一个对象**

@@ org . apache . commons . lang . math . F r a c t i o n : 852 @@
– f o r (i n t j = k ; j < j a c o b i a n . l e n g t h ; ++j) {
+ f o r (i n t j = k ; j < beta . 长度 ; ++j) {

变异体 6 (Lang-27)：**替换一个对象**

@@ org . apache . commons . lang3 . math . 数字工具类 : 526 @@
– i f (!(f . i s I n f i n i t e ( ) || (f . f l o a t V a l u e ( ) == 0 . 0 F && ! a l l Z e r o s))) {
+ i f (!(f . i s I n f i n i t e ( ) || (f . r o u n d ( ) == 0 . 0 F && ! Z E R O))) {

模式（变异算子）的设计，这些模式通常基于目标语言语法定义 [8], [47] ，随后通过实证研究加以改进 [33], [40], [44] ，旨在减少其生成的变异体中的冗余和噪声。随着这方面不断取得进展，基于模式的变异测试工具及其版本也不断涌现 [17], [35], [39], ，其中一些已逐渐受到欢迎，并被研究人员和从业者广泛采用，例如 PiTest [17],。我们选取了 PiTest 的三种配置作为比较基准。

近期研究着重于提高人工故障的代表性，以便缩小变异空间，使其更接近真实故障。例如，Brown 等人 [12] 提出从真实缺陷修复中推断变异算子，而不是依据编程语言语法来设计变异算子。同样，Tufano 等人 [54]提出了一种神经机器翻译技术，能够学习如何从真实缺陷修复中注入故障。沿着相同思路，Patra 等人 [50] 提出了一种语义感知学习方法，能够学习故障模式，并将其适配到目标项目中。他们的结果颇有前景，但这些技术依赖于大量、多样、全面且相互独立的修复提交 [27] （即针对非耦合故障的提交 [43], ），而这类提交在实践中往往难以获取，因此可能会影响其效果。考虑到注入位置， [13], [42], Khanfir 等人 [32]将从缺陷报告中提取的信息与反向的自动程序修复模式结合起来，以复现可由原始修复模式修复的真实故障。他们的结果表明，他们能够生成模拟真实故障的故障；然而，他们的方法仍依赖于优质缺陷报告的存在，因而受到限制。总体而言，根据已知故障空间设计变异算子，可以产生更多样、涵盖更多故障类型的变异体。然而，这些扩展后的算子集往往会增加生成的变异体数量，从而提高变异活动的总体成本。例如，Brown 等人和 Khanfir 等人提出的故障模式除了包含新的变异算子外，也纳入了大多数传统变异算子。与这些技术不同， μBERT 利用预训练模型，基于代码知识而非故障知识来引入变异体。由于代码比故障更易获取，因此它能提供比故障更灵活、更全面的知识库，也就是说，它有助于克服以下限制和工作量：1）收集干净的缺陷修复提交；2）捕捉故障行为；3）设计故障模式，无论是手动设计还是借助机器学习技术。

为了减少生成的变异体数量，研究人员提出了不同策略来生成相关变异体。例如，有研究表明，变异体的强度不仅取决于其诱发模式，还取决于注入位置 [13], [42], ，这凸显了选择相关位置进行变异的重要性。对此，Sun 等人 [53] 建议在不同的程序执行路径中变异多个位置。Gong 等人 [26] 也提出，在通过图分析提取的程序不同位置进行变异。同样，Mirshokraie 等人 [41]通过程序执行计算复杂度指标，以提取可观测性良好的位

置进行变异。其他方法则将故障注入限制在程序的特定位置，例如最近提交所影响的代码 [38],[58] ，以提高其在持续集成中的实用性；或者针对与给定缺陷报告相关的位置 [32] ，以测试特定功能或行为等。近期的进展带来了强大的技术，能够以较低成本筛选变异体，也就是避免分析冗余变异体（基本上是等价变异体和被包含的变异体） [24], [25], [28]。尤其是 Garg 等人 [24] 的研究利用嵌入向量空间的变异体周边上下文信息，预测某个变异体是否可能包含其他变异体。在本研究中，我们不针对任何特定代码部分或狭窄的使用场景，而是像变异测试一样，以暴力方式进行故障注入：遍历每条程序语句，并屏蔽其中涉及的每个词元。

多项研究聚焦于人工故障与真实故障之间的关系 [47]。Ojdanic 等人开展的研究 [45], 、Papadakis 等人 [49]、Just 等人 [30] 以及 Andrews 等人 [9] 的研究结果表明，被错误破坏的测试与杀死变异体的测试之间存在相关性。这意味着，在受控研究中，人工故障可以作为真实故障的替代方案。此外，Chekam 等人 [14], 、Frankl 等人 [23]和 Li 等人 [36] 的研究结果表明，与其他测试充分性准则相比，以变异体指导测试能够显著提高故障揭示能力。基于这些发现，我们根据注入故障与真实故障在破坏测试方面的关系，对我们的方法进行评估。更确切地说，我们开展故障检测有效性和成本效益研究，以评估我们的方法所生成的、用于指导测试的变异体，并将其与最先进的技术进行比较。此外，我们通过真实示例讨论μBERT 变异体的多样性和可读性。

与之最接近的相关工作，是最近在 2022 年变异测试研讨会 [18] 上展示的 μBERT 初步实现。在我们的评估中，该实现记作 μBERT_{conv}，包含传统变异（由模型预测词元并对其进行掩码和替换），但不包含能显著提升故障检测效果的条件播种加性变异。此外，μBERT_{conv} 仅在 Defects4J 的 40 个缺陷上进行了评估，且只与 PiTest 的早期版本（类似于 Pit-rv-all）进行了比较。在本研究中，我们进行了大规模实验评估，涵盖 Defects4J 的 689 个缺陷，并将 μBERT 的有效性与 PiTest 的三种不同配置进行了比较。此外，我们表明， μBERT 平均发现的缺陷多于 μBERT_{conv} ，且无需付出更多工作量。

9 结论

我们介绍了 μBERT，这是一种基于预训练语言模型的故障注入方法。 μBERT 为研究人员和从业者提供了易于理解的“自然”变异体，帮助他们编写故障揭示能力更强的测试。

与最先进的方法不同，它既不需要也不依赖任何故障知识或编程语言语法，而是依赖实际的代码定义及其分布，这些代码由众多项目中的开发者编写。这使其更易于开发、维护、集成，并扩展到不同的编程语言。事实上，它降低了学习如何进行变异的成本，无论是通过创建和选择模式，还是收集优质的缺陷修复并从其补丁中学习。

简而言之， μBERT 将给定程序作为输入，并使用预训练生成式语言模型作出的预测替换其代码库中的不同部分，从而生成多个可能出现的变异。该方法面向多种业务代码位置，通过扩展控制流条件，注入简单的单词元替换变异体或更复杂的变异体。这种方法能够提供可能类似开发者所引入、影响程序不同功能的故障，且相关性更高，也能降低开发者的工作量。我们的结果进一步印证了这一点：μBERT 能以较低工作量生成故障检测能力强的测试套件，在这方面优于最先进的技术（PiTest）。

我们已公开实现和结果 [5]，以确保研究可复现并支持未来的研究。

致谢

本研究得到卢森堡国家研究基金（FNR）项目 C20/IS/14761415/TestFlakes 和 TestFast（参考编号：12630949）的支持。

参考文献

[1] Amazon CodeWhisperer。https://aws.amazon.com/codewhisperer/. [2] CodeBERT。https://github.com/microsoft/CodeBERT. [3] CodeBERT-nt。https://github.com/Ahmedfir/CodeBERT-nt. [4] GitHub Copilot。https://github.com/features/copilot. [5] mBERTa。https://github.com/Ahmedfir/mBERTa. [6] PiTest。http://pitest.org/. [7] PiTest-rv-plugin。https://github.com/pitest/pitest-rv-plugin. [8] Paul Ammann 和 Jeff Offutt。《软件测试导论》。剑桥大学出版社，2008年。[9] James H. Andrews、Lionel C. Briand、Yvan Labiche 和 Ak- bar Siami Namin。使用变异分析评估和比较测试覆盖准则。《IEEE软件工程汇刊》，32(8):608–624，2006年。[10] Jean Arlat、Alain Costes、Yves Crouzet、Jean-Claude Laprie 和 David Powell。容错系统的故障注入与可信性评估。《IEEE计算机汇刊》，42(8):913–923，1993年。[11] Moritz Beller、Chu-Pan Wong、Johannes Bader、Andrew Scott、Mateusz Machalica、Satish Chandra 和 Erik Meijer。在工业界应用变异测试需要什么——Facebook的一项研究。载于第*43*届*IEEE/ACM*国际软件工程会议：软件工程实践，*ICSE (SEIP)*，第268–277页。IEEE，2021年。[12] David Bingham Brown、Michael Vaughn、Ben Liblit 和 Thomas W. Reps。野生变异体的维护与培育。载于*2017*年第*11*届软件工程基础联合会议论文集，*ESEC/FSE*，第511–522页。ACM，2017年。[13] Thierry Titcheu Chekam、Mike Papadakis、Tegawend´e F. Bis- syand´e、Yves Le Traon 和 Koushik Sen。选择揭示故障的变异体。《实证软件工程》，25(1):434–487，2020年。[14] Thierry Titcheu Chekam、Mike Papadakis、Yves Le Traon 和 Mark Harman。一项关于变异、语句和分支覆盖故障揭示能力的实证研究，避免了不可靠的无故障程序假设。载于第*39*届国际软件工程会议论文集，*ICSE 2017*，阿根廷布宜诺斯艾利斯，*2017*年*5*月*20*日至*28*日，第597–608页，2017年。

[15]Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、J Kaplan、H Edwards、Y Burda、N Joseph、G Brockman 等。评估在代码上训练的大语言模型。(2021)。arXiv预印本 arXiv:2107.03374，2021年。[16] J¨orgen Christmansson 和 Ram Chillarege。基于现场数据生成模拟软件故障的错误集。载于论文摘要：FTCS-26，第二十六届年度国际容错计算研讨会，1996年，第304–313页。IEEE计算机学会，1996年。[17] Henry Coles、Thomas Laurent、Christopher Henard、Mike Pa- padakis 和 Anthony Ventresque。Pit：一款实用的 Java 变异测试工具（演示）。载于第25届国际软件测试与分析研讨会论文集，ISSTA 2016，第449–452页，美国纽约，2016年。计算机协会。[18] Renzo Degiovanni 和 Mike Papadakis。 μBERT：使用预训练语言模型进行变异测试。载于第15届IEEE软件测试、验证与确认国际会议研讨会，ICST研讨会 2022，西班牙瓦伦西亚，2022年4月4日至13日，第160–169页。IEEE，2022年。[19] Richard A. DeMillo、Richard J. Lipton 和 Frederick G. Sayward。测试数据选择提示：帮助实践中的程序员。IEEE Computer，11(4):34–41，1978年。[20] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。BERT：用于语言理解的深度双向变换器预训练。arXiv预印本 arXiv:1810.04805，2018年。[21] J DT Eclipse。Eclipse Java开发工具（JDT），2013年。[22] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。CodeBERT：一种面向编程语言和自然语言的预训练模型。载于2020年自然语言处理实证方法会议论文集：研究成果（EMNLP），ACL研究成果系列第EMNLP 2020卷，第1536–1547页。计算语言学协会，2020年。[23] Phyllis G. Frankl、Stewart N. Weiss 和 Cang Hu。全使用覆盖与变异测试：有效性的实验比较。《系统与软件杂志》，38(3):235–253，1997年。[24] Aayush Garg、Milos Ojdanic、Renzo Degiovanni、Thierry Titcheu Chekam、Mike Papadakis 和 Yves Le Traon。Cerebro：静态包含变异体选择。《IEEE软件工程汇刊》。[25]Rohit Gheyi、M´arcio Ribeiro、Beatriz Souza、Marcio Augusto Guimar˜aes、Leo Fernandes、Marcelo d'Amorim、Vander Alves、Leopoldo Teixeira 和 Baldoino Fonseca。使用 Z3 识别方法级变异体支配关系。《信息与软件技术》，132:106496，2021年。[26] Dunwei Gong、Gongjie Zhang、Xiangjuan Yao 和 Fanlin Meng。基于支配关系的弱变异测试变异体缩减。《信息与软件技术》，81:82–96，2017年。[27] Kim Herzig 和 Andreas Zeller。理清变更。未发表手稿，9月，37:38–40，2011年。[28] Claudinei Brito Junior、Vinicius H. S. Durelli、Rafael Serapilha Durelli、Simone R. S. Souza、Auri M. R. Vincenzi 和 M´arcio Ed- uardo Delamaro。使用机器学习算法识别最小变异体和等价变异体的初步研究。载于第13届IEEE软件测试、验证与确认国际会议研讨会（ICSTW），第304–313页。IEEE，2020年。[29] Ren´e Just、Darioush Jalali 和 Michael D. Ernst。Defects4J：现有故障数据库，用于支持针对 Java 程序的受控测试研究。载于2014年国际软件测试与分析研讨会论文集，ISSTA 2014，第437–440页，美国纽约，2014年。计算机协会。[30] Ren´e Just、Darioush Jalali、Laura Inozemtseva、Michael D. Ernst、Reid Holmes 和 Gordon Fraser。变异体能否在软件测试中有效替代真实故障？载于第22届ACM SIGSOFT软件工程基础国际研讨会论文集（FSE-22），中国香港，2014年11月16日至22日，第654–665页。ACM，2014年。[31] Ahmed Khanfir、Matthieu Jimenez、Mike Papadakis 和 Yves Le Traon。CodeBERT-nt：通过CodeBERT衡量代码自然度。arXiv预印本 arXiv:2208.06042，2022年。[32] Ahmed Khanfir、Anil Koyuncu、Mike Papadakis、Maxime Cordy、Tegawende F. Bissyand´e、Jacques Klein 和 Yves Le Traon。Ibir：由缺陷报告驱动的故障注入。《ACM软件工程与方法学汇刊》，2022年5月。[33] Marinos Kintis、Mike Papadakis、Andreas Papadopoulos、Evan- gelos Valvis、Nicos Malevris 和 Yves Le Traon。变异测试工具的有效性如何？Java 变异的实证分析

结合人工分析和真实故障的测试工具。《Empir. Softw. Eng.》，23(4):2426–2463，2018年。[34] Bob Kurtz、Paul Ammann、Jeff Offutt、M´arcio Eduardo Delamaro、Mariet Kurtz 和 Nida G¨okc。e。使用支配变异体分析选择性变异的有效性。载于第24届 ACM SIGSOFT软件工程基础国际研讨会论文集，FSE 2016，美国华盛顿州西雅图，2016年11月13日至18日，第571–582页，2016年。[35] T. Laurent、M. Papadakis、M. Kintis、C. Henard、Y. L. Traon 和 A. Ventresque。评估并改进 PIT 的变异测试实践。载于2017年 IEEE软件测试、验证与确认国际会议（ICST），第430–435页，2017年3月。[36] Nan Li、Upsorn Praphamontripong 和 Jeff Offutt。四种单元测试标准的实验比较：变异、边对、全使用和素路径覆盖。载于第二届国际软件测试、验证与确认会议（ICST）2009年研讨会论文集，第220–229页。IEEE计算机学会，2009年。[37] Yiling Lou、Ali Ghanbari、Xia Li、Lingming Zhang、Haotian Zhang、Dan Hao 和 Lu Zhang。自动化程序修复能否改进故障定位？一种统一调试方法。载于 ISSTA '20：第29届 ACM SIGSOFT国际软件测试与分析研讨会，线上会议，美国，2020年7月18日至22日，第75–87页。ACM，2020年。[38] Wei Ma、Thomas Laurent、Milos Ojdanic、Thierry Titcheu Chekam、Anthony Ventresque 和 Mike Papadakis。感知提交的变异测试。载于第36届 IEEE软件维护与演化国际会议论文集（ICSME），2020年。[39] Yu-Seung Ma、Jeff Offutt 和 Yong Rae Kwon。MuJava：自动化类变异系统。《软件测试、验证与可靠性》，15(2):97–133，2005年。[40] Micha¨el Marcozzi、S´ebastien Bardin、Nikolai Kosmatov、Mike Pa-padakis、Virgile Prevosto 和 Lo¨ıc Correnson。是时候清理测试目标了。载于 Michel Chaudron、Ivica Crnkovic、Marsha Chechik 和 Mark Harman（编），第40届国际软件工程会议论文集（ICSE 2018），瑞典哥德堡，2018年5月27日至6月3日，第456–467页。ACM，2018年。[41] Shabnam Mirshokraie、Ali Mesbah 和 Karthik Pattabiraman。面向 JavaScript Web应用的引导式变异测试。《IEEE软件工程汇刊》，41(5):429–444，2015年。[42] Roberto Natella、Domenico Cotroneo、Jo˜ao Dur˜aes 和 Henrique Madeira。论软件故障注入的故障代表性。《IEEE软件工程汇刊》，39(1):80–96，2013年。[43] A. Jefferson Offutt。软件测试耦合效应研究。《ACM软件工程方法学汇刊》，1(1):5–20，1992年。[44] A. Jefferson Offutt、Ammei Lee、Gregg Rothermel、Roland H. Untch 和 Christian Zapf。对充分变异算子的实验性判定。《ACM软件工程方法学汇刊》，5(2):99–118，1996年。[45] Milos Ojdanic、Aayush Garg、Ahmed Khanfir、Renzo Degiovanni、Mike Papadakis 和 Yves Le Traon。变异测试研究中人工故障与真实故障的句法相似性与语义相似性。arXiv预印本 arXiv:2112.14508，2021年。[46] Mike Papadakis、Thierry Titcheu Chekam 和 Yves Le Traon。变异体质量指标。载于2018年 IEEE软件测试、验证与确认国际会议研讨会，第32–39页。IEEE计算机学会，2018年。[47] Mike Papadakis、Marinos Kintis、Jie Zhang、Yue Jia、Yves Le Traon 和 Mark Harman。第六章——变异测试进展：分析与综述。《计算机进展》，112:275–378，2019年。[48] Mike Papadakis 和 Yves Le Traon。Metallaxis-FL：基于变异的故障定位。《软件测试、验证与可靠性》，25(5-7):605–628，2015年。[49] Mike Papadakis、Donghwan Shin、Shin Yoo 和 Doo-Hwan Bae。变异分数与真实故障检测相关吗？关于变异体与真实故障关系的大规模实证研究。载于第40届国际软件工程会议论文集（ICSE 2018），瑞典哥德堡，2018年5月27日至6月3日，第537–548页，2018年。[50] Jibesh Patra 和 Michael Pradel。语义缺陷植入：一种创建逼真缺陷的基于学习的方法。ESEC/FSE 2021，第906–918页，美国纽约，2021年。计算机协会。[51] Renaud Pawlak、Martin Monperrus、Nicolas Petitprez、Carlos Noguera 和 Lionel Seinturier。Spoon：用于实现 Java源代码分析与转换的库。《软件：实践与经验》，46:1155–1179，2015年。

[52] Ana Bel´en S´anchez、Pedro Delgado-P´erez、Inmaculada Medina-Bulo 和 Sergio Segura。真实环境中的变异测试：来自 GitHub 的发现。《*Empir. Softw. Eng.*》，27(6):132，2022年。[53] Chang-ai Sun、Feifei Xue、Huai Liu 和 Xiangyu Zhang。一种用于变异测试中变异体缩减的路径感知方法。《信息与软件技术》，81:65–81，2017年。[54] Michele Tufano、Cody Watson、Gabriele Bavota、Massimiliano Di Penta、Martin White 和 Denys Poshyvanyk。从缺陷修复中学习如何变异源代码。载于 *2019* 年 *IEEE* 软件维护与演化国际会议（*ICSME 2019*），第301–312页。IEEE，2019年。[55] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit, Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。注意力机制就是你所需要的一切。《神经信息处理系统进展》，30，2017。[56] Jeffrey M. Voas、Frank Charron、Gary McGraw、Keith W. Miller 和 Michael Friedman。预测“优秀”软件可能表现得有多糟糕。*IEEE Softw.*，14(4):73–83，1997。[57] Jeffrey M. Voas 和 Gary McGraw。《软件故障注入：为程序接种以抵御错误》。John Wiley & Sons, Inc.，美国，1997。[58] Lingming Zhang、Darko Marinov、Lu Zhang 和 Sarfraz Khurshid。回归变异测试。ISSTA 2012，第331–341页，美国纽约州纽约市，2012。计算机协会。